This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2023.3300151



# High Performance Delay Monitoring for SRv6 Based SD-WANs

Carmine Scarpitta*†, Giulio Sidoretti*†, Andrea Mayer*†, Stefano Salsano*†,
Ahmed Abdelsalam§, Clarence Filsfils§
*University of Rome Tor Vergata, †CNIT, §Cisco Systems

***Abstract*—Software-Defined Wide Area Networks (SD-WANs) are used to provide services to enterprises with geographically dispersed locations in a flexible and efficient way. We focus on SD-WAN services based on the Segment Routing over IPv6 (SRv6) technology. Performance Monitoring solutions are needed in SD-WANs to detect performance degradation and outages, and optimize network operations.**

**In this paper, we describe a high performance solution for end-to-end delay monitoring for SRv6 based SD-WAN services. The proposed solution leverages the Simple Two-way Active Measurement Protocol (STAMP) to monitor the delay of an SRv6 path between two nodes called STAMP Session-Sender and Session-Reflector. We describe three implementations of the STAMP Session-Sender and Session-Reflector for a Linux software router and compare their performance. In particular, two implementations are based on user space processing and one is based on eBPF. The results show that the eBPF-based implementation outperforms the user space implementations and has a negligible impact on the forwarding capacity of the Linux software router.**

***Index Terms*—SD-WAN, Software Defined WAN, Performance Measurement, Segment Routing, SRv6, Delay Monitoring.**

## I. Introduction

IT is common for enterprises to have multiple data centers and branch offices spread over large geographical areas. The reference scenario is shown in Fig. 1. Traditional Wide Area Networks for enterprises were based on static interconnections of remote sites. With the advent of cloud computing, many enterprises moved their applications to cloud systems. Traditional Wide Area Networks (WANs) started to exhibit limitations because they were not designed for cloud systems. First, traditional WANs do not provide the desired level of flexibility to users. Extending traditional WANs and adding new services require human intervention and are time consuming. Moreover, traditional WANs do not support cloud ecosystems natively. To provide access to cloud applications, traditional WANs typically require backhauling all traffic to a data center. Then, from the data center, the traffic is sent to the cloud. Software-Defined Wide Area Networking (SD-WAN) is a paradigm that aims at overcoming the limitations of traditional WANs. SD-WAN uses a software-defined approach to control the network and build the interconnections among the different locations. An SD-WAN builds interconnections among users and applications hosted on clouds or remote branches by leveraging any combination of transport services.

Over the years, many SD-WAN solutions have been proposed. Most SD-WAN solutions are commercial, such as Cisco

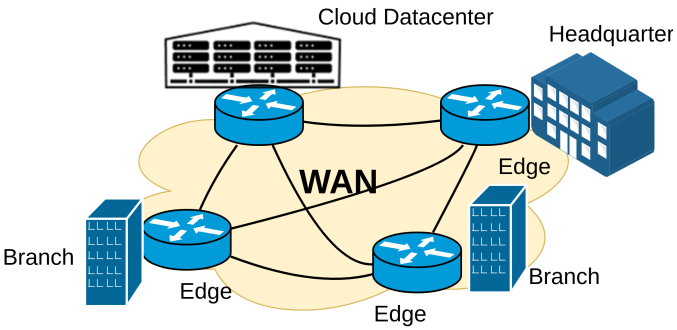



Fig. 1: Enterprise WAN reference scenario.

SD-WAN [1]. The Google B4 WAN [2] [3] is a proprietary SD-WAN solution that connects Google's data centers across the world. B4 relies on a hybrid Software Defined Networking (SDN) approach: the WAN sites are interconnected using traditional routing protocols, an SDN-based Traffic Engineering service runs on top of the network to maximize links utilization and perform load balancing, OpenFlow is used to control and program the switches. FlexiWAN [4] was the first open source solution. It uses Virtual Extensible LAN (VXLAN) [5] tunnels to establish the SD-WAN interconnections. In our previous work [6], we presented an open-source SD-WAN solution called EveryWAN, which is capable of using Segment Routing over IPv6 (SRv6) to establish the SD-WAN interconnections. To the best of our knowledge, EveryWAN is the first open-source solution to leverage SRv6 technology to create SD-WAN services. EveryWAN is based on Linux networking and can be deployed on software routers located at the edge of an SD-WAN.

In fact, software routers can play a role in SD-WAN scenarios, thanks to their flexibility, complementing hardware-based solutions. For example, they can be easily deployed in virtualized environments in cloud and data center scenarios as VNFs (Virtual Network Functions) as shown in Fig. 1. For this reason, we believe that it is fundamental to work on the design and implementation of open-source SD-WAN solutions suitable for software routers.

An important function to be executed in wide area networks is Performance Monitoring (PM). PM allows network operators to detect failures and outages and assess network performance. Effective network monitoring is essential, and new tools and protocols have been designed accordingly for

SDN-based networks [7]. Important application scenarios in which we can benefit from network monitoring are SLA-assurance in SD-WANs and Enterprise networks, Internet of Things (IoT) [8] and security [9].

In this paper, we focus on the delay monitoring of SRv6 networks. We consider a number of research and technological questions:

- Is it possible to design an effective solution for delay monitoring of SRv6 networks based on current Internet Engineering Task Force (IETF) standards and work-in-progress Internet drafts?
- Can we implement the solution in a working open-source prototype based on Linux software routers?
- What is the impact of the delay monitoring solutions on the forwarding capacity of software routers? Can we implement delay monitoring with negligible impact on forwarding performance?

The main novel contributions are as follows:

- Realization of a High Performance End-to-End Delay Monitoring solution for SRv6 networks compliant with available standards and Internet drafts;
- Design and implementation of a gRPC Remote Procedure Call (gRPC) [10] Southbound interface to control the SRv6 nodes;
- Implementation of two user space solutions and of a kernel solution based on eBPF (extended Berkeley Packet Filter) [11];
- Evaluation of the performance degradation introduced by the Delay Monitoring solution and comparison between the two user space and the eBPF-based implementations.

This paper is organized as follows. In Section II, we present an introduction to the SRv6 technology and its main use cases. Section III presents how SRv6 technology can be used to realize SD-WAN services. In Section IV, we introduce EveryWAN, the SD-WAN prototype that we have extended. Section V presents our Delay Monitoring solution. The implementations are discussed in Section VI. In Section VII, we show how we integrated our performance measurement solution in EveryWAN to measure the delay of Virtual Private Network (VPN) services. In Section VIII, we present a performance evaluation and comparison of the implementations. In Section IX we present the related works. Finally, Section X concludes the paper.

## II. SRv6 Technology

Segment Routing (SR) is a routing technology based on the *loose source routing* paradigm ([12], [13]). It allows a source node to steer a packet through a list of instructions called *segments*. A segment can represent a topological instruction (e.g., forward the packet via a specific nexthop) or a function to be applied to the packet (e.g., execute an operation on the packet). A segment is identified by an identifier known as *Segment ID (SID)*. The list of SIDs of a packet, called *Segment List* or *SID List*, is carried in the packet header. SR can be implemented using either Multiprotocol Label Switching (MPLS) or IPv6 as data plane technology. In MPLS Segment Routing (SR-MPLS) [14], the SIDs are encoded as MPLS labels. The Segment List is encoded as a stack of labels. In Segment Routing over IPv6 (SRv6), the SIDs are encoded as IPv6 addresses. The Segment List is carried in an IPv6 Extension Header called Segment Routing Header (SRH) [15]. A set of standardized SRv6 functions is presented in [16]. In this paper, we focus on SRv6.

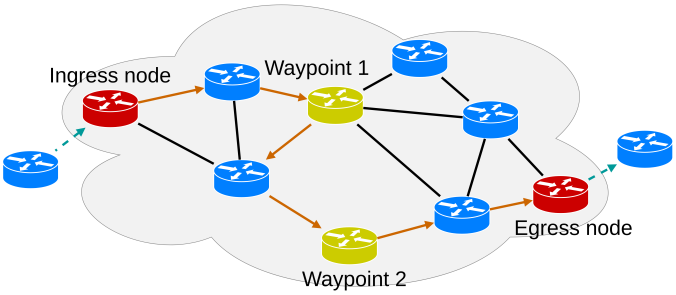



Fig. 2: SRv6 example network scenario.

Fig. 2 shows an example of an SRv6 network scenario. The gray cloud represents an SRv6 domain. An ingress node processes the packets entering the SRv6 domain and encapsulates each received packet in an outer IPv6 header with an SRH. In the example, the SRH carries a SID List containing three SIDs. The first two SIDs represent the two waypoints that the packets should traverse before reaching the destination. The ingress node forwards the encapsulated packets towards the first waypoint. The path to reach the waypoint is decided by the traditional routing protocol (e.g., IS-IS or OSPF). The first waypoint forwards the packet toward the second waypoint, which in turn forwards the packet toward the egress node, identified by the third SID in the segment list. The third SID is also used in the egress node to determine the operation to be performed. In this case, the egress node performs a decapsulation operation (i.e., removes the outer IPv6 header which contains the SRH) and forwards the packets to the destination. It is also possible to use two different SIDs instead of the third single one: a SID to reach the egress node and another SID to identify the operation to be performed, but this is less efficient as four SIDs instead of three would be carried in the Segment List.

The SRv6 technology has been proposed in the recent past and has raised great interest in academia and industry. Since then, its development has progressed very rapidly. Today, SRv6 is supported in many hardware deployments [17] and software routers such as the Linux kernel and the Vector Packet Processor (VPP) [18]. The Linux kernel has supported SRv6 packet generation and forwarding capabilities since version 4.10 (released in February 2017). Later, it has been extended to support many of the SRv6 behaviors described in [16].

SRv6 enables many use cases such as overlay Virtual Private Networks (VPNs) [19], Traffic Engineering [20], Fast Rerouting [20], and Service Function Chaining (SFC) [21]. An overview of SRv6 implementation and deployment status is available at [22] and [17]. The Research on Open SRv6 Ecosystem (ROSE) project [23] aims to build a Linux-based Open Ecosystem for SRv6. It tackles multiple aspects of the

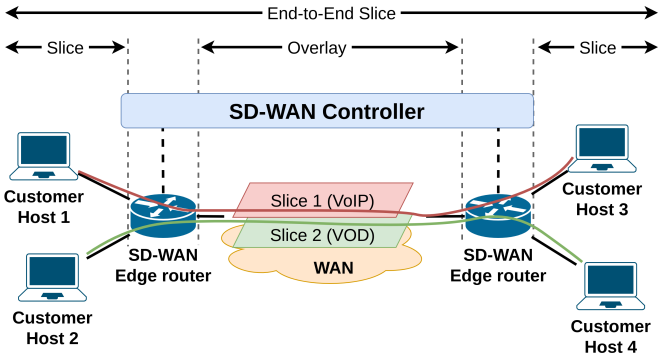



Fig. 3: SD-WAN service scenario.

SRv6 technology, including the Data Plane, Control Plane, SRv6 host networking stack, integration with applications, and integration with Cloud/Data Center Infrastructures. ROSE comprises several sub-projects which are the foundation of the work presented here.

## III. SD-WAN Services Based on SRv6

In the UCSS project[1] (*User Controlled SD-WAN Services with Performance Monitoring over GÉANT)* [25] we designed, implemented and deployed SD-WAN services based on the SRv6 technology.

An SD-WAN can offer different services. We focus on the *Network Slicing* service. The reference scenario is shown in Fig. 3. Network Slicing allows customers to create different logical instances of virtual networks over the WAN connections. It is also possible to use different WAN providers for the different slices. In this way, multiple applications can run in isolation over the WAN and different service levels can be used for the different slices. Among the different types of slicing, we focus on *Routed End-to-End Slices*. A Routed End-to-End Slice is an implementation of a Layer 3 VPN (L3VPN), in which the devices attached to the SD-WAN Edges belong to different broadcast domains. The SD-WAN Edge routers act as gateways to route traffic between these broadcast domains.

In our terminology, a *Slice* (or *Local Slice*) is a portion of the customer network where users or applications are located. Each Local Slice is terminated in an SD-WAN Edge router. The SD-WAN Edge router forwards the traffic of the connected Local Slice to an egress SD-WAN Edge router. The interconnections between two different SD-WAN Edge routers are realized by using a set of *Tunnels* (also called *Overlays*). The Overlay, together with the Local Slices, forms the so-called *End-to-End Slice* (*E2E Slice*). Several technologies can be used to realize an Overlay. We focus on SRv6-based Overlays. Fig. 4 shows the reference scenario for the SD-WAN service based on SRv6 technology. An ingress SD-WAN Edge router receives IP packets from a customer source host. It classifies and associates each incoming packet with a specific End-to-End Network Slice according to various criteria, such as the incoming interface, the source IP address, or the protocol. After the classification, the ingress SD-WAN Edge router performs a lookup in its Forwarding Information Base (FIB) to discover the SD-WAN egress Edge router attached to the destination host. Then, the ingress SD-WAN Edge router applies the *H.Encaps* behavior described in [16] to the packet. This behavior steers the packet into an SRv6 Policy. Steering is realized by encapsulating the IP packet into an outer IPv6 header that contains an SRH. The SRH carries two SIDs. The first SID represents an instruction to deliver the packet to the egress SD-WAN Edge router. The second SID is an *End.DT6* instruction. End.DT6 forces the egress router to strip the outer IPv6+SRH header and deliver the original packet to the correct Slice.

In SD-WAN solutions, the SD-WAN Edge routers are deployed in all the locations where the SD-WAN interconnections need to be established. An *SD-WAN controller* manages and programs the SD-WAN Edge routers. Depending on the location and the characteristics of the SD-WAN Edge routers, three scenarios are possible:

1) the SD-WAN Edge routers are located within the provider network and are under the network operator's control;
2) the SD-WAN Edge routers are outside the provider network, and they have no control over the transport services;
3) the SD-WAN Edge routers are outside the provider network but can interact with the provider network to deploy the SD-WAN services.

We focus on Scenario 2. The SRv6-based SD-WAN services were deployed in scenarios where SD-WAN Edge routers do not interact with the provider networks. We have deployed several SD-WAN Edge routers as Virtual Machines (VMs) across Europe. These SD-WAN Edge routers were located in different kinds of networks, like university campus networks, NRENs (National Research and Education Networks), and commercial provider networks. We analyzed and classified the IPv6/SRv6 connectivity between these VMs and introduced the concept of *SRv6 Transparency*. SRv6 Transparency is the ability of an IPv6 network to carry SRv6 traffic. Several factors can reduce the SRv6 Transparency of a network, such as firewalls that block IPv6 packets carrying an SRH. We found different SRv6 Transparency levels in the networks that we considered. We have shown that it is possible to configure the SRv6-based SD-WAN services, taking into account the SRv6 Transparency level of the network providing IPv6 connectivity and we have practically deployed SD-WAN services across operational networks over the Internet. An in-depth discussion of the SRv6 Transparency problem and the configuration of SRv6-based SD-WAN services can be found in the UCSS report [25].

A great advantage of SRv6 technology is that minimal extensions are needed to support scenarios in which the SD-WAN router is part of the provider network or can interact with it. For example, a useful service is an overlay with traffic engineering in the underlay transport network. If the SD-WAN router is under the control of the transport network operator, the SD-WAN controller can provide an extended SID list that at the same time implements the SD-WAN service and provides control over the path in the underlay network

[1]part of the GÉANT Innovation Programme [24]

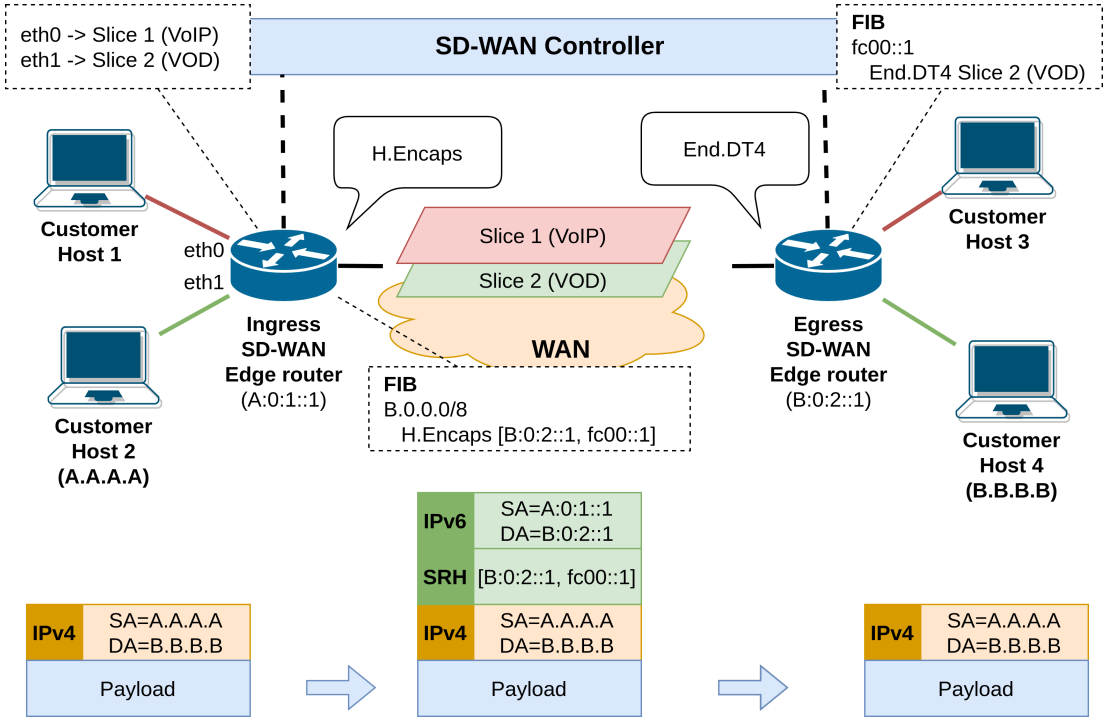



Fig. 4: SD-WAN service scenario based on SRv6.

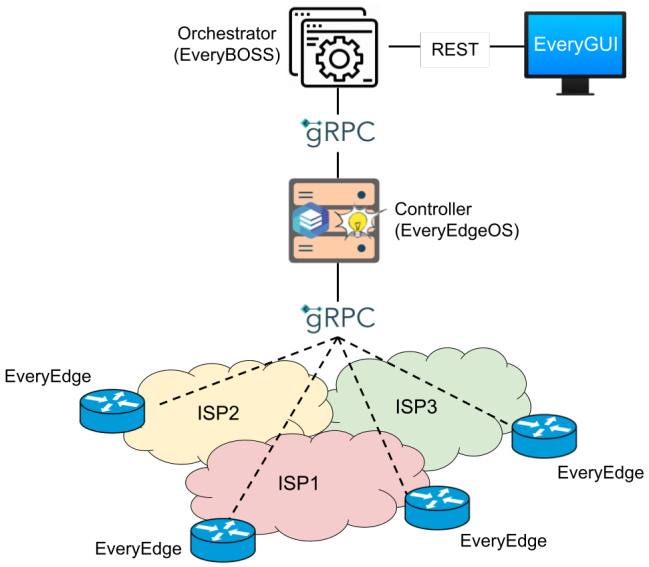



Fig. 5: EveryWAN Architecture.

according to traffic engineering considerations.

## IV. The EveryWAN Architecture

*EveryWAN* [6] is an open-source SD-WAN prototype based on Linux networking. Fig. 5 shows the EveryWAN architecture. At the lowest level, we have the SD-WAN Edge routers called *EveryEdge routers*. The EveryEdge routers take care of the interconnections among all the sites. EveryEdge routers can be deployed as Virtual Network Functions (VNFs) over a Linux OS in the sites to be interconnected. An SD-WAN Controller, called *EveryEdgeOS*, manages all the EveryEdge routers through an API based on the gRPC Remote Procedure Call (gRPC) protocol. gRPC [10] is a high-performance RPC (Remote Procedure Call) framework that was initially developed by Google and is now maintained and supported by an active community of developers. EveryEdgeOS deals with many configuration and management aspects of the EveryEdge routers, ranging from their initial registration, authentication, and configuration to the activation of the policies that implement the SD-WAN services. On top of the controller, there is an SD-WAN Orchestrator named *EveryBOSS*, which automates the deployment of the EveryEdge routers and SD-WAN services. The orchestrator also offers a GUI that allows the customers to configure the EveryEdge routers and manage the SD-WAN services. The EveryEdgeOS and the EveryBOSS orchestrator can run either in a self-managed private cloud or in a public cloud.

The EveryEdge router comprises several open-source components installed on a general-purpose Linux distribution (e.g., Ubuntu Server). It uses Linux networking capabilities to forward the traffic. A component called *EveryEdgeManager* offers a Southbound API that is used by the EveryEdgeOS controller to program and configure the router. Through the Southboud API, the controller can send commands to the EveryEdge router (e.g., install a specific route or set the IP address of a network interface). The EveryEdgeManager translates the received commands into lower-level actions. Then, it sends these actions to the Linux kernel using the open-source pyroute2 [26] library. Pyroute2 is a Python package that provides a programming interface for network configuration and management on Linux systems. Pyroute uses the Netlink

protocol, which is a mechanism for communication between the kernel and user-space processes. A detailed description of the EveryEdge router architecture can be found in [6].

The main service offered by EveryWAN is Network Slicing (described in Section III), which allows customers to create End-to-End Slices among the remote sites. The EveryEdge router receives ingress IP packets over the customer-facing interfaces, i.e., the Local interfaces (LAN). It classifies and associates each packet with a particular End-to-End Network Slice. To perform the classification, the EveryEdge leverages the *Virtual Routing and Forwarding* (VRF) technology offered by the Linux kernel. VRFs provide the ability to create isolated virtual routing and forwarding domains. Each VRF serves a particular slice. Each customer-facing interface in the EveryEdge router is mapped to a slice and enslaved to the VRF that serves that slice. Based on the destination IP address, the EveryEdge router forwards the packets associated with a slice to the remote EveryEdge routers over the WAN interfaces.

A transport technology ensures that the network delivers the packets to the remote EveryEdge router. EveryWAN supports two transport technologies: VXLAN [5] and SRv6. In this work, we only consider SRv6. To transmit the packets using SRv6, the EveryEdge routers use the *H.Encaps* and the *End.DT4/End.DT6* behaviors as depicted in Fig. 4 and discussed in the previous section.

A detailed description of the EveryWAN architecture can be found in the white paper [27].

## V. STAMP Delay Monitoring for SRv6

In this section, we present the proposed End-to-End Delay Monitoring solution for SRv6 networks based on the Simple Two-Way Active Measurement Protocol (STAMP) [28]. STAMP enables the measurement of several performance metrics, including *packet loss*, *delay*, and *jitter*. It supports both one-way and round-trip measurements in IP networks. RFC 8762 [28] defines the base functionalities of STAMP and describes the format of the packets that collect and carry measurement data. RFC 8972 [29] introduces the *STAMP Session IDentifier* (*SSID*) and defines optional STAMP extensions that enhance the STAMP base functions. The drafts [30] and [31] present general guidelines for measuring various performance metrics in SR networks using STAMP. In the following subsections, we present a solution based on STAMP to measure the end-to-end delay of SRv6 paths. Note that, in general, the applicability of the STAMP protocol goes beyond Segment Routing and SRv6 as it can be integrated into monitoring tools and applications to evaluate the performance metric of any kind of network.

Fig. 6 shows our STAMP reference scenario. We use a *STAMP Session* to measure the end-to-end delay on an SRv6 path between two nodes called *STAMP Session-Sender* and *Session-Reflector*. For delay measurements to be meaningful, the Session-Sender and Session-Reflector clocks must be synchronized[2]. RFC 8762 does not envisage any particular approach to configure and manage the STAMP Session-Sender, Session-Reflector, and the STAMP Session, which can be achieved in different ways, such as using a Command Line Interface (CLI) or an SDN controller. The proposed solution leverages an SDN controller to manage the STAMP Session and configure the STAMP Session-Sender and Session-Reflector. The public documentation of our delay monitoring solution with links to code repositories is available in [34].

[2]Clock synchronization mechanisms are out of scope for this paper, we assume that the clocks are synchronized. Depending on the required precision, software synchronization mechanisms like NTP [32] or hardware assisted mechanisms (typically based on GPS [33]) can be used.

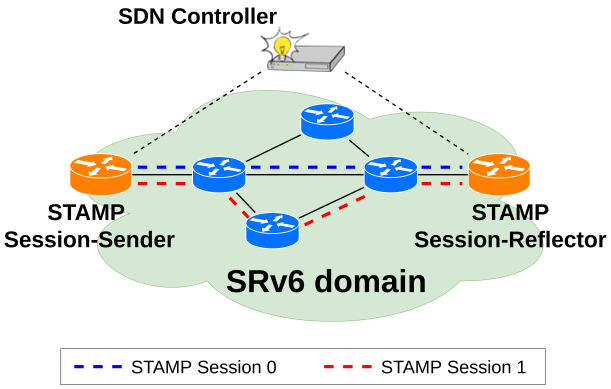
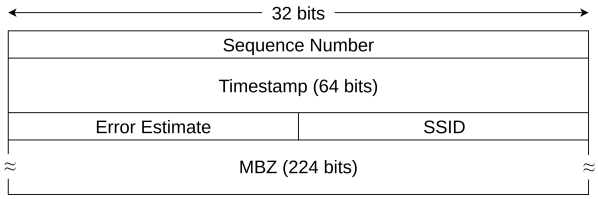
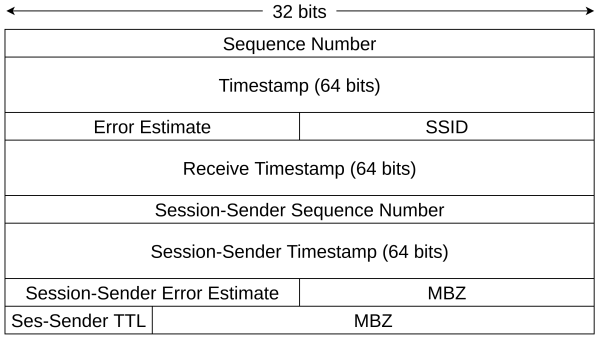



Fig. 6: STAMP reference scenario.




(a) STAMP Session-Sender Test Packet.




(b) STAMP Session-Reflector Test Packet.

Fig. 7: STAMP Test Packets defined in [29].

### A. Data Plane Protocol

A STAMP session measures the end-to-end delay on a given SRv6 path between two nodes, the STAMP Session-Sender and Session-Reflector. A STAMP session consists of a bidirectional packet exchange between the STAMP Session-Sender and the Session-Reflector. Each STAMP session is identified by a unique 16-bit nonzero unsigned integer called *STAMP Session IDentifier* (*SSID*).

The STAMP Session-Sender transmits a STAMP Session-Sender test packet to the STAMP Session-Reflector. The test packet is an IPv6/UDP packet sent to the STAMP UDP port of the Session-Reflector. By default, the STAMP Session-Reflector uses the UDP port 862. The SDN controller can set a different port during the configuration of the STAMP Session-Reflector. The STAMP Session-Sender test packet is transmitted on the same path as the data traffic flow under measurement to measure the delay experienced by the data traffic flow. To enforce the path, the STAMP Session-Sender adds an SRH to the IPv6 header. The SRH contains the SID List that encodes the path under measurement from the STAMP Session-Sender to the Session-Reflector. The test packet carries the payload shown in Fig. 7a. The *Sequence Number* field contains a 32-bit unsigned integer. It starts at zero and is incremented by one with each sent packet. The *Timestamp* field carries the time when the Session-Sender sent the test packet. In the rest of this section, we refer to this timestamp as $T_1$ (see Fig. 8). RFC 8762 specifies two different timestamp formats: Network Time Protocol (NTP) [32] and the IEEE 1588v2 Precision Time Protocol (PTP) [35], both using 64 bits. By default, the STAMP Session-Sender uses NTP as timestamp format, as specified in RFC 8762. The SDN controller can select a different timestamp format during the STAMP Session-Sender or STAMP Session configuration.

The *SSID* (STAMP Session IDentifier) field contains the SSID of the STAMP Session to which the test packet belongs. It associates the STAMP Session-Sender test packet with the corresponding STAMP Session. The remaining 28 bytes (224 bits) are set to zero (*Must-Be-Zero* or *MBZ* field). The content of STAMP Session-Reflector test packet is larger than the content of a STAMP Session-Sender test packet. The MBZ field makes the size of the Session-Sender test packet equal to the size of the Session-Reflector test packet.

Following the SRv6 path under measurement, the test packet is delivered to the Session-Reflector. The Session-Reflector receives the STAMP Session-Sender test packet and verifies it. If the packet is valid and the SSID corresponds to an active STAMP Session, the Session-Reflector creates and sends a STAMP Session-Reflector test packet to the STAMP UDP port of the Session-Sender. The STAMP Session-Reflector test packet carries the payload depicted in Fig. 7b. Bytes 24-33 contain an exact copy of the STAMP Session-Sender test packet. The *Sequence Number* field contains a 32-bit unsigned integer. The STAMP Session-Reflector can work in two modes: i) *stateless* mode; ii) *stateful* mode. In the stateless mode, the STAMP Session-Reflector reuses the same Sequence Number value contained in the STAMP Session-Sender test packet. In the stateful mode, the STAMP Session-Reflector maintains a counter for the transmitted packets. The *Receive Timestamp* field contains the time when the Session-Reflector received the Session-Sender test packet, denoted as $T_2$ (see Fig. 8). The *Timestamp* field contains the time when the Session-Reflector starts transmitting the Session-Reflector test packet, denoted as $T_3$. The *SSID* 16-bit field contains the STAMP Session IDentifier and allows the STAMP Session-Sender to associate the received STAMP Session-Reflector packets with the correct STAMP Session. The *Session-Sender*

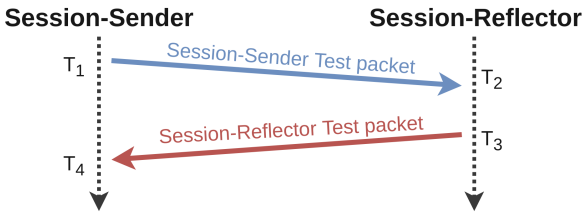



Fig. 8: STAMP time diagram.

*TTL* is a copy of the Hop Limit field of the IPv6 header contained in the received STAMP Session-Sender test packet. The *MBZ* fields are used to achieve an alignment on a four-byte boundary. The Session-Reflector test packet is transmitted on the same path as the data traffic flow under measurement to measure the delay experienced by the data traffic flow. This can be the same path as the Session-Sender test packet or a different path. The draft [30] defines a TLV called *Return Path TLV* that allows the Session-Sender to request the Session-Reflector to transmit the Session-Reflector test packet on a specific path. However, we do not use the Return Path TLV in our solution. We leverage the SDN controller to set up the return path as part of the STAMP Session configuration. Before sending the STAMP Session-Reflector test packet, the Session-Reflector adds an SRH to the IPv6 header to enforce the return path. The SRH contains a SID List that encodes the path under measurement from the STAMP Session-Reflector to the Session-Sender.

Following the path specified in the SID List, the STAMP Session-Reflector test packet is delivered to the Session-Sender. The Session-Sender verifies the packet and validates the SSID. If the SSID corresponds to an active STAMP Session, it generates a new timestamp $T_4$, which is the time when the Session-Sender received the Session-Reflector test packet. The Session-Sender collects the three timestamps from the session reflector test packet and adds $T_4$ creating a measurement record ($T_1$, $T_2$, $T_3$, $T_4$) that is stored locally. The generated records need to be sent to the SDN controller for post-processing, as it will be discussed later. Considering its role in the processing of the STAMP test packets coming back from the Session-Reflector, we can refer to the Session-Sender as the final *Collector* of the STAMP test packets.

### B. Configuration and Management

We have defined the API offered by the STAMP Session-Sender and by the STAMP Session-Reflector to the SDN controller for the configuration of the STAMP measurement service. The configuration involves setting various parameters, including the STAMP UDP port, the network interfaces on which the STAMP Session-Sender/Session-Reflector expects to receive the STAMP Test packets, and the source IPv6 address to be used in the STAMP Test packets. The controller can also create and manage the STAMP Sessions using the API exposed by the STAMP Session-Sender/Session-Reflector. In particular, to create a STAMP Session, the SDN controller must provide the following parameters: 1) the SSID of the

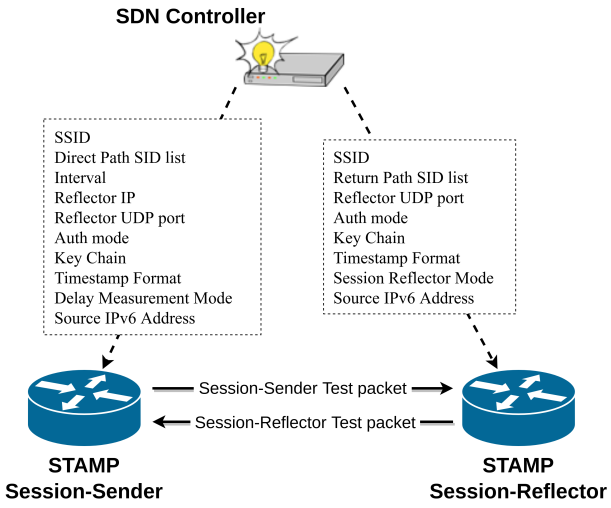



Fig. 9: STAMP control protocol.

STAMP Session; 2) the SID List of the path under measurement; 3) the interval between two consecutive STAMP Test packets; 4) the source IPv6 address of the STAMP Test packets; 5) the authentication mode (i.e., unauthenticated or authenticated); 6) the timestamp format (i.e., NTP or PTPv2); 7) the delay measurement mode (i.e., one-way or two-way); 8) the IP address of the STAMP Session-Reflector; 9) the STAMP UDP port of the STAMP Session-Sender and Session-Reflector; 10) the Session-Reflector mode (i.e., stateful or stateless). Fig. 9 shows the interaction of the SDN controller with the STAMP Session-Sender and Session-Reflector required to create a STAMP Session.

### C. Data Collection

The STAMP Session-Sender and STAMP Session-Reflector exchange STAMP Test packets containing the timestamps required to compute the delay. The STAMP Session-Sender collects all the timestamps. The SDN controller can interact with the Session-Sender to fetch the timestamps. In general, there are two approaches the SDN controller can use to fetch the timestamps: polling mode and notification mode. In polling mode, the controller periodically polls the Session-Sender to gather the collected timestamps. In notification mode, the Session-Sender will "push" the information toward the SDN controller, either by sending the single measurement records or aggregating a set of measurement records in a single notification. In our solution we have implemented the polling mode.

When the measurements records are available to the SDN controller, it can compute the delay of the direct path $d_d$ (i.e., the path from the Session-Sender to the Session-Reflector) and return path $d_r$ (i.e., the path from the Session-Reflector to the Session-Sender):

$$d_d = T_2 - T_1 \tag{1}$$

$$d_r = T_4 - T_3 \tag{2}$$

where $T_1$, $T_2$, $T_3$, and $T_4$ are the four timestamps defined in Section V-A, $d_d$ and $d_r$ are the delay of the direct path and return path, respectively. Of course, the clocks of the Session-Sender and of the Session-Receiver must be synchronized and the accuracy of this delay estimates $d_d$ and $d_r$ depends on the accuracy of the clock synchronization.

## VI. STAMP for SRv6: Router Implementations

We have realized an open source prototype of the proposed STAMP for SRv6 solution, see [36]. In this section we describe the implementation of the router functionality (Session-Sender and Session-Reflector), in section VII we focus on the SDN Controller and Orchestrator.

The main Data Plane tasks of the Session-Sender (described in Section V) are the following: i) generate and send STAMP Session-Sender Test packets to the STAMP Session-Reflector; ii) receive STAMP Session-Reflector Test packets from the Session-Reflector and collect the timestamps. Concerning the STAMP Session-Reflector, its main Data Plane tasks are: i) receive STAMP Session-Sender Test packets from the Session-Sender; ii) send a STAMP Session-Reflector Test packet to the Session-Sender for each received STAMP Test packet. The Session-Reflector is implemented in its stateless version. In the Control Plane, both the Session-Sender and the Session-Reflector interact with the SDN controller by offering an API (see subsection VI-A).

As for the Data Plane, we have implemented three versions of the Session-Sender and Session-Reflector with the goal of improving their performance: two User Space implementations (referred to as *basic* and *optimized*, see subsection VI-B) and a Kernel Space implementation based on the extended Berkeley Packet Filter (eBPF) framework [11], see subsection VI-C. We evaluate and compare the performance of the different implementations in Section VIII.

### A. Control Plane functionalities

Both the STAMP Session-Sender and Session-Reflector expose a Southbound API that allows an SDN controller to create/start/stop/destroy a STAMP Session and fetch the results of a STAMP Session. This API follows the design ideas discussed in Section V. We decided to extend the Southbound API proposed in [37], based on the gRPC protocol [10]. The implementation of our Southbound interface is open-source and available at [36]. The Southbound API supports the following operations:

- `Init` provides the global configuration parameters (i.e., the parameters common to all the STAMP Sessions) to the STAMP Session-Sender and Session-Reflector.
- `Reset` resets the configuration parameters and stops the packet sniffer.
- `CreateStampSession` prepares the STAMP Session-Sender/Session-Reflector to run a STAMP Session and send/receive the STAMP Test packets. In the Sesssion-Sender, a queue is allocated to store the received measurement results.
- `StartStampSession` and `StopStampSession` take care of starting and stopping a STAMP Session,

respectively. When a session is started in the Session-Sender, a thread is activated that periodically sends STAMP Session-Sender Test packets to the Session-Reflector.
- `DestroyStampSession` removes a STAMP Session and deallocates all the related data structures.
- `GetStampSessionResults` allows the controller to fetch the measurement results (i.e., the timestamps) collected by the STAMP Session-Sender. This RPC is supported only by the Session-Sender as the Session-Reflector does not collect any information during the STAMP Session.

A more detailed description of the Southbound interface can be found in [38].

### B. User Space Implementations for Data Plane

In this subsection, we describe our user-space implementations of the STAMP Session-Sender and Session-Reflector, compliant with RFC 8762 [28], RFC 8972 [29], and draft [31]. The implementations are based on Scapy [39], a packet manipulation library written in Python, Scapy provides a programming abstraction to generate and send network packets as well as to receive and decode them. Several protocols are already supported by Scapy and it can be extended to support new protocols. We have developed a first STAMP implementation (referred to as *basic*) and then designed an improved version (referred to as *optimized*). Hereafter, we first describe the *basic* Scapy user space implementation and then we discuss how we have tackled its performance issues with the *optimized* implementation. Our implementations are available as open-source at [36].

The Session-Sender and Session-Reflector leverage the Scapy library to generate the STAMP Test packets. When we started our work, the latest release of Scapy (version 2.4.5) did not implement the RFC 8762 (STAMP). Scapy modular design allows developers to define new protocol layers easily. We have added the support for both STAMP Session-Sender and STAMP Session-Reflector Test packets in unauthenticated mode. Our contribution has been accepted and merged in the mainstream distribution of Scapy, adding the support of the STAMP protocol. Both the Session-Sender Test packet and Session-Reflector Test packet are compliant with the formats defined in RFC 8962 and described in Section V. The STAMP Test packets contain the timestamps used to compute the delay. As discussed in Section V, STAMP can support two timestamp formats: NTP and PTPv2. Our current implementation only supports NTP timestamps.

After generating the STAMP Test packets, the Session-Sender and the Session-Reflector use the Scapy library to send the packets on the outgoing network interface. In particular, before sending a STAMP Test packet, the Session-Sender adds an UDP header and an IPv6+SRH header to the packet. The UDP header contains the STAMP port of the Session-Reflector as destination port. The SRH contains the Segment List of the path under measurement (i.e., the path from the Session-Sender to the Session-Reflector). The Session-Reflector performs the specular operations adding the proper UDP header and IPv6+SRH header to send the packet to the Session-Sender. Then, the Session-Sender and the Session-Reflector pass the packet to an `L3RawSocket6`. The `L3RawSocket6` is a Scapy socket built on top of a AF_INET6/SOCK_RAW Linux socket. The Linux kernel adds a Layer 2 header and sends the packet to the destination (i.e., the Session-Reflector or the Session-Sender) according to the usual L2/L3 rules.

Both the Session-Sender and the Session-Reflector need to process the incoming STAMP Test packets. The Session-Reflector receives the STAMP Session-Sender Test packets from the Session-Sender and it has to reply to these packet by adding the proper timestamps. The STAMP Session-Sender receives STAMP Session-Reflector Test packets from the Session-Reflector and processes them, acting as a measurement data collector.

The Session-Sender and the Session-Reflector run a dedicated thread to capture, validate and process the STAMP Session Test packets. To capture the incoming STAMP Test packets, the *basic* implementation of Session-Sender uses a Scapy AsyncSniffer. The AsyncSniffer captures all the incoming packets received on a given interface and passes the captured packets to a user space callback named `stamp_reply_packet_received`. This callback drops any non-STAMP Test packet and processes only the valid STAMP Test packets. Since `stamp_reply_packet_received` operates in user space, calling it for each received packet can have a big impact on the CPU usage. In order to reduce the impact on the CPU usage, it is important to reduce the number of packets processed by the `stamp_reply_packet_received`. In our implementation, we attach a BPF filter to the AsyncSniffer. This filter allows the AsyncSniffer to capture only the STAMP Test packets by filtering non-STAMP Test packets at kernel level. Thus, `stamp_reply_packet_received` is invoked only when a STAMP Test packet is received. For each captured STAMP Test packet, `stamp_reply_packet_received` performs several validation checks. If the packet passes all the validation checks, the Session-Sender extracts the timestamps and collects them in a FIFO queue. The controller periodically can send a `GetStampSessionResults` command to fetch the latest results from the Session-Sender. The results are kept in the FIFO queue until they are fetched, then they are permanently removed from the queue.

The *basic* implementation of the Session-Reflector performs similar operations to capture STAMP Session-Sender Test packets and send STAMP Session-Reflector Test packets.

During our performance evaluation, we found that the *basic* Scapy solution exhibited very poor performance.

As explained previously, the *basic* implementation relies on the Scapy AsyncSniffer to capture the STAMP Test packets. AsyncSniffer is implemented using a Linux AF_PACKET/SOCK_RAW socket. An AF_PACKET/SOCK_RAW socket captures all the packets received on a given interface. The capture process of a plain AF_PACKET socket is very inefficient, because it uses very limited buffers and requires a system call to capture each packet.

The second bottleneck of the *basic* implementation is related to the process of building and dissecting the STAMP Test packets. The Session-Sender periodically generates and sends STAMP Test packets to the Session-Reflector. Generating a STAMP Test packet involves several operations, such as building each layer, filling each header with the proper information, stick all the layers together, and computing the checksum. We found that repeating this sequence of operations for building each packet to be transmitted is very expensive.

Therefore, we designed an improved implementation of the STAMP Session-Sender that mitigates the above described performance issues. We refer to this improved version as *optimized*. This implementation uses the PACKET_MMAP [40] socket option. PACKET_MMAP improves the capture process by using a circular buffer mapped in user space that can be used to send and receive packets. This buffer is shared between the kernel and our user space application. A shared buffer between the kernel and the user also has the advantage of minimizing packet copies. When a packet arrives, the kernel stores the packet in the buffer. Since the buffer is shared between the kernel and our user space STAMP application, the application can read the packet without issuing any system call.

In order to fix the inefficiencies in the sending procedures, we observed that packets sent in the context of a STAMP Session are very similar to each other. Most of the packet fields are equal for each packet in a STAMP Session. These fields include the SSID, the Segment List, the source and destination IP addresses, and the UDP ports. Few fields need to be changed, such as the timestamp fields and the sequence number contained in the STAMP Test packets. Instead of generating a new packet for each STAMP packet to be sent, the *optimized* implementation of the Session-Sender allocates a STAMP Session-Sender Test packet when the STAMP Session is created (`CreateStampSession` operation). When a new packet needs to be sent, the Session-Sender only changes the variable fields of the packet (e.g., the timestamps and the sequence number). Then it computes the UDP checksum and sends the packet to the Session-Reflector. In this way, we avoid the overhead related to generating a new STAMP Test packet from scratch. To further improve performance, we save the STAMP Test packet as a bytes array instead of a Python object. In this way, we avoid the overhead due to converting the packet from Python representation to a bytes array before sending it on the network. We also optimized the logic used to parse the received packets. For each received STAMP Session-Reflector Test packet, Scapy performs the so-called packet dissection, i.e., it reads the bytes of the packet and builds a Python object to represent the packet. Then, it collects the timestamps from the packet. In the *optimized* solution we bypassed the Scapy dissector and we extract the timestamps directly from the bytes representation of the packets.

As for the Session-Reflector, its *optimized* implementation improves the efficiency of the *basic* version using the same approaches that we have discussed for the Session-Sender.

The optimized versions of the Session-Sender and Session-Reflector STAMP implementation have been integrated in the EveryWAN prototype as described in Sec. VII.

### C. eBPF Implementation for Data Plane

eBPF [11] is a Linux technology that can be used to accelerate network packet processing. With eBPF it is possible to deploy programs in kernel space and in a sandboxed environment, without having to write ad-hoc kernel modules or change the kernel code. eBPF can offer high performance to specific packet processing tasks. We designed and implemented a proof-of-concept eBPF implementation with the goal to assess its performance.

Our eBPF deployment is based on the HIKe / eCLAT [41] [42] framework. HIKe (Heal, Improve and desKill eBPF) is a virtual machine abstraction for eBPF. It makes it possible to chain multiple eBPF programs in a larger and more complex program. eCLAT (eBPF Chains Language And Toolset) is a python-like language and programming framework. Its scripts compile to HIKe chains, providing a high-level, simpler language that can be used to compose complex eBPF programs in a modular fashion.

**Algorithm 1** HIKe chain high level structure for STAMP Session-Reflector.

```
if packet is STAMP then
    process headers for layers 2, 3, 4
    compute UDP checksum
    cross connect to layer 2 interface
else
    pass packet to kernel
end if
```

The high-level pseudocode 1 shows the structure of the HIKe chain for the STAMP Session-Reflector. The chain is attached to the eXpress Data Path (XDP) hook on the desired interface and the entire processing is performed without letting the packet enter the Linux kernel networking stack. The first eBPF program filters only STAMP Test packets, everything else is passed to the kernel without further processing. The chain then manipulates the STAMP fields adding the new timestamps. Then, the address/port fields in MAC, IPv6 and UDP headers are changed before forwarding the packet. Lastly, the UDP checksum is recalculated and the packet is forwarded on the desired interface.

The Collector implementation is simpler because the packet does not need to be forwarded. The chain comprises a filter so that only STAMP packets are processed, while other packets are sent to the kernel networking stack. Then we have the actual Collector eBPF program. It parses the STAMP payload of the packet and extracts the timestamps. The extracted timestamp records are written inside an eBPF map, accessible from the userspace, so that it is possible to read the measurements.

The code for the eBPF implementation can be found in the repository [43]. The deployment and configuration of the eBPF implementation is not integrated in the EveryWAN prototype. The configuration is performed manually as the eBPF proof-of-concept implementation is only used for the performance experiments described in section VIII.

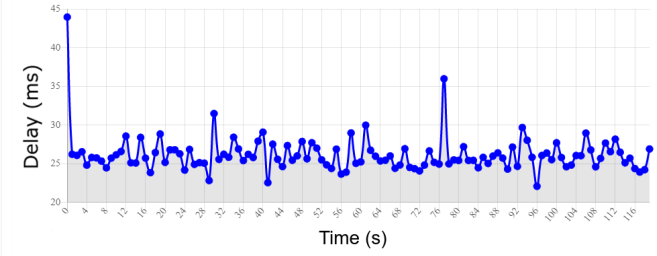



Fig. 10: Delay monitoring through the EveryWAN GUI.

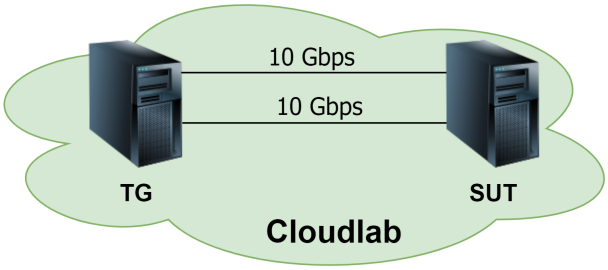



Fig. 11: Performance Evaluation Testbed on Cloudlab.

## VII. Delay Monitoring through EveryWAN Controller

We integrated the delay monitoring in the EveryWAN prototype. As explained in the EveryWAN white paper [27], the EveryEdgeOS controller exposes a Northbound API that allows users to configure the EveryEdge routers and deploy the SD-WAN services. We integrated the STAMP-based delay monitoring capabilities into EveryEdge routers and extended the EveryEdgeOS controller to support STAMP operations. We also extended the Northbound API to offer the basic operations to create, control, and destroy the STAMP Sessions. Furthermore, we added a section to EveryGUI where users can monitor in real time the delay of the deployed SRv6-based VPNs. The result of a measurement session presented on EveryGUI is shown in Fig. 10. In the x-axis there is the time in which each measure is performed. Delays are reported on the y-axis. The observed variability of the delay is due to the random fluctuation of background traffic. A walkthrough documentation showing the use of delay monitoring in EveryWAN is available in [34].

In addition to the instant delays, the controller also computes the average delay for both the direct and return paths. The average delay is updated using the *Welford online algorithm* [44] [45] whenever new $d_{d,new}$ and $d_{r,new}$ values are available:

$$d_{d,avg} = d_{d,avg} + \frac{d_{d,new} - d_{d,avg}}{N} \tag{3}$$

$$d_{r,avg} = d_{r,avg} + \frac{d_{r,new} - d_{r,avg}}{N} \tag{4}$$

where $d_{d,avg}$ is the average delay of the direct path, $d_{r,avg}$ is the average delay of the return path, N is the number of collected delays, and $d_{d,new}$ and $d_{r,new}$ are the new delay values of the direct path and return path, respectively.

## VIII. Experiments and Results

In this section, we describe the testbed and the methodology used to assess the performance of our STAMP implementations, and we present a comparison between the different implementations.

### A. Testbed and Performance Evaluation Methodology

To evaluate the performance of our three implementations, we have deployed a testbed according to RFC 2544 [46], which provides a methodology to benchmark network devices. The testbed (shown in Fig. 11) includes two nodes: *Traffic Generator (TG)* and *System Under Test (SUT)*. We have deployed our testbed in the Wisconsin cluster of CloudLab [47], a platform dedicated to scientific research on the future of cloud computing. The testbed nodes (TG and SUT) are bare metal servers equipped with two Intel E5-2630 v3 processors with 16 cores (hyper-threaded) clocked at 2.40GHz, 128 GB of RAM, and two Intel 82599ES 10-Gigabit network interface cards. The TG and SUT nodes are physically connected to the same switch. The two NICs ensure back-to-back connectivity between the two nodes. The logical topology is shown in Fig. 11. On the TG node, we installed TRex [48], an open source traffic generator powered by DPDK [49]. The SUT node runs an Ubuntu 20.04 LTS Linux distribution with Linux kernel release 5.13 and hosts our STAMP implementations. To control Linux networking capabilities (e.g., network interfaces, routing, and SRv6 behaviors), we installed the 5.13 release of the iproute2 [50] suite. We also installed ethtool 5.13 to configure the hardware capabilities of the NIC, such as offloading [51].

To perform the experiments, we used SRPerf [52], a performance evaluation framework for software and hardware implementations of SRv6. SRPerf orchestrates and automates the execution of the experiments using the TRex Python automation libraries [53]. It interacts with the TRex generator installed on the TG node. The TG generates packets using the TRex traffic generator and sends them to the SUT. The SUT processes the received packets. The TG evaluates the maximum throughput that can be processed by the SUT. SRPerf supports different throughput measurements, such as *No-Drop Rate (NDR)*, *Partial Drop Rate (PDR)*, and *Maximum Receive Rate (MRR)*. In our experiments, we used the Partial Drop Rate at a 0.5% drop ratio (in short, PDR@0.5%) as throughput measurement, which is defined as the maximum packet rate at which the packet drop ratio is less than or equal to 0.5%. For further details on this metric and how it is evaluated by the SRPerf tools, we refer to [52].

Our experiment is meant to evaluate the processing performance of the SD-WAN edge router, represented by the SUT in Fig. 11. In particular, our goal is to evaluate the impact of STAMP measurement procedures on the packet processing capabilities of a Linux software router. As a reference, we consider the scenario in which the router is only processing

regular data packets, then we intermix regular data packets with STAMP measurement packets in different percentages.

For the processing of regular data packets, we consider an SRv6 ingress node that performs packet encapsulation: it receives IPv6 packets and applies the H.Encaps behavior to encapsulate the packets in an outer IPv6+SRH packet. Therefore, in our baseline scenario the TG generates IPv6 packets, the SUT receives the packets on one interface, performs the encapsulation, and forwards the packets on the second interface.

For the processing of the STAMP measurement packets, we have considered two cases:

1) the SUT is configured as a STAMP Session-Reflector, it receives STAMP Session-Sender Test packets, processes them, and for each STAMP Test packet it sends a STAMP Session-Reflector Test packet to the TG;
2) the SUT is configured as a STAMP Session-Sender, it receives STAMP Session-Reflector Test packets, extracts, and collects the timestamps from the packets, performing the role of the Collector.

The impact of STAMP measurements is evaluated by changing the fraction of STAMP packets and measuring the packet processing capacity using the PDR@0.5% metric. When the SUT acts as a Session-Reflector (case 1), the methodology to evaluate the packet drop ratio described above can be applied easily, as both the data packets and the STAMP test packets are forwarded back by the SUT towards the TG (the data packets are encapsulated, the STAMP packets are processed and properly updated). To evaluate the packet drop ratio, the TG simply compares the number of transmitted and received packets in an experiment session (summing up the data and STAMP test packets). On the other hand, when the SUT acts as a Session-Sender/Collector (case 2), it does not forward the received STAMP test packets back to the TG, because it receives the STAMP packets and produces the measurement records. Therefore, the TG cannot simply count the packets transmitted back by the SUT to evaluate the packet drop ratio. In fact, the number of packets correctly processed by the SUT corresponds to the sum of data packets that are forwarded back and of the STAMP test packets that are properly processed by the SUT (i.e., by collecting the STAMP measurement metrics). A STAMP packet that is not processed by the SUT must count as a dropped packet. Therefore, the TG must retrieve the counter of processed STAMP packets from the router under test after each experiment session. To solve this problem, we have designed and implemented a gRPC based API. The SUT/router acts as a gRPC server, whereas a gRPC client in the TG queries the server after each experiment session and retrieves the number of processed STAMP packets. In this way, the TG can sum up this number with the number of received data packets and can properly evaluate the packet drop ratio.

To run the performance experiments, a careful configuration of the SUT node is needed because we need to saturate the capacity of a CPU to measure the PDR@0.5% metric. Therefore, we need that all tasks of our interest are executed by the selected CPU and we need to avoid that any other task is executed in the same CPU. A detailed discussion on these aspects can be found in the Appendix of [38]. A walk-through documentation of how to setup the testbed and run the experiment is available in [34].

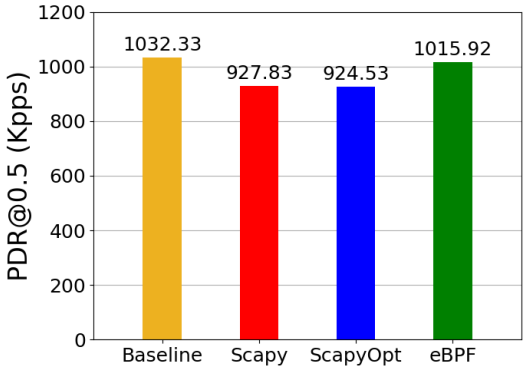



Fig. 12: Collector throughput, only data traffic.

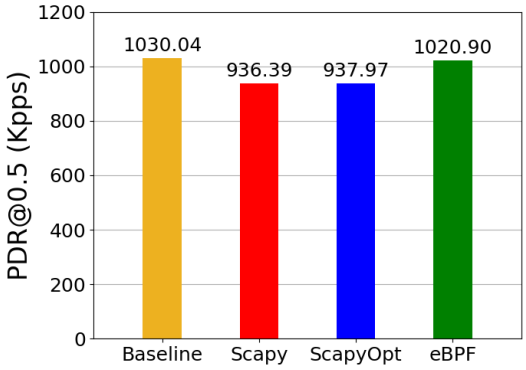



Fig. 13: Reflector throughput, only data traffic.

### B. *Performance analysis*

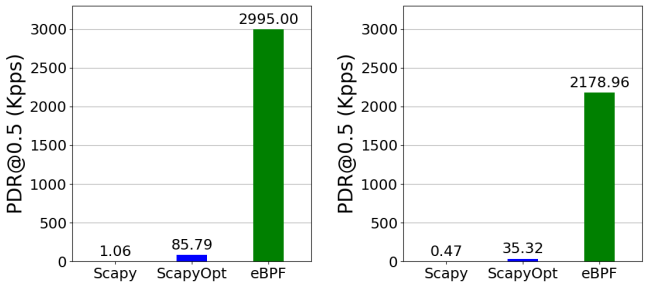
We report several experiments to evaluate the impact of our Session-Sender and Session-Reflector implementations on the user traffic. First, we evaluate the forwarding capability in the scenario with only data traffic (no STAMP test packets) without running any STAMP implementation. We consider this throughput as our baseline. Then, we run the Session-Sender or the Session-Reflector on the SUT and we evaluate the maximum achievable throughput for different combinations of data and STAMP test packets using our three different STAMP implementations.

The forwarding capacity of the node is measured using the PDR@0.5% metric as discussed in the previous subsection. The results reported in Figs. 12-16 are always the average of 10 evaluations (every single evaluation is carried out using the SRPerf tool [52]). We do not report error bars with confidence intervals in our figures, as we obtained stable results and the 95% confidence intervals are so close to the average that they are not noticeable. The tables with the detailed results are reported in the Appendix of [38].

The comparison among the STAMP Session-Sender/Collector implementations is shown in Fig. 12, where Scapy and ScapyOpt denote the basic and optimized Scapy implementations, respectively. Scapy denotes the basic Scapy

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2023.3300151






(a) Collector. (b) Reflector.

Fig. 14: Throughput, only STAMP traffic.

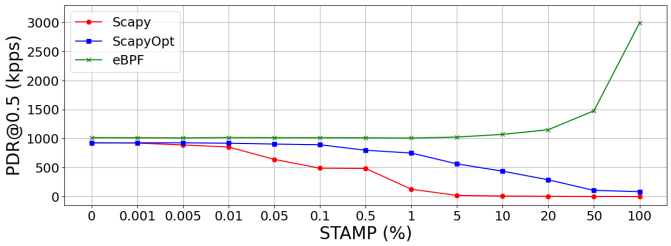



Fig. 15: Collector throughput.

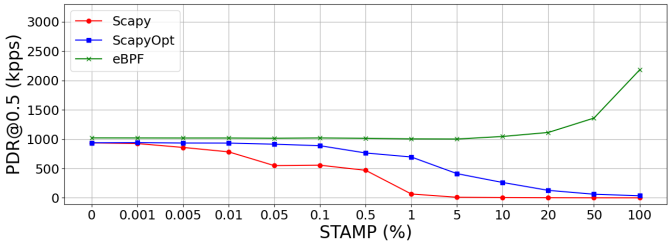



Fig. 16: Reflector throughput.

The Scapy implementations suffer a 10.4% performance degradation compared to the baseline performance. This performance degradation is due to the fact that even if there are no STAMP Test packets to be processed, the Session-Sender still has to look at all the incoming packets to capture the STAMP Test packets. This operation is very efficient, as it is executed in kernel mode. Both user space implementations have the same performance (≈925 kpps). The reason lies in the fact that even if the two implementations differ greatly in the processing of STAMP Test packets, the mechanisms used to filter the STAMP Test packets are the same. Thus, when there is only data traffic, the two implementations exhibit the same performance degradation. The packet rate of the *eBPF-based* implementation (≈1016 kpps) is higher than the two user space implementations. This is due to the fact that the HIKe eBPF chain contains a more efficient eBPF filter with respect to the filter of the user space implementation. Since this test is performed without STAMP packets, the performance is only affected by the filter that the packet traverses before being sent to the kernel networking stack. The performance drop of the *eBPF-based* implementation with respect to the baseline is 1.6%.

The STAMP Session-Reflector implementations exhibit the same behavior when processing only data traffic. A comparison among the Session-Reflector implementations is shown in Fig. 13.

We evaluated the PDR@0.5% in the opposite scenario in which there is only measurement traffic (i.e., only STAMP Test packets). The results are shown in Fig. 14.

Regarding the Session-Reflector (shown in Fig. 14a), the *basic* implementation reaches a packet rate of ≈1.06 kpps, which is much lower than the other two implementations. As discussed in Section VI, the reasons for this poor performance are related to the inefficiency of the Scapy AsyncSniffer and the high overhead of the Scapy builder and dissector. In the *optimized* implementation, we mitigated these issues. This allows the Session-Sender to reach an higher packet rate, ≈85.8 kpps. The performance of *eBPF-based* implementation is much higher (≈2995 kpps). The reason is that eBPF performs all the processing in kernel space, while *optimized* is a user space solution.

Concerning the performance of the Session-Reflector (shown in Fig. 14b), we observe the same trend (Fig. 14b). The *basic* implementation reaches a packet rate of ≈470 pps, which is lower than the packet rates of the *optimized* (≈35.3 kpps) and *eBPF-based* implementation (≈2179 kpps). The performance of the Session-Sender is always better than the Session-Reflector. The reason is that the Session-Sender processing is less expensive than the Session-Reflector processing. For each received STAMP Session-Reflector packet, the Session-Sender must collect and store the timestamps. Instead, when the Session-Reflector receives a STAMP Session-Sender Test packet, it must generate a STAMP Session-Reflector Test packet and forward the packet towards the Session-Sender. These operations are much more expensive than storing the timestamps.

Clearly, the scenario described above with only measurement traffic is unrealistic. We only use it to assess and compare the performance of the different implementations. In real scenarios, the measurement traffic (i.e., STAMP) is a small fraction of the overall traffic and will never reach 100% link capacity. For this reason, we analysed the performance considering different fraction of STAMP measurement packets.

Fig. 15 shows the maximum achievable throughput for the Session-Sender, varying the fraction of STAMP measurement packets. The *basic* implementation starts at ≈927.8 kpps at 0% STAMP, drops to ≈641.3 kpps (at 0.05% STAMP) and ≈20.3 kpps (at 5% STAMP), and then it continues to slowly drop to ≈1.06 kpps (100% STAMP). The throughput of the *optimized* implementation starts at ≈924.5 kpps and remains stable until the measurement traffic is 0.1% of the total traffic. The packet rate of the *eBPF-based* implementation starts at ≈1015.9 kpps when there is no measurement traffic (i.e., no STAMP packets) and it remains almost stable until the measurement traffic is 10% of the total traffic. Then, we observe a trend in contrast with the two user space implementations. The performance goes up to ≈1152.1 kpps when the measurement traffic is 20% of the total traffic and reaches ≈2994.9 kpps when the

measurement traffic is 100%.

The reason why the eBPF implementation starts with a higher throughput (PDR@0.5%) when the STAMP traffic is low, is that its BPF filter used to select the STAMP traffic is lighter than the one used by the Scapy implementations. When the percentage of STAMP traffic is very low, it does not affect the overall performance and the filtering is the only factor that plays a role. When the STAMP traffic increases, the throughput of the eBPF implementation increases because the STAMP packets are not sent to the kernel networking stack and they are processed faster by our eBPF program than the SRv6 packets that the kernel is encapsulating. On the other hand, the Scapy implementations process the STAMP packets in the user space, hence the performance is reduced when the fraction of STAMP packets increases.

The Session-Reflector throughput for different value of the percentage of STAMP measurement packets is shown in Fig. 16. The results for the three implementations are consistent with what we have discussed for the Session-Sender/Collector implementation. For high value of the percentage of STAMP traffic, it can be noted that the performance is slightly lower, this is because the Session-Reflector sends back the STAMP measurement packets. Apparently, this is heavier than storing the STAMP measurement records as done by the Session-Sender/Collector.

## IX. Related Works

Several solutions have been proposed for performance monitoring in a network. Some of them like Nagios [54] and Zabbix [55] focus on the monitoring of network devices. Other solutions like Ceilometer [56] target cloud environments. Concerning SDN, several solutions have been proposed. OpenNetMon [57] is a framework to measure throughput, delay, and packet loss in OpenFlow networks. A monitoring framework for SDN Virtual Networks is proposed in [58]. Other solutions for OpenFlow networks can be found in [59] and [60]. [7] proposes a review of the monitoring techniques used in SDN.

Internet Engineering Task Force (IETF) worked on the standardization of a protocol to measure the performance of IP and MPLS networks. This protocol is defined in RFC 4656 [61] and it is called *One-Way Active Measurement Protocol (OWAMP)*. OWAMP only focused on the one-way performance metrics, such as one-way delay and one-way packet loss. Another protocol was defined later, called *Two-Way Active Measurement Protocol (TWAMP)*. TWAMP (defined in RFC 5357 [62]) introduced the two-way measurements. RFC 5357 defines both the test protocol (i.e., the format of the messages exchanged to collect the measures) and the control protocol (i.e., the protocol used to setup the parameters required by the measurement session). RFC 8762 [28] introduces a new protocol, known as *Simple Two-Way Active Measurement Protocol (STAMP)*. RFC 8972 [29] proposes optional extensions, such as TLV (Type-Length-Value) coding to specify the Return Path. Later on, the STAMP protocol has been extended to support SR networks (both SR-MPLS and SRv6) [31]. This solution can measure metrics like delay or packet loss of a SRv6 path. The measurement mechanism is based on packets exchanged on the SRv6 path under measurement. These packets carry information used to compute the performance.

In [63], the authors described a per-flow packet loss measurement solution based on the *alternate marking* method called PF-PLM. They also proposed and compared two different implementations of the proposed solution, realized by extending Netfilter/Xtables and IP set Linux frameworks, respectively. In our previous work [64], we proposed an open source solution for Performance Monitoring of SRv6 networks, called SRv6-PM. SRv6-PM includes a cloud-native infrastructure that supports ingestion, processing, storage and visualization of PM data. We also provided an implementation based on the eBPF framework. Both works focused on packet loss monitoring.

An open source implementation of TWAMP and TWAMP light (STAMP) called *twampy* is available in [65]. Twampy has been released by a Nokia team with the goal of providing functional validation of the TWAMP implementation on Nokia devices. Twampy is coded in Python, running in user-space, and it does not support SRv6.

In [66], the authors described SRA, a user space implementation of the SRv6 data plane based on AF XDP. The proposed solution supports a custom SRv6 behavior called End.DM which enables the measurement of the delay in SRv6 networks. SRA collects the timestamps in each node of the SRv6 path. Our solution does not implement an SRv6 dataplane, it only implements the STAMP protocol and leaves the SRv6 packets to the Linux kernel. Moreover, STAMP is focused on the end-to-end delay, so it is not necessary to record all the intermediate nodes timestamps.

## X. Conclusion

In this paper, we proposed a solution to support the delay monitoring of SRv6 SD-WAN services. Our solution is based on the STAMP protocol and its extensions to support performance measurements in SRv6 networks, currently under discussion in the IETF. The main components of the solution are the STAMP Session-Sender and Session-Reflector which run in the SRv6 routers and perform the delay monitoring operations in the data plane. These data plane components need to be configured to execute the monitoring procedures. We defined and implemented an API that allows an SDN controller to interact with the Session-Sender and Session-Reflector. We integrated the proposed solution in EveryWAN, an SD-WAN open source prototype. Therefore, we deployed and tested a complete open source framework for delay monitoring of SRv6 based SD-WANs. In this respect, we have given a positive answer to the first two research and technological questions outlined in the introduction: i) the proposed approach based on IETF standards and current Internet drafts is an effective solution for delay monitoring of SRv6 networks; ii) we were able to implement the Delay Monitoring in an open source prototype based on Linux software routers, covering both the data plane aspects and the control plane aspects.

Then, we have addressed the research questions related to the performance impact of delay monitoring procedures on

a Linux software router. We have implemented the proposed solution in three different versions and executed a number of performance experiments to evaluate and compare the three implementations. We have started with a naive user space implementation of STAMP based delay monitoring, but we realized that its performance was poor, with a high reduction of the forwarding capacity of the software router. We have optimized the user space implementation, achieving an acceptable performance impact. In particular, with the optimized user space implementation the impact is acceptable when the fraction of measurement packets is kept within reasonable limits (e.g. less than 0.1%). We think that these limits will not be exceeded under practical operational conditions, as the number of measurement packets will always be a small fraction of the data traffic. Therefore, we have integrated the optimized user space implementation in our open source SD-WAN framework, which now offers a running prototype of the delay monitoring solution. We further considered a third implementation, based on the Linux eBPF technology. This proof-of-concept implementation providee a positive answer to question about the feasibility of delay monitoring in SD-WANs with negligible impact on the forwarding capability of a Linux software router.

## ACKNOWLEDGMENT

This work has received funding from the Cisco University Research Program, from Europeand Union under the GÉANT Innovation Programme and under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART").

## REFERENCES

[1] Cisco SD-WAN. [Online]. Available: https://www.cisco.com/c/en/us/solutions/enterprise-networks/sd-wan/index.html

[2] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined wan," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, p. 3–14, aug 2013. [Online]. Available: https://doi.org/10.1145/2534169.2486019

[3] C.-Y. Hong *et al.*, "B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined wan," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 74–87. [Online]. Available: https://doi.org/10.1145/3230543.3230545

[4] flexiWAN. [Online]. Available: https://flexiwan.com/

[5] M. Mahalingam *et al.*, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks," RFC 7348, Aug. 2014. [Online]. Available: https://www.rfc-editor.org/info/rfc7348

[6] C. Scarpitta *et al.*, "Everywan- an open source sd-wan solution," in *2021 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*, 2021, pp. 1–7.

[7] P.-W. Tsai *et al.*, "Network monitoring in software-defined networking: A review," *IEEE Systems Journal*, vol. 12, no. 4, pp. 3958–3969, 2018.

[8] W. Bekri, R. Jmal, and L. C. Fourati, "Softwarized internet of things network monitoring," *IEEE Systems Journal*, vol. 15, no. 1, pp. 826–834, 2021.

[9] V. R. Kebande, N. M. Karie, and R. A. Ikuesan, "Real-time monitoring as a supplementary security component of vigilantism in modern network environments," *International Journal of Information Technology*, vol. 13, no. 1, pp. 5–17, dec 2020. [Online]. Available: https://doi.org/10.1007/s41870-020-00585-8

[10] gRPC - A high performance, open source universal RPC framework. [Online]. Available: https://grpc.io/

[11] "ebpf," https://ebpf.io/, accessed: 2022-09-08.

[12] C. Filsfils *et al.*, "The segment routing architecture," in *2015 IEEE Global Communications Conference (GLOBECOM)*, 2015, pp. 1–6.

[13] C. Filsfils *et al.*, "Segment Routing Architecture," RFC 8402, Jul. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8402

[14] X. Xu *et al.*, "MPLS Segment Routing over IP," RFC 8663, Dec. 2019. [Online]. Available: https://www.rfc-editor.org/info/rfc8663

[15] C. Filsfils *et al.*, "IPv6 Segment Routing Header (SRH)," RFC 8754, Mar. 2020. [Online]. Available: https://www.rfc-editor.org/info/rfc8754

[16] C. Filsfils *et al.*, "Segment Routing over IPv6 (SRv6) Network Programming," RFC 8986, Feb. 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc8986

[17] S. Matsushima *et al.*, "SRv6 Implementation and Deployment Status," Internet Engineering Task Force, Internet-Draft draft-matsushima-spring-srv6-deployment-status-15, Apr. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-matsushima-spring-srv6-deployment-status-15

[18] What is VPP? [Online]. Available: https://wiki.fd.io/view/VPP

[19] G. Dawra *et al.*, "SRv6 BGP based Overlay Services," Internet Engineering Task Force, Internet-Draft draft-ietf-bess-srv6-services-15, Mar. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-bess-srv6-services/15/

[20] S. Previdi *et al.*, "Source Packet Routing in Networking (SPRING) Problem Statement and Requirements," RFC 7855, May 2016. [Online]. Available: https://www.rfc-editor.org/info/rfc7855

[21] C. Li *et al.*, "A Framework for Constructing Service Function Chaining Systems Based on Segment Routing," Internet Engineering Task Force, Internet-Draft draft-li-spring-sr-sfc-control-plane-framework-06, Apr. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-li-spring-sr-sfc-control-plane-framework/06/

[22] P. L. Ventre *et al.*, "Segment routing: A comprehensive survey of research activities, standardization efforts, and implementation results," *IEEE Communications Surveys Tutorials*, vol. 23, no. 1, pp. 182–221, 2021.

[23] ROSE Project. [Online]. Available: https://netgroup.github.io/rose/

[24] Innovation Programme - GÉANT Community. [Online]. Available: https://community.geant.org/community-programme-portfolio/innovation-programme/

[25] User Controlled SD-WAN Services with Performance Monitoring over GÉANT report. [Online]. Available: https://github.com/everywan-io/everywan-io.github.io/raw/master/docs/everywan-ucss-report-v01.pdf

[26] Svinota. (1999) Pyroue2. [Online]. Available: https://github.com/svinota/pyroute2

[27] EveryWAN white paper. [Online]. Available: https://github.com/everywan-io/everywan-io.github.io/raw/master/docs/EveryWAN-WhitePaper-v1.pdf

[28] G. Mirsky *et al.*, "Simple Two-Way Active Measurement Protocol," RFC 8762, Mar. 2020. [Online]. Available: https://www.rfc-editor.org/info/rfc8762

[29] G. Mirsky *et al.*, "Simple Two-Way Active Measurement Protocol Optional Extensions," RFC 8972, Jan. 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc8972

[30] R. Gandhi *et al.*, "Simple TWAMP (STAMP) Extensions for Segment Routing Networks," Internet Engineering Task Force, Internet-Draft draft-ietf-ippm-stamp-srpm-03, Feb. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-ippm-stamp-srpm-03

[31] R. Gandhi *et al.*, "Performance Measurement Using Simple TWAMP (STAMP) for Segment Routing Networks," Internet Engineering Task Force, Internet-Draft draft-ietf-spring-stamp-srpm-03, Feb. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-spring-stamp-srpm-03

[32] J. Martin *et al.*, "Network Time Protocol Version 4: Protocol and Algorithms Specification," RFC 5905, Jun. 2010. [Online]. Available: https://www.rfc-editor.org/info/rfc5905

[33] "[a guide to gps ntp servers for network time synchronization]," https://timetoolsltd.com/gps/gps-ntp-server/, accessed: 2023-07-07.

[34] SRv6 Delay Monitoring Home Page. [Online]. Available: https://netgroup.github.io/srv6-delay-mon/

[35] "IEEE standard for a precision clock synchronization protocol for networked measurement and control systems." [Online]. Available: https://doi.org/10.1109/ieeestd.2008.4579760

[36] SRv6 Delay Monitoring Code Repository. [Online]. Available: https://github.com/everywan-io/srv6pm-delay-measurement

[37] P. L. Ventre *et al.*, "Sdn architecture and southbound apis for ipv6 segment routing enabled wide area networks," *IEEE Transactions on Network and Service Management*, vol. 15, no. 4, pp. 1378–1392, 2018.

[38] C. Scarpitta *et al.* (2022) High Performance Delay Monitoring for SRv6 Based SD-WANs. arXiv preprint arXiv:2212.12627. [Online]. Available: https://arxiv.org/pdf/2212.12627

[39] Scapy - Packet crafting for Python2 and Python3. [Online]. Available: https://scapy.net/

[40] Packet MMAP. [Online]. Available: https://www.kernel.org/doc/html/latest/networking/packet_mmap.html

[41] "Hike/eclat," https://hike-eclat.readthedocs.io/en/latest/, accessed: 2022-09-08.

[42] A. Mayer *et al.*, "eBPF Programming Made Easy with eCLAT," in *2022 18th International Conference on Network and Service Management (CNSM)*, 2022, pp. 28–36.

[43] G. Sidoretti and S. Salsano. HIKe package STAMP. [Online]. Available: https://github.com/netgroup/hikepkg-stamp

[44] B. P. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962. [Online]. Available: http://www.jstor.org/stable/1266577

[45] Algorithms for calculating variance. [Online]. Available: https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance

[46] "Benchmarking Methodology for Network Interconnect Devices," RFC 2544, Mar. 1999. [Online]. Available: https://www.rfc-editor.org/info/rfc2544

[47] CloudLab home page. [Online]. Available: https://www.cloudlab.us/

[48] TRex realistic traffic generator. [Online]. Available: https://trex-tgn.cisco.com/

[49] DPDK. [Online]. Available: https://www.dpdk.org/

[50] Linux Foundation Wiki - iproute2. [Online]. Available: https://wiki.linuxfoundation.org/networking/iproute2

[51] ethtool - Linux man page. [Online]. Available: https://linux.die.net/man/8/ethtool

[52] A. Abdelsalam *et al.*, "Srperf: A performance evaluation framework for ipv6 segment routing," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 2320–2333, 2021.

[53] TRex Stateless Python API. [Online]. Available: https://trex-tgn.cisco.com/trex/doc/cp_stl_docs/index.html

[54] "Nagios it infrastructure monitoring," https://www.nagios.org/, accessed: 2022-08-04.

[55] "Zabbix monitoring everything," https://www.zabbix.com/, accessed: 2022-08-04.

[56] "Openstack celiometer," https://docs.openstack.org/ceilometer/latest/, accessed: 2022-08-04.

[57] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers, "Opennetmon: Network monitoring in openflow software-defined networks," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, 2014, pp. 1–8.

[58] G. Yang *et al.*, "Network monitoring for sdn virtual networks," in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, 2020, pp. 1261–1270.

[59] W. Queiroz, M. A. Capretz, and M. Dantas, "An approach for sdn traffic monitoring based on big data techniques," *Journal of Network and Computer Applications*, vol. 131, pp. 28–39, 2019.

[60] R. B. Santos, T. R. Ribeiro, and C. de AC César, "A network monitor and controller using only openflow," in *2015 Latin American Network Operations and Management Symposium (LANOMS)*. IEEE, 2015, pp. 9–16.

[61] S. Shalunov, B. Teitelbaum, et. al, "A One-way Active Measurement Protocol (OWAMP)," IETF RFC 4656, Sep. 2006. [Online]. Available: https://tools.ietf.org/html/rfc4656

[62] K. Hedayat, R. Krzanowski, et al., "A Two-Way Active Measurement Protocol (TWAMP)," IETF RFC 5357, Sep. 2006. [Online]. Available: https://tools.ietf.org/html/rfc5357

[63] P. Loreti *et al.*, "Implementation of accurate per-flow packet loss monitoring in segment routing over ipv6 networks," in *2020 IEEE 21st International Conference on High Performance Switching and Routing (HPSR)*, 2020, pp. 1–8.

[64] P. Loreti *et al.*, "Srv6-pm: A cloud-native architecture for performance monitoring of srv6 networks," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 611–626, 2021.

[65] "Python tools for twamp and twamp light," https://github.com/nokia/twampy, 2017, accessed: 2023-07-05.

[66] B. Zhao *et al.*, "Sra: Leveraging af_xdp for programmable network functions with ipv6 segment routing," in *2022 IEEE 47th Conference on Local Computer Networks (LCN)*, 2022, pp. 455–462.