Performance of IPv6 Segment Routing in Linux Kernel

Ahmed Abdelsalam*, Pier Luigi Ventre†, Andrea Mayer†, Stefano Salsano†,
Pablo Camarillo‡, Francois Clad‡, Clarence Filsfils‡
*Gran Sasso Science Institute, †University of Rome Tor Vergata, ‡Cisco Systems

***Abstract*—IPv6 Segment Routing (SRv6) is a promising solution to support advanced services such as Traffic Engineering, Service Function Chaining, Virtual Private Networks, and Load Balancing. The SRv6 data-plane is supported in many different forwarding engines including the Linux kernel. It has been introduced into the 4.10 release of the Linux kernel to support both endhost and router functionalities. The implementation has been updated several times, with every new kernel release, to include new features and also to improve the performance of existing ones. In this paper, we present SRPerf, a performance evaluation framework for software and hardware implementations of SRv6. SRPerf is able to perform different benchmarking tests such as throughput and latency. The architecture of SRPerf can be easily extended to support new benchmarking methodologies as well as different SRv6 implementations. We have used SRPerf to evaluate the performance of the SRv6 implementation in the Linux kernel.**

***Index Terms*—Segment Routing, SRv6, Performance, Linux kernel, Data-Plane**

I. Introduction

Segment Routing (SR) is a network architecture based on source routing [1]. It delivers all the benefits of source routing mechanisms required for today's networks, such as the flexibility in specifying a forwarding path other than the regular shortest path [2]. At the same time, it addresses the security concerns that were the reason for deprecating previous source routing techniques [3] [4]. In the SR architecture, the source can insert into a packet an ordered list of instructions, denoted as *segments*, to steer the packet through a set of intermediate nodes in the path towards its final destination. Each segment is represented with a segment identifier (SID), which is based on the data-plane implementation of the SR architecture. Currently, the SR architecture has two different data-plane implementations: MPLS and IPv6 where SIDs are respectively represented as MPLS labels and IPv6 addresses.

IPv6 Segment Routing (SRv6) is the instantiation of the SR architecture over the IPv6 data plane. It defines a new type of IPv6 routing extension header known as Segment Routing Header (SRH) [5]. The SRH carries the list of SIDs that must be traversed by the packet and a pointer to the active *segment* in the list. The initial SRv6 architecture is extended from the simple steering of packets across nodes to a general network programming approach in [6]. It is possible to encode instructions and not only locations in a the SIDs list. Each instruction represents a function to be executed at a specific location in the network. A set of well-known functions that can be associated with a SID are defined in [6]. However, this set is not exhaustive and network operators can define their own functions. SRv6, as a network underlay technology, fulfills the needs of the various overlay services, such as traffic engineering, load balancing, and service chaining of network functions [2].

The data-plane implementations of SRv6 have been supported in many different routers implementations including: open-source software routers such as the Linux kernel and the Vector Packet Processing (VPP) platform [7], and hardware implementations from different network vendors [5]. The SRv6 implementations have drawn a lot of attention to researchers from academia and industry. The interoperability between several software and hardware implementations of SRv6 is reported in [8]. In this paper, we focus on the SRv6 implementation in the Linux kernel. It has been introduced into the 4.10 release of Linux kernel to support both endhosts and router functionalities. The implementation has been updated several times to include new functionalities and also to improve the performance of supported features with every new kernel release. Several research activities have leveraged from the SRv6 capabilities in Linux kernel [9] [10] [11] [12].

The introduction of SRv6 in ISP networks requires the assessment of its non-functional properties like scalability and fault tolerance. Hence, it is required to have a realistic performance evaluation framework. The platform should allow scaling up to the current transmission line rates and should be available for re-use by academic researchers on any commodity hardware. However, to the best of our knowledge, the only reported performance for the SRv6 implementation in the Linux kernel are in [13] [14]. They are early evaluations reporting the performance of the very first implementations of SRv6. Moreover, the works do not provide such required framework. Therefore, we advocate the need of an open source reference platform.

The design of a performance evaluation framework for data-plane implementations is a very challenging task [15]. As they are required to forward packets at an extremely high rate using a limited CPU budget to process each of these packets. The IETF has defined the guidelines and the tests for benchmarking forwarding implementations [16]. The benchmarking tests include: throughput, latency, jitter and frame loss rate. Throughput is the most commonly used benchmarking measure for forwarding implementations [17]. It is defined as the maximum rate at which all received packets

This work has been partially supported by the Cisco University Research Program (URP) fund

978-3-903176-14-0 © 2018 IFIP

are forwarded by the device and often reported in number of packet per second (pps). There are different variations of the throughput including No-Drop Rate (NDR), and Partial Drop Rate (PDR) [18].

In this paper, we present SRPerf, a performance evaluation framework for software and hardware implementations of SRv6. Currently, SRPerf supports only the SRv6 forwarding in the Linux kernel, but it can be easily extended to support other forwarding engines such as VPP. It can report different throughput measures such as NDR and PDR. The current design relies on TRex as a traffic generator [19]. We have used SRPerf to evaluate the performance of the SRv6 implementation in the Linux kernel. In particular, we report the PDR measures for different SRv6 forwarding behaviors supported by the Linux kernel. The implementation of SRPerf is open-source and publicly available at [20].

The paper is structured as follows: Section II presents the SRv6 support in the Linux kernel. The design of SRPerf and the evaluation methodology are described in Section III. Section IV explains the testbed and elaborates on the experiments we have performed. Finally, we draw some conclusions and highlight the directions for future work in Section V.

II. SRv6 SUPPORT IN THE LINUX KERNEL

The SRv6 implementation was merged in Linux kernel 4.10 [14]. It has been improved several times to track the evolution of the SRv6 network programming concept defined in [6]. The model defines two different set of SRv6 behaviors, known as *transit* and *endpoint* behaviors. *Transit* behaviors steer received packets into the SRv6 policy matching the packet information. Each SRv6 policy has a list of SIDs to be attached to the matched packets. On the other hand, an SRv6 *endpoint* behavior represents a function to be executed on packets at specific location in the network. Such function can be a simple routing instruction, but also can be any advanced network function (e.g., firewall, NAT).

In the Linux kernel, the SRv6 behaviors are implemented as Linux lightweight tunnel (`lwtunnel`). The `lwtunnel` is an infrastructure that was introduced in Linux kernel 4.3 to allow for scalable flow-based encapsulation such as MPLS and VXLAN. In the Linux kernel, SRv6 SIDs are configured as IPv6 FIB entries into the main routing table, or any secondary routing table [21]. In order to support adding SIDs associated with an SRv6 behavior, the iproute2 user-space utility has been extended [22]. The SRv6 capabilities in the Linux kernel were improved to include the netfilter framework [23] as well as the eBPF framework [24]. In the netfilter framework, a new iptables match extension (`srh`) was added to support matching of the SRH fields. The `srh` match extension is a part of the SERA firewall [10]. In the eBPF framework, a new feature is added to support implementing custom SRv6 network functions in eBPF and install them in the kernel.

Several SRv6 *transit* behaviors are supported in the Linux kernel, including: *T.Insert*, *T.Encaps*, and *T.Encaps.L2*. The *T.Insert* behavior inserts an SRH in the original IPv6 packet, immediately after the IPv6 header and before the transport level header. The original IPv6 header is modified, in particular the IPv6 destination address is replaced with the IPv6 address of the first segment in the segment list, while the original IPv6 destination address is carried in the SRH header as the last SID of the SIDs list. The *T.Encaps* behavior encapsulates the original IPv6 packet as the inner packet of an IPv6-in-IPv6 encapsulated packet. The outer IPv6 header carries the SRH header, which carries the SIDs list. The *T.Encaps.L2* behavior is the same as the *T.Encaps* behavior, with the difference that *T.Encaps.L2* encapsulates the full received layer-2 frame rather than the IP packet.

The Linux kernel also supports several SRv6 *endpoint* behaviors including: *End*, *End.T*, *End.X*, *End.DX2*, *End.DT6*, and *End.DX6*. The *End* behavior represents the most basic SRv6 function. It replaces the IPv6 destination address of the packet with the next active SID from the SIDs list. Then, forwards the packet based on a FIB lookup using the updated destination address. The *End.T* behavior is a variant of the *End* behavior relying on the multiple tables lookup functionality: the FIB lookup is performed in a specific IPv6 table associated with the SID rather than the main FIB table. The *End.X* behavior is another variant of the *End* behavior where the packet is forwarded to one of the layer-3 adjacencies bound to the SID rather than the IPv6 destination address. The *End.DX2*, *End.DT6*, and *End.DX6* behaviors are variants of the *End* behavior that require removing the SRv6 encapsulation (outer IPv6 header and its extension headers) before forwarding the packet. They are used to implement the termination of different types of layer-2 and layer-3 VPN use-cases. The *End.DX2* behavior pops out the SRv6 encapsulation and forwards the resulting frame via the output interface associated to the SID. The *End.DT6* behavior pops out SRv6 encapsulation and perform a FIB lookup with the IPv6 destination address of the exposed inner packet in an IPv6 table associated with the SID. The *End.DX6* behavior removes the SRv6 encapsulation from the packet and forwards the resulting IPv6 packet to one of the layer-3 adjacencies bound to the SID.

Another set of SRv6 *endpoint* behaviors were defined in [25] to the chaining of SR-unaware network functions. Some of these behaviors are implemented in an external Linux kernel module such as *End.AD* and *End.AM* [26]. The details and performance evaluation of these behaviors as well as other SRv6 *endpoint* behaviors, not listed above, have not been considered in this work and are left for future works.

III. SRPERF

In this section, we illustrate our performance evaluation framework (SRPerf). At first, we describe the internal design and the high level architecture of SRPerf (Section III-A). Then, Section III-B elaborates on our evaluation methodology which leverages the PDR to characterize the performance of a forwarding node.

A. Design and high level architecture

We designed SRPerf following the network benchmarking guidelines defined in RFC 2544 [16]. The architecture of

SRPerf is composed of two main building blocks: testbed and orchestrator as shown in Figure 1. The testbed is composed of two nodes, the tester and the System Under Test (SUT). The nodes have two network interfaces cards (NIC) each and are connected back-to-back using both NICs. The tester sends traffic towards the SUT through one port, which is then received back through the the port, after being forwarded by the SUT. Accordingly, the tester can easily perform all different kinds of throughput measurements as well as round-trip delay and jitter. In our current design, we chose the open source project TRex as Traffic Generator (TG) that supports both transmitting and receiving ports [19].

The orchestrator is responsible for the automation of the performance evaluation process. It controls the TG (deployed in the tester) through the high level API provided by *TG driver*. The *TG driver* translates the calls coming from the other modules in commands to be executed on the python client of the TRex automation API [27]; such API provides the means to generate different traffic patterns (e.g., layer-3 traffic) at different rates (up to the line rate of the TG sending port). It also provides the means to extract statistics from the traffic generator after each successful run.

The *cfg manager* controls the forwarding engine in the SUT. It is responsible for enforcing the required configuration in the forwarding engine. The orchestrator implements a mapping between the supported forwarding behaviors and required configuration for each behavior. Hence, the orchestrator is able to properly instruct the *cfg manager*. For example, to test the *End* behavior, the *cfg manager* has to configure the forwarding engine in the SUT with two FIB entries; one as an SRv6 SID with *End* behavior and the second as plain IPv6 FIB entry to forward the packet once performed the *End* function. The configuration can be as simple as adding a FIB entry to forward the received packets back to the tester, but it can also be more complex configuration that manipulate the incoming packets before forwarding them back to the tester. The orchestrator implements another mapping between the supported forwarding behaviors and the type of traffic required to test each behavior. For example, to test the *End* behavior, it is necessary to use an SRv6 packet with an SRH containing a SIDs list of at least two SIDs and the active SID must not be the last SID.

The orchestrator supports different algorithms for calculating the throughput and the delay measurements. Each algorithm provides an API through which the orchestrator can run an experiment. An example of currently supported throughput measurement algorithms is the Partial Drop Rate (PDR), described in Section III-B.

The configuration file depicted in the upper part of the Figure 1 represents the necessary input to run the experiments. It defines the forwarding engine in the SUT node, the set of SRv6 behaviors to be tested, the type of test to be performed and the algorithm for calculating it and number of runs. The configuration file can be represented in any configuration language. Currently, SRPerf supports only YAML [28] for the configuration format. The orchestrator leverages the *cfg parser* to extract the configuration parameter and to initialize the experiment variables. The SRPerf implementation is open source and available at [20]. It supports another API for the automatic generation of the configuration files. Moreover, it provides a different configuration scripts to to deploy an experiment using SRPerf on any commodity hardware.

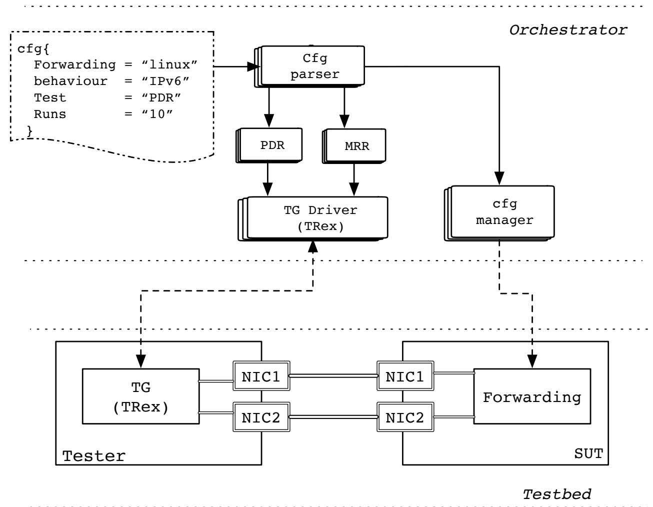


Fig. 1: SRPerf architecture.

The framework is modular and can be expanded in different directions. For example, it can be extended to support new traffic generators by simply creating a new driver for each. A new forwarding behavior can be added by updating the *cfg manager* with the configuration required for such behavior. New algorithms for calculating throughput and delay can be developed and plugged into the orchestrator. It can support different *forwarding* engines in the SUT, which only requires the *cfg manager* to be updated to recognize them.

B. Evaluation methodology and PDR finder algorithm

Throughput, defined in RFC 1242 [17], is the maximum rate at which all received packets are forwarded by the device. It is used as a standard measure to compare performance of network devices from different vendors. Throughput can be reported in number of bits per second (bps) as well as number of packet per second (pps). The precision of throughput measurement depends on type of the traffic to be forwarded. Some protocols can not tolerate even the loss of one frame, while others can tolerate a certain percentage of packets loss. Hence, there are two variations of the throughput, No-Drop Rate (NDR) and Partial Drop Rate (PDR). NDR is the highest throughput achieved without dropping packets. PDR is the highest throughput achieved without dropping more than a pre-defined threshold [29]. NDR can be described as PDR with threshold of 0%. Here, we consider only the PDR since it is more generic than the NDR. Moreover, in certain conditions it is not feasible to measure a zero loss rate.

Finding the PDR of a given forwarding behavior is a time consuming process since it requires the scanning of a broad range of possible traffic rates. In order to explain the process,

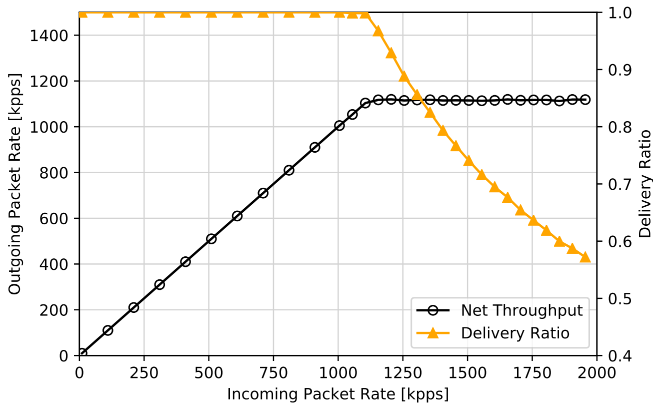


Fig. 2: Throughput of plain IPv6 forwarding

we plotted the IPv6 forwarding in the Linux kernel as shown in Figure 2. The figure reports a set of different sending rates along with the throughput and the Delivery Ratio (DR) for each rate. DR is the ratio between the input and the output packet rates of a device. It should be always be 100% for all data rates less than the device throughput. Initially, the throughput increases linearly with the increase in the sending rate. This region is often referred to as no drop region where the DR is always 100%. The CPU usage at the SUT node increases with the increase in traffic sending rate by the tester. Ideally, the SUT node should be able to forward all received packets until it becomes 100% CPU saturated. After the saturation point, the SUT node starts to drop some of the received packets. Once the drop threshold is reached, the process should terminate and report the calculated value as PDR. The same logic is valid for calculating the NDR, by having a drop threshold of 0%.

In order to automate the PDR finding process, we have designed and developed the PDR finder algorithm. It scans a range of traffic rates with the objective of estimating the PDR value. The algorithm performs a combination of both exponential and logarithmic search. It returns an interval $[a, b]$ of traffic rates for the PDR value. The maximum interval distance (ϵ) is an configurable option to tune the algorithm precision. The algorithm terminates when the difference between a and b is less or equal than ϵ ($b - a \leq \epsilon$).

The exponential search is the first phase of the PDR finder algorithm. Firstly, the algorithm defines a searching window $[start, end]$ where $start$ is the initial sending rate, which can be any values that guarantees no loss, and end is the maximum sending rate that the TG can not go beyond and usually based on the sending NIC capabilities. The algorithms defines β as the threshold of drop rate that can be tolerated. For each sending rate the algorithm calculates the throughput and DR. If the $DR \geq (100 - \beta)\%$, the new sending rate is set to be twice the current sending rate. Otherwise, the algorithm starts the logarithmic search phase.

The logarithmic search leverages the output of the exponential search as an input. The algorithm starts to decrease the amplitude of the searching window until such value becomes less than the minimum interval width (ϵ). At each iteration, the size of the searching window is halved and the DR is evaluated for the window middle point, which is considered to be the current traffic rate. If the DR of the middle point is less than the threshold, the upper bound of the window is set to the current rate. Otherwise, the lower bound of the searching window is set with the current rate. This process is iterated until the exit condition is triggered.

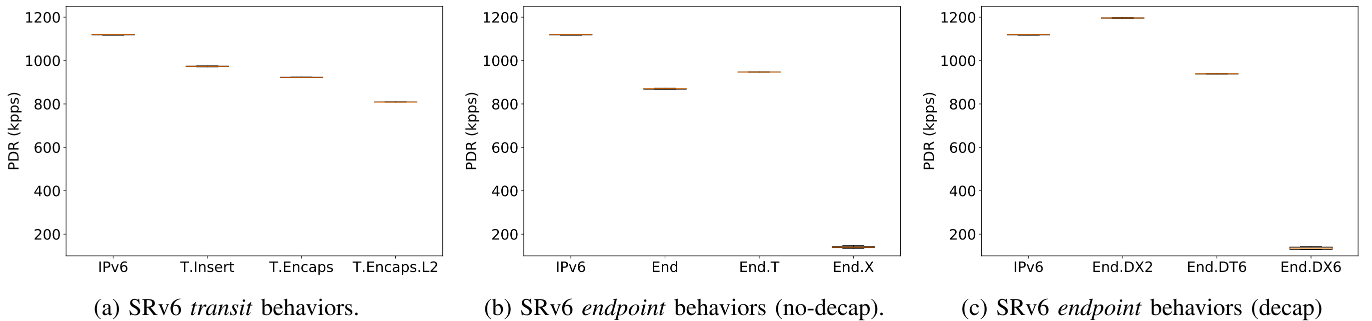
IV. SRv6 Performance in the Linux kernel

We deployed our testbed illustrated in Figure 1 on CloudLab [30]. Each of the testbed nodes (Tester and SUT) is powered by a bare metal server equipped with an Intel Xeon E5-2630 v3 processor with 16 cores (hyper-threaded) clocked at 2.40GHz and 128 GB of RAM. Each bare metal server has two Intel 82599ES 10-Gigabit network interface cards to provide back-to-back connectivity between the testbed nodes. The tester is running TRex in the stateless mode and has the TRex python automation libraries installed. The SUT machine is running Linux kernel 4.18.5 vanilla. It has the 4.18.0 release of the *iproute2* [22] installed, which provides the means to program the SRv6 behaviors. In addition, *ethtool* (release 4.18) is installed to provide the means to configure the NIC hardware capabilities such as offloading [31].

Performance evaluation of forwarding behaviors often includes performance for both single as well as well multiple CPU cores. Here we show only the performance of SRv6 behaviors for the case of single CPU core. The single CPU measures provide the base performance for a given behavior. In order to force the single CPU core processing of all received traffic we rely on the Receive-Side Scaling (RSS) and SMP IRQ affinity features. RSS is responsible for distributing the received packets across several hardware-based receive queues. The number of receive queues scales with the number of CPU threads. Each receive queue is assigned a CPU thread to process its packets. The distribution of packets across the receive queues is based on a hash function which assigns packets of the same traffic flow to the same receive queue, hence being processed by the same CPU thread. However, we also used the SMP IRQ affinity feature to assign all the receive queues to the same CPU core to guarantee the single CPU processing independently from the hash function feature. Moreover, in order to get the base performance independent of the NIC hardware capabilities, we disabled all the NIC hardware offloading capabilities such as large receive offload (LRO), generic receive offload (GRO), generic segmentation offload (GSO), and all checksum offloading features. Finally, we disabled the hyper-threading feature of the SUT node from the BIOS settings.

A. Performance evaluation of SRv6 behaviors

We performed three experiments as follows: i) SRV6 *transit* behaviors; ii) SRv6 *endpoint* behaviors with no decapsulation (no-decap); iii) SRv6 *endpoint* behaviors with decapsulation (decap). The *decap* behaviors are required to remove the SRv6 encapsulation from packets before forwarding them. Conversely, the no-decap behaviors forwards SRv6 packets



(a) SRv6 *transit* behaviors. (b) SRv6 *endpoint* behaviors (no-decap). (c) SRv6 *endpoint* behaviors (decap)

Fig. 3: SRv6 performance in the Linux kernel.

	IPv6	T.Insert	T.Encaps	T.Encaps.L2	End	End.T	End.X	End.DX2	End.DT6	End.DX6
Mean	1119.25	973.437	922.625	809.031	869.875	946.937	140.406	1196	938.875	135.375
CV	0.003%	0.053%	0.015%	0.024%	0.1%	0.013%	0.838%	0.041%	0.037%	1.138%
CI_{95}	0.002%	0.033%	0.01%	0.015%	0.063%	0.008%	0.53%	0.026%	0.023%	0.72%

TABLE I: SRv6 performance in the Linux kernel (kpps). Mean, CV and CI_{95}

without removing the SRv6 encapsulation from packets. In the first experiment, we use an IPv6 packet of size 64 bytes. While in experiments 2 and 3, we use an SRv6 packet of size 64 bytes plus SRv6 encapsulation. The SRv6 encapsulation is 80 bytes representing 40 bytes of outer IPv6 header and 40 bytes of SRH with two SIDS. The results of the experiments are plotted in Figure 3. We compare the performance of the SRv6 behaviors with the plain IPv6 forwarding, which represents the baseline in our experiments, to provide a characterization of the scalability of SRv6 forwarding in the Linux kernel. We use the PDR described in Section III-B as a metric to report our results. The trail period in our experiments is 10 seconds. We use the boxplot to plot our results, where each boxplot represents 10 PDR values. Table I reports respectively the average, the Coefficient of Variation (CV) and the 95% Confidence Interval (CI_{95}) of each analyzed forwarding behavior.

In the first experiment we evaluated the performance of the three different SRv6 *transit* behaviors: *T.Insert*, *T.Encaps*, and *T.Encaps.L2*. The results are shown in Figure 3a along with the IPv6 base line. The *T.Insert* shows a forwarding performance of $\approx$ 973 kpps compared to $\approx$ 922 kpps for *T.Encaps*. For the *T.Encaps.L2* behavior, the SUT node is able to forward $\approx$ 809 kpps. The performance of *T.Insert* behavior is a slightly better compared to *T.Encaps* since the former needs to push only an SRH while the latter needs to push an outer IPv6 header along with the SRH. In general, the SRv6 *transit* behaviors have shown very stable performance as witnessed by the low values for the CV.

Experiment 2 reports the performance of the *no-decap* SRv6 *endpoint* behaviors. We evaluated the performance of the *End*, *End.T*, and *End.X* behaviors. The performance of these behaviors is compared to IPv6 base line in Figure 3b. In case of the *End* behavior the SUT node is able to forward $\approx$ 869 kpps with 20% decrease in the performance compared to plain IPv6. The *End.T* performs better than the *End* since the the routing table used for the lookup is defined by the control plane, hence the kernel saves the cost of performing IP rules lookup that are executed in case of the *End* behavior. The *End.T* forwarding performance is $\approx$ 946 kpps with 13% of performance drop compared to plain IPv6 and 7% of increase in performance compared to the *End*. The *End.X* shows a very poor performance of $\approx$ 140 kpps with around 87% drop in the performance compared to plain IPv6. The performance is less stable with the respect to the other behaviors. We explain the reason for such bad performance in Section IV-B.

Finally, last experiment compares the performance of the SRv6 *decap endpoint* behaviors. The results are compared to the plain IPv6 forwarding in Figure 3c. The *End.DX2* behavior has a throughput of $\approx$ 1196 kpps which is 7% better than the plain IPv6. The reason why *End.DX2* is performing better than IPv6 is that the kernel does not need to perform Layer-3 lookup once the packet has been decapsulated. Instead, it pushes the packet directly into the transmit queue of the interface towards the next-hop. As for *End.DT6* and *End.DX6* we have a performance drop respectively of 27% and of 86%. Moreover, the latter shows more instability compared to the other behaviors with a CV $\approx$ 1.14%; we shed some lights in the following section (IV-B) for such decrease in the performance.

B. Analysis of cross-connect behaviors

The *End.X* and *End.DX6* behaviors exhibit poor performances and much higher variance across the runs compared to the other SRv6 *endpoint* behaviors as shown in Figure 3 and Table I. Figure 4 focuses the y-axis in the range from 100 to 200 kpps to show a zoomed view of the PDR values of their performance. The two behaviors share the same logic, they perform different variations of Layer 3 cross-connect to an adjacency set by the control plane. Since the next-hop is already known, the lookup in the routing tables has to be bypassed. Typically, when a packet has to be forwarded, the routing subsystem needs to find a route in the routing tables and returns a structure `rt6_info` as results of the lookup. Caches are widely used to avoid a lookup in the routing tables

and further memory allocations for each packet. However, when the aforementioned behaviors activated caches are not used and a structure `rt6_info` is allocated for each packet to emulate the lookup process. The memory allocation has a huge performance impact leading to a performance drop around 90% compared to the default IPv6 forwarding.

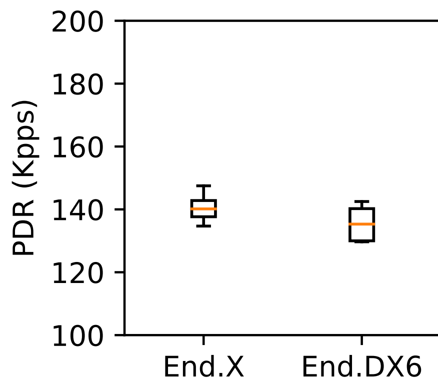


Fig. 4: Performance of cross-connect behaviors

V. Conclusions

In this paper, we have described the design and implementation of SRPerf, a performance evaluation framework for SRv6 implementations. SRPerf has been designed to be extendable: it can support different forwarding engines including software and hardware forwarding, but can also be extended to support different traffic generators. We have used SRPerf to evaluate the performance of some SRv6 behaviors in the Linux kernel. Results show a reasonable performance for some SRv6 behaviors compared to the plain IPv6 forwarding. However, some other behaviors such as *End.X* and *End.DX6* show a decrease of around 90% in the forwarding performance due to some implementation issues that require memory allocation operation for each packet.

Potential directions for future work include evaluating the performance of the SRv6 behaviors such as *End.DX4*, IPv4 variant of *T.Encaps*, *End.AD*, *End.AM*, *End.B6* and *End.B6.Encaps*. Moreover, we plan to extend SRPerf to support VPP as alternative SRv6 forwarding engine, so we can provide a comparison between the Linux kernel and the VPP implementations.

References

[1] C. Filsfils *et al.*, "Segment Routing Architecture," Internet Requests for Comments, RFC Editor, RFC 8402, July 2018. [Online]. Available: https://tools.ietf.org/html/rfc8402

[2] S. Previdi *et al.*, "Source Packet Routing in Networking (SPRING) Problem Statement and Requirements," Internet Requests for Comments, RFC Editor, RFC 7855, May 2016. [Online]. Available: https://tools.ietf.org/html/rfc7855

[3] F. Gont *et al.*, "Recommendations on Filtering of IPv4 Packets Containing IPv4 Options," Internet Requests for Comments, RFC Editor, RFC 7126, February 2014. [Online]. Available: https://tools.ietf.org/html/rfc7126

[4] J. Abley *et al.*, "Deprecation of Type 0 Routing Headers in IPv6," Internet Requests for Comments, RFC Editor, RFC 5095, December 2007. [Online]. Available: https://tools.ietf.org/html/rfc5095

[5] C. Filsfils et al., "IPv6 Segment Routing Header (SRH)," IETF, Internet-Draft, June 2018. [Online]. Available: https://tools.ietf.org/html/draft-ietf-6man-segment-routing-header-14

[6] C. Filsfils *et al.*, "SRv6 Network Programming," IETF, Internet-Draft, July 2018. [Online]. Available: https://tools.ietf.org/html/draft-filsfils-spring-srv6-network-programming-05

[7] "What is VPP ?" https://wiki.fd.io/view/VPP.

[8] C. Filsfils *et al.*, "SRv6 interoperability report," IETF, Internet-Draft, September 2018. [Online]. Available: https://tools.ietf.org/html/draft-filsfils-spring-srv6-interop-01

[9] F. Duchêne *et al.*, "SRv6Pipes: enabling in-network bytestream functions," in *IFIP Networking 2018*, 2018.

[10] A. Abdelsalam *et al.*, "SERA: SEgment Routing Aware Firewall for Service Function Chaining scenarios," in *IFIP Networking 2018*. IEEE, May 2018.

[11] A. Abdelsalam *et al.*, "Implementation of Virtual Network Function chaining through Segment Routing in a Linux-based NFV infrastructure," in *2017 IEEE Conference on Network Softwarization (NetSoft)*, July 2017, pp. 1–5.

[12] P. Ventre *et al.*, "SDN Architecture and Southbound APIs for IPv6 Segment Routing Enabled Wide Area Networks," *arXiv preprint arXiv:1810.06008*, 2018.

[13] D. Lebrun, "Reaping the benefits of IPv6 Segment Routing," 2017. [Online]. Available: https://inl.info.ucl.ac.be/system/files/phdthesis-lebrun.pdf

[14] D. Lebrun and O. Bonaventure, "Implementing IPv6 Segment Routing in the Linux Kernel," in *Proceedings of the Applied Networking Research Workshop*. ACM, 2017, pp. 35–41.

[15] M. Konstantynowicz *et al.*, "Benchmarking and Analysis of Software Data Planes," Dec 2017. [Online]. Available: https://fd.io/wp-content/uploads/sites/34/2018/01/performance_analysis_sw_data_planes_dec21_2017.pdf

[16] S. Bradner and J. McQuaid, "Benchmarking Methodology for Network Interconnect Devices," Internet Requests for Comments, RFC Editor, RFC 2544, March 1999. [Online]. Available: https://tools.ietf.org/html/rfc2544

[17] S. Bradner, "Benchmarking Terminology for Network Interconnection Devices," Internet Requests for Comments, RFC Editor, RFC 1242, July 1991. [Online]. Available: https://tools.ietf.org/html/rfc1242

[18] "CSIT REPORT - The Fast Data I/O Project (FD.io) Continuous System Integration and Testing (CSIT) project report for CSIT master system testing of VPP-18.04 release." [Online]. Available: https://docs.fd.io/csit/master/report/_static/archive/csit_master.pdf

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

[20] "SRPerf - Performance Evaluation Framework for Segment Routing." [Online]. Available: https://github.com/SRouting/SRPerf

[21] "SRv6 - Linux kernel implementation." [Online]. Available: https://segment-routing.org/index.php/Main/HomePage

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

[23] "Linux netfilter hacking." [Online]. Available: https://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO-3.html

[24] "A thorough introduction to eBPF." [Online]. Available: https://lwn.net/Articles/740157/

[25] F. Clad *et al.*, "Service Programming with Segment Routing," IETF, Internet-Draft, July 2018. [Online]. Available: https://tools.ietf.org/html/draft-xuclad-spring-sr-service-programming-00

[26] "srext - a Linux kernel module implementing SRv6 Network Programming model." [Online]. Available: https://github.com/netgroup/SRv6-net-prog/

[27] "Trex stateless python api." [Online]. Available: https://trex-tgn.cisco.com/trex/doc/cp_stl_docs/index.html

[28] O. Ben-Kiki *et al.*, "Yaml ain't markup language," http://www.yaml.org/spec/1.2/spec.html, 2009.

[29] A. Hothan *et al.*, "NFVBench Documentation - Release 1.5.1," June 2018. [Online]. Available: https://media.readthedocs.org/pdf/opnfv-nfvbench/stable/opnfv-nfvbench.pdf

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

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