Universiteit van Amsterdam

System and Network Engineering

Segment Routing in Container Networks

Ben de Graaff

Supervisor: Marijke Kaat

July 12th, 2017

Abstract

Segment routing makes use of the source packet routing paradigm to allow for arbitrary forwarding paths. This can be used in container networks to build overlay networks with traffic engineering capabilities, and to deploy network functions such as load balancers. It further adds the ability to add metadata on a per-packet basis, which can be used to shape container policies. We show a practical implementation based on the IPv6 data plane based on the userspace functionality available on Linux, including eBPF.

Presentation:
http://rp.delaat.net/2016-2017/p95/presentation.pdf

Contents

1 Introduction

Segment routing is a set of open standards built upon the source routing paradigm, which allows nodes to select a forwarding path of their choosing[11][24]. Nodes can constrain paths to specific segments, between which the least cost path is then used. This allows a node to steer packet flows through any topological path or network function chain while enabling fast reroute in case of failure. The only required state is stored at the network edge, moving complexity away from the network core.

The standard further aims to simplify the network by doing away with unnecessary protocols where possible. This inherent simplicity and flexibility makes it a good candidate for a modern and scalable approach to software defined networking[7]. Despite the standard having been in development for a number of years already, there is not a large body of research or information on how to effectively utilize segment routing with regard to end-to-end host connectivity.

Container networking, which involves interconnecting containers on multiple hosts and containers to the outside world, is a good candidate for experimenting with segment routing in practice. Typical container network deployments use features such as overlay networks for reachability and multi-tenancy, network functions such as firewalling and load balancing, and could benefit from traffic engineering for optimal network capacity usage. Because container platforms are primarily based on Linux it is useful to investigate in which ways a Linux based container platform can leverage segment routing to perform these tasks.

Container networks can span over multiple network or administrative domains, for example when spanning multiple geographical sites. Segment routing is not limited to a single domain or data center, because transit domains or routers do not need to directly support segment routing as long as they are capable of routing IPv6[22]. The segments traversed by the packet can remain visible end-to-end which allows use of this data to make informed policy decisions.

This project aims to research and document the current possibilities, limitations, and usability of segment routing by creating a proof of concept implementation of a container network based on segment routing on Linux. In order to gain insight into segment routing and how segment routing can be used to build container-to-container networks the following research question needs to be answered:

1. What are the possibilities for container-level integration of segment routing based on current standards, hardware, and software?

Based on the available functionality provided by segment routing, the following sub-questions arise regarding the requirements of container networks and the underlying topology:

2. What is a practical method for applying segment routing policies in container

networks from the container host?
3. What are the infrastructure and application requirements to use segment routing effectively from container to container?

2 Background

This section provides the necessary background information to understand the basic operation of segment routing. It also introduces container networks and related terminology, the Linux Extended Berkeley Packet Filter interface, and the Vector Packet Processing software router.

2.1 Segment routing

Segment routing comes in two data plane flavors with forwarding based on MPLS[12] (often abbreviated as SR), and directly on top of IPv6 using the segment routing header[22] (SRv6). It is explicitly based on current, open standards and does not introduce any new protocols. Instead it builds on existing routing protocols such as BGP, IS-IS, and OSPF, on path computation protocols such as PCEP[27]. In fact, it has as design goal to make existing protocols such as RSVP-TE[3] largely unnecessary[9][12].

A segment routing domain consists of a topology defined by its segments and their identifiers (SIDs). When addressed, each segment effectively expresses the next action a packet will take in the network, e.g. whether it is to forward a packet using the least cost path, on a specific interface, or to deliver the packet to an application[13]. These identifiers are expressed as MPLS labels or IPv6 addresses, depending on the underlying data plane.

The primary segment types are the node, prefix, and adjacency segments. The *node segment* identifies a specific device, e.g. its loopback interface. A *prefix segment* specifies the reachability of a protocol specific (e.g. IP) prefix via that segment, using the least cost path as per usual IGP. An *adjacency segment* identifies the interface(s) towards a specific neighbor, without further least cost path considerations.

Segment routing on the IPv6 data plane is implemented by adding a segment routing header, shown in figure 1, on a per-packet basis. The segment list contains a list of *all* the segments that will be traversed, ordered from final destination to initial segment. To route a packet over a segment routed domain the destination address of the packet is set to the next desired segment, which directly addresses a specific SRv6-capable device. When a router receives a packet not explicitly destined to itself the packet is routed using typical best path first algorithms, meaning that intermediate routers do not need to inspect the routing header or even be SRv6-capable at all.

The segment routing header can optionally include additional data, which is passed

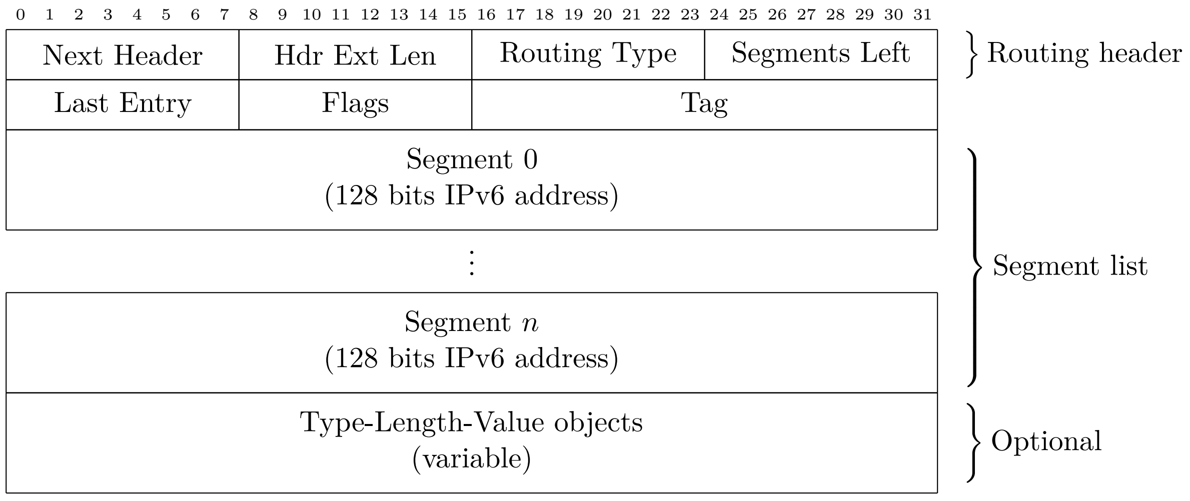

Figure 1: The IPv6 segment routing header (SRH) is a routing header with type 4.

along to the destination node. This data is encoded as *type-length-value* objects with a maximum length of 256 bytes. One use of this extension mechanism is to sign the segment list by adding an HMAC signature, useful for allowing routers to authenticate the SRH when routing between multiple domains.

The MPLS data plane is implemented in a conceptually similar manner, but has some important technical differences. One major difference is that segment identifiers are expressed as MPLS labels. Because MPLS labels are locally managed within an administrative domain using segment routing between MPLS domains may require coordination or label translation[12]. In contrast to the IPv6 data plane, the segment path is lost when a next segment is activated as the label is popped from the stack. There is also no mechanism to include additional metadata in packets.

2.1.1 Network programming with segment routing

Network programmability is achieved in segment routing by attaching specific network functions to SIDs. In the case of IPv6 such a SID typically follows a locator/function style, where the left-most L address bits specify the locator and the right most F bits the function. Note that these SIDs do not necessarily need to be originated in an IGP or be routable at all. A number of standard behaviors are defined in [13] and the most relevant functions are summarized in this section.

A router inspects the segment routing header only when a packet is addressed to itself based on the destination address of the packet. If the destination address matches an entry in its local SID table the function associated with that SID is executed for the packet. These functions are called *endpoint functions*. Generally this means the *segments* *left* field is decremented and the address of the subsequent segment is placed in the destination address field, followed by e.g. forwarding, encapsulation, or decapsulation, or a combination of these actions. In practice the behavior of these functions is entirely up to the implementer. Functions are chained together by their adjacency in the segment list.

A number of transit behaviors are proposed, typically used when traffic enters (a part of) the network. Because the destination address will not match a local address when processing the packet on a transit router, the segment routing header is never acted upon. It is however possible to insert or extend the segment list by using the `T.Insert` transit behavior, or optionally by encapsulating the entire packet with `T.Encaps`, or the frame with `T.Encaps.L2`, which can be used to build virtual private networks.

When a packet is addressed to a specific device via an endpoint function, the segment routing header will be acted upon. The most basic endpoint function is the `End` function, which does nothing but update the destination address to the next segment. Traffic engineering can be accomplished by specifying specific links using the *adjacency SID*, effectively implementing a layer 3 cross-connect (`End.X`). Virtual private networks can be built with support for decapsulation of the outer IPv6 header on layer 3 (`End.DX4`, `End.DX6`), or even on layer 2 (`End.DX2`).

Because `End` and `End.X` functions are directly tied to the topology they are best expressed as auxiliary information in existing interior gateway protocols, such as IS-IS[23] or OSPF[25]. Signaling of functions over multiple domains is proposed using BPG-LS[21].

2.2 Container networks

Container networks consist of container hosts with numerous containers, potentially spanning multiple data centers and administrative domains. The container hosts run a container platform such as Docker[8] or LXC[19], and either the built-in networking solution that comes with the platform or an external integration such as Weave[28], Cilium[6], et cetera. Three distinct types of end-to-end connectivity exist within such a network, namely traffic between local containers, between containers on different hosts, and traffic between the containers and external domains.

The containers may use user-defined addressing schemes that are not directly routable on the underlying network, e.g. some container platforms even allow virtual layer 2 domains to be created over multiple hosts. To ensure container reachability these container platforms deploy overlay networks by creating tunnels between the various container hosts. Overlays can also be used to create isolated networks for sets of containers, commonly used in multi-tenant container networks.

2.3 Extended Berkeley Packet Filter

The Berkeley Packet Filter (BPF)[29] interface provides a fast, kernel-based method for filtering and manipulating network packets. BPF programs are built using a simple bytecode language, for which a frontend for LLVM exists to translate C code to BPF objects[26]. Optional *just in time* compilation in-kernel can be enabled for increased performance, and there are even *Smart NICs* on the market for which the bytecode can be translated to NIC specific code[18]. The eBPF interface extends the capabilities of BPF programs by adding the `call` instruction capable of calling a predefined set of kernel helper functions[5], for example to read and write packet data from a socket buffer, or to redirect delivery to a different network interface.

In order to ensure stable kernel operation, the functionality of eBPF programs is severely limited. Before the program is accepted by the kernel a validator is run to check if the program terminates in a reasonable time. To make this validation possible, the total instruction count of the program is limited, the program can only jump forward (e.g. allowing only unrolled loops), and can only call the helper functions with pre-validated parameters. Furthermore, the stack size is very limited and out-of-bounds reads and writes are prevented by rejecting access to unvalidated offsets, i.e. without proper boundary checks.

eBPF programs can be inserted in the network pipeline in multiple ways. They can be attached directly to all interface ingress or egress traffic (*tc qdisc clsact*), via routes (*ip route*), and to tunnel interfaces (*ip tunnel*). Which kernel helpers are available is limited depending on the context the program is placed in. For example, packet modification is not allowed in IP tunneling use cases.

For performance reasons eBPF programs are often compiled with a number of fixed parameters instead of using dynamic runtime configuration. However, userspace applications can communicate with eBPF programs using so called *maps* of various types: arrays, hash maps, tries, et cetera. Map entries can be added and removed via the `bpf` system call on the map's file descriptor. These maps can be exposed via a special filesystem called *bpffs*. This allows eBPF programs to still be dynamically influenced by external applications.

2.4 Vector Packet Processing router

The Fast Data Project[1], part of the Linux Foundation, is a set of open source projects that make use of the Data Plane Development Kit to implement various high-performance applications, such as the Vector Packet Processing software router. This software router was initially developed by Cisco and is one of the first complete SRv6-capable routing implementations.

[1] https://www.fd.io/

The VPP router also offers great debugging facilities for its packet processing pipeline, which makes testing the interoperability of the segment routing header much easier. For example, a packet trace can be created by executing `vppctl trace add af-packet-input 3` on a VPP host, meaning the next 3 packets will be traced. After sending some traffic through the network the command `vppctl show trace` can be used to see a detailed list of actions executed on that packet, including helpful diagnostics when e.g. the SRH is malformed.

3 Related work

A number of technical workshops and use cases have been presented by companies such as Cisco, Bell Canada, Comcast, and others on the official segment routing site.[2] This work has provided valuable insights into the potential applications of segment routing and reveals a number of interesting conceptual approaches, for example how to build layer 3 VPNs and on replacing stateful multicast from the network core with unicast to the network edge with a 'spray' segment identifier, which lets the network edge then deliver packets to end customers via multicast. This highlights the ability the develop custom behaviors for segment identifiers.

Ahmed AbdelSalam et al. presented their model on deploying virtual network functions in SRv6-based topologies[1] at NetSoft 2017. While the paper had not been made available in time to be included in this research, a preliminary reading shows works focuses on the architectural requirements for deploying and chaining network functions, including supporting network functions that are not SRv6-aware. For their work they have implemented a competing SRv6 implementation for the Linux kernel called *srext*.

4 Implementation

Various technologies exist on Linux-based platforms which can be used to implement segment routing in container and overlay networks. With these technologies container platforms can make policy decisions on a per-container, per-tenant basis. The goal of this project is to create the proof of concept tools that can be used to evaluate segment routing in container networking scenarios. The tool to develop is a tool that can create arbitrary, virtual SRv6-capable topologies to be used as IPv6-only underlay. This allows us to attach virtual machines running a container platform to this topology. The virtual topology can then be used to experiment with segment routing, and to trace and verify the implementation of the segment routing policies.

The next step is to develop a program that can apply policies on traffic generated by the containers. The policy is based on a mapping between a tenant and container to a

[2]http://www.segment-routing.net/

list of segments that need to be applied. A program to achieve inverse on ingress also needs to be developed, i.e. mapping a segment list to a tenant/container. Based on these programs a simple overlay with public and with locally managed container addresses will be evaluated. This work is then expanded to a multi-tenant overlay network.

Once the proof of concept is able to apply and verify segment routing policies, network functions can be added to the setup. The goal is to experiment with how these functions can be implemented on a Linux based system and what their capabilities are in the context of segment routing.

4.1 Overlays and encapsulation

Overlay networks are an important aspect of container networks to ensure reachability between containers hosted on different machines, in a manner transparent to the container itself. Where direct routing is not possible these overlays are typically achieved using encapsulation to route traffic over the underlay. Some examples of common protocols in use today are GRE[15] (layer 3 encapsulation), VXLAN[20], and Geneve[14] (layer 2 encapsulation), although many other variants exist.

In principle IPv6 traffic needs no further encapsulation with SRv6. However, IPv6-in-IPv6 encapsulation is typically used to avoid modifying the original packet when adding the segment routing header at the network ingress point. The segment routing specification itself does not currently state whether such encapsulation is strictly required. Unless the network employs an IPv4/IPv6 translation technology, transporting IPv4 or Ethernet traffic over an IPv6-only network always requires such encapsulation, especially when the segment routing header is to be used.

Encapsulation may also be required for practical reasons, for example if source address filtering is used in the underlay, e.g. for traffic being sent over the Internet through domains which implement BCP 38/RFC 2827[10]. In this case IPv6-in-IPv6 encapsulation ensures that the source and destination addresses match their origin in the underlay.

SRv6 predefines all the required building blocks for implementing layer 2 and 3 overlays in the form of segment identifiers with cross-connect behaviors such as `End.DX2` (layer 2), `End.DX4` (IPv4), and `End.DX6` (IPv6). Cross-connects for IPv6 are necessary if the destination address of the targeted container is itself ‘unroutable’ on the underlay, therefore requiring a single extra segment indicating the location of the container host. In the case of traffic encapsulated with IPv6 the segment routing header may not even be required, because a segment routing header with a single segment identifier can simply be encoded as just the destination addresses of the outer IPv6 header. The forwarding implementation after decapsulation takes care of delivery from that point on.

Direct container addressing is slightly complicated by the inclusion of non-IPv6 support, in which case there is by default no ‘natural’ IPv6 address related the container as source or final segment. In this case, to simplify application design the destination IPv4 address can be converted to the IPv4-compatible form, i.e. *::w.x.y.z*, potentially with a prefix originating at container host. A more general approach would be to assign an IPv6 address per container regardless of whether IPv6 connectivity is used so that the container can be addressed in a manner that is agnostic to the protocol being transported, for example similar to IPv6 autoconfiguration by embedding the EUI-64 in the address.

Encapsulation adds additional limitations to the maximum transferable unit of packets traversing the overlay. For example, GRE is built on top of IP, while VXLAN and Geneve are built on top of UDP. Each of these protocols consist of a fixed header, followed by the layer 2 frame header (if applicable) and the original layer 3 packet. GRE is the smallest with a minimal header being 4 bytes, but also allows optional features such as checksumming, keying, and sequence numbers, enlarging the header. VXLAN and Geneve both include an 8 byte UDP header and 8 byte protocol header. Like with SRv6, Geneve allows arbitrary extension data to be added on a per-packet basis.

The IP-in-IP encapsulation suggested by SRv6 has a minimal amount of overhead, since with encapsulation in *any* case two IP headers always need to be included. However, if the underlay would otherwise be based on IPv4 the encapsulation overhead can still be less in such a network than just the size of a *single* IPv6 header. The additional overhead of the SRH is $8 + 16 * n$ bytes where n is the number of segments, plus the size of any additional metadata.

4.2 Multi-tenancy

A feature that is often implemented in container networks is multi-tenant overlays. These private overlays isolate different users of the underlay from each other by encapsulating or otherwise separating traffic. The packets will including a marking that allows a receiving node to differentiate between tenants. The most idiomatic way of implementing private overlays in SRv6 is to assign a unique segment identifier per tenant on each container host. This segment identifier then acts as a cross-connect with a specific (i.e. per tenant) routing table lookup to the containers on that hosts.

Combined with a location based addressing scheme for the containers, with prefixes that are derived from a host-specific prefix this approach requires very little configuration. The tenants can select arbitrary prefixes for their container addressing, although they must have a 1:1 mapping to addressing of the container host. These addresses can be mapped on the underlay to a tenant SID based on the host prefix as locator and the remaining bits can be used to encode the tenant identifier. An example is shown in section 5.4.

If the overlay network itself cannot be organized in this manner, for example because containers must be mobile or a legacy/non-hierarchical addressing scheme must be used, techniques from existing overlay technologies can be adopted and implemented in SRv6, for example exchanging of the *address to location* mapping using MP-BGP, discussed in section 6.1. In such cases the tenant SID of the host running the targeted container must be determined by a lookup based on the destination address of the container in relation to that tenant.

Isolation, like other multi-tenancy approaches, depends on the fact that the container itself cannot influence the tenant identifier or inject its own segment routing header. The same applies to ingress traffic coming from other network domains. The container platform must validate or reject the presence of the segment routing header in packets sent by the container to prevent this isolation from being broken.

4.3 Network functions

One of the major attractions of segment routing is the ability the steer traffic through network functions, so that functionality can be delegated to the network and need not be concentrated at the container host. In this section we describe some of the scenarios in which application controlled network functions could be useful in container networks. While investigating the potential applications of network functions to container networks, a number of patterns emerged.

The first pattern is a routing pattern. Traffic is routed through such a function to determine intermediate or penultimate segments that need to be traversed in order to successfully or optimally reach the final destination, useful when the sender does not want to maintain such routing state or does not have access to it. Examples of this would be container mobility, virtual/VPN routers (layer 2 or 3), service discovery, and load balancing. In this case the sending container host is aware that applying such a function is necessary to reach the destination address.

The next pattern does not involve packet or segment list modification, but instead has side-effects related to the traffic passing through. This primarily concerns monitoring applications, such as intrusion detection, network monitoring, and application monitoring. The monitoring itself does not necessarily need to take place in the function itself, for example Cisco has developed a so called *spray* behavior which can duplicate the packet to another destination.

The final example is a selective forwarding function. Examples of this are (stateless) firewalls, and dynamic DoS mitigation. One implementation of this was introduced in [16], which aims to prevent resource exhaustion attacks by handling TCP SYN packets for the protected service.

For this project we investigated the necessary steps to build and deploy these network functions. We implemented an example routing network function, described in section 5.5.

4.4 Ingress/egress application design

The design proposed in the section will be used to implement per-container policies making use of the capabilities of the segment routing header to build the (multi-tenant) overlays introduced in the previous sections, including the ability to steer traffic through network functions. Figure 2 shows an overview of the components in this design for a single container host. A container host has a single interface to a network that connects it to the other containers. Packets that arrive on this interface will be sent to the ingress policy function that validates the SRH and based on the segment list routes the packet to a container, delivers it locally, or drops the packet. Similarly, traffic originated from each container interface is sent through an egress policy function that applies a policy by inserting the SRH with the required segments.

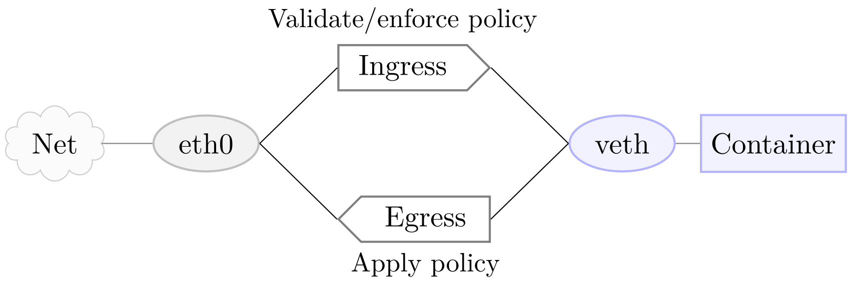

Figure 2: Diagram of the connectivity between components

For traffic coming in from the network we need to be able to map the final segments to a container. If there is only one segment remaining a container is being addressed directly. In overlay scenarios two segments need to be processed. In this case the penultimate segment signifies which overlay network is being addressed and the final segment addresses the container in that overlay. The list of remaining segments is used to look up the values described in table 1, which are then used by the ingress policy function to deliver the packet to the container. These behaviors can be mixed per container, for example if a container both has a publicly reachable IPv6 address and also participates in an overlay network.

Table 1: Information stored per ingress container

Field	Type	Description
`ifindex`	`uint`	Container interface
`mac`	`byte[6]`	Destination MAC address

The ingress policy is agnostic to prefix sizes of addresses used on the network since the full addresses are always used in the lookup process. There is no explicit need to know the address prefix of the host because if traffic is not addressed to a container on the host the lookup process will simply fail. Additional checks may include requiring additional segments to be present in arbitrary positions within the segment list, or the presence of certain extension data.

Table 2: Information stored per outgoing address prefix

Field	Type	Description
`is_overlay`	`bool`	Indicates if overlay segment is needed
`overlay_pfx`	`byte[]`	Address prefix on the underlay
`segments`	`sid[]`	Policy segments to apply

Outgoing traffic requires some additional configuration such as the interface index of the interface facing the network and the MAC address of the gateway router. The egress function applies a policy based on the destination address of the packet, for example to route the traffic over an overlay or to a public address. Effectively, this is used to map container IP addresses to other container hosts. The data structure used to store this information is shown in table 2. For the overlays, we assume a location-based addressing scheme as described in section 4.2 is used. For per-container policies we can map a unique container identifier such as its interface index to the container's tenant ID and per-container segments shown in table 3.

Table 3: Information stored per egress container

Field	Type	Description
`ifindex`	`uint`	Container interface
`tenant`	`uint`	Tenant ID
`segments`	`sid[]`	Policy segments to apply

As a simplification of the design, we will assume there is a single output interface on the container host with a fixed upstream gateway. Features such as ECMP and first-hop routing are left as future work. We further assume that in most cases traffic between local containers is a special case which can be handled by e.g. a shared bridge, although the inclusion of mandatory network functions means a packet will be forwarded to the network unconditionally. Exploring mechanisms for obtaining or exchanging dynamic policies is considered out of scope for the purpose of this proof of concept.

5 Results

This section describes the findings that were made during experimentation. The availability of hardware and software support for SRv6 is covered and a number of methods for implementing dynamic segment routing policies on Linux based on the experimental design discussed in the previous section are described.

5.1 Vendor support for segment routing

As shown in table 4, vendor support for segment routing is still in its early stages despite multiple vendors and companies being listed as having contributed to the technical discussions and standards. The dates listed are based on an estimated public availability of devices with support for segment routing, based on product change logs and published articles. The MPLS data plane variant has decent support, likely because it has been in development for much longer and because it mostly requires changes to the software stack related to label management and the IGP daemons. In the case of a MPLS data plane basic segment routing functionality can in the worst case be ‘emulated’ to some degree by manually and consistently assigning labels in a network.

Table 4: Vendors with hardware and software support for the required extensions and technologies to make use of segment routing

Vendor	MPLS	IPv6
Cisco	2014	2017-05
Ericsson	2015-02	-
Juniper	2016-08	-
Arista	2016	-

At the moment the Cisco NCS 5500 is the first and only device available that supports handling of the IPv6 SRH in hardware, though support for the ASR 9000 and ASR 1000 is also planned. Very little information has been published by other vendors that suggests they will release SRv6-capable hardware soon.

5.1.1 Programmable switches

Switches with a programmable data plane can in some cases also make use of segment routing. For example, Bell Canada has an implementation based on Barefoot Networks P4-capable switches.[3] Although the code has not yet been made publicly available, Bell Canada claims to have created a working implementation in about one month. A previous study[4] has been successful at parsing up to two IPv6 extension headers with P4 at rates up to 100Gbps, suggesting real world deployments with P4-capable devices are possible.

In contrast, OpenFlow based switches are not very suitable for working with SRv6.

[3]http://www.segment-routing.net/images/20170517-bell-barefoot-cisco-P4%20Workshop%202017%20v2.pdf

Because of a more rigid matching and action pipeline, adding the SRH is not possible. However, support for eBPF matching in OpenFlow proposed in [17] could potentially work around these limitations. Some OpenFlow switches, notably OpenvSwitch, have a maximum MPLS label stack depth of 3 labels, meaning supporting MPLS data planes may be possible but with limitations.

5.2 Virtual SRv6 topologies

Virtual topologies aid in the development and testing of the host-to-host behavior of SRv6. While segment routing is topology independent, features such as traffic engineering benefit from having multiple paths available in the network.

The source code listed in appendix A includes a script to start a user-defined virtual SRv6 topology. This script sets up LXC containers running a VPP router, interconnects the routers as defined, and finally adds static routes to every node in the network. Because SRv6 does not depend on any address prefix size, a common prefix size of 64 bits was chosen. Each router will have an `End` function SID where the right-most bits are all zero, which can be used to route traffic via that router. To select specific egress links, all routers have `End.X` function SIDs with an address in the form of `fd0x::1:y` where 'x' is the ID of the router and 'y' the ID of the adjacent router.

For every point-to-point connection a virtual Ethernet interface pair (`veth`) is created between the VPP container and the host. Routers are interconnected by placing the host end `veth` interfaces in a bridge, creating one bridge per link. While it is technically possible to create point-to-point `veth` pairs directly between containers instead of putting the interfaces in a bridge, LXC has no method of defining such a pair in its configuration. An additional benefit of bridges is that monitoring from the host is made easier and that arbitrary external interfaces can be placed in the bridge, for example to connect VMs or network functions to the topology.

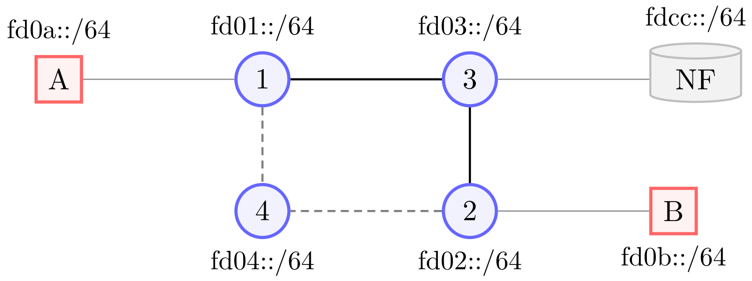

Figure 3: A simple example segment routing domain with 4 routers, 2 container hosts, and network functions

A simple topology used for experimentation is shown in figure 3. The topology consists of four VPP-based software routers. There are two container hosts, *A* and *B*, implemented as Ubuntu VMs running a number of LXC-based containers. Because there is currently no working routing protocol support in VPP the routes in the topology are statically defined by the script.

The virtual characteristics of the links can be changed by using the *tc qdisc netem* command, e.g. this topology includes two paths from A to B with unequal costs because of the simulated difference in latency characteristics. A basic example of using traffic engineering in this topology is varying the path from A to B from the primary path $1-3-2$ to $1-4-2$. This can be done by adding a SRH on egress from A that contains an `End` address of node 4 and the address of B as final segment.

5.3 Linux support for segment routing

Linux kernel and iproute2 version 4.10 released on February 19th, 2017 added basic support for SRv6, discussed in more detail in the next subsection. These versions are also required for supporting all of the eBPF features used during this experiment. Besides direct support for segment routing, the Linux kernel also offers userspace creation and processing of packets which can be used to implement segment routing.

5.3.1 Linux kernel behavior

Linux has the ability to add the segment routing header on egress. A fixed segment list can be added via a regular routing table entry, meaning both host-originated and transit egress traffic is supported. These routing entries support both adding the SRH 'inline' and by adding an outer IPv6 header with SRH. It is also possible for an application to attach a segment routing policy to a socket via `setsockopt` with the `IPV6_RTHDR` option.

```
ip -6 route add 2001:db8:1:/64
    encap seg6 mode inline
    segs 2001:db8:1::539,2001:db8:beef::1 dev eth0
ip -6 route add 2001:db8:1::539/128
    via fe80::8bd:4 dev eth0
```

Figure 4: An *ip route* example that adds segments to a route

Figure 4 shows an example command that sets up a route that adds the SRH inline (i.e. not using IPv6-in-IPv6 encapsulation) with two segments. Note that the first segment – which will replace the initial destination address – must have a (more specific) forwarding entry because it happens to fall within the route that is being encapsulated. The original destination address will automatically be added as final segment, meaning packets using this route will have a segment list with 3 entries.

In the current Linux implementation the delivery of packets with a segment routing header is not fully correct. The first issue is that the behavior specified in section 4.4 of the IPv6 standard (RFC 2460) is not followed:

> If *Segments Left* is zero, the node must ignore the routing header and proceed to process the next header in the packet.

While for security reasons it may be sensible to not process the instructions in the segment routing header when it is not explicitly enabled, the Linux implementation currently drops *all* packets addressed to itself with a SRH, even if the packet contains no further routing instructions, unless it is told not to.

Supporting the segment routing header at all therefore requires explicitly enabling segment routing via the per-interface `seg6_enabled` kernel option, *including* on the container side interface. This enables both the ability to deliver packets with zero segments left, but also processing of the segment routing header when there are still segments left.

With *seg6* enabled the behavior in the Linux kernel is fixed: it is not possible to attach specific behaviors to user defined segment identifiers or to disable endpoint functionality. If *segments left* is zero and the payload is another IPv6 payload, the packet is *always* decapsulated and reprocessed by the kernel. Otherwise the packet is delivered to the host via the normal path. In the case that *segments left* is non-zero, the kernel will always attempt to activate the next segment and forward to this address.

Another interesting behavior surfaced when we attempted a layer 2 optimization. The idea was to 'ignore' layer 2 addressing in order to reduce the amount of state required by the packet forwarding program and always use a broadcast MAC address, since all the container interfaces are effectively point-to-point links. This approach works when no segment routing header is present, but the Linux IPv6 routing header handling function has an explicit check that differentiates between traffic directed at the host (i.e. matching interface MAC, or `PACKET_HOST`) versus traffic directed at an IPv6 multicast address or non-matching MAC addresses, e.g. `PACKET_BROADCAST` and `PACKET_OTHERHOST`. The latter behavior is tied to the link layer address and not the active segment identifier, again unexpected compared to behavior proposed in the specification.

The only extension supported by this implementation is the HMAC extension, with support for both signing and verification with static keys. The Linux kernel implementation can be set to only accept authenticated headers by enabling the per-interface `seg6_required_hmac` option.

According to the authors of the *seg6* code using the segment routing header may also result in performance issues,[4] because of packet checksum mismatches due to SRv6's modification of the destination address. This should also be taken into consideration when using packet sniffers, as packets that contain protocols with a checksum that include the destination address but that do not yet have the final segment active may be marked

[4]http://www.segment-routing.org/index.php/Implementation/Issues

as having an incorrect checksum. IPv6-in-IPv6 encapsulation works around this issue because in that case the fields of the inner IPv6 header are used for checksum calculation instead.

5.3.2 eBPF

An initial implementation of the design proposed in section 4.4 was attempted using eBPF. As described in section 2.3, the eBPF interface allows applications to supply the kernel with packet filtering and manipulation programs. Programs can be attached to the ingress or egress point of an interface using the *tc* command. Due to the way the virtual Ethernet interfaces work traffic coming from the container to the host-end of the *veth* pair is also considered ingress from the perspective of the host.

The ingress eBPF program first verifies that the packet is indeed an IPv6 packet and not destined to a link-local address. Such traffic is simply passed along to the host. The next step is to build a *key* that will be used to look up whether the destination address is a known container. If the segment routing header is present and there is one segment left, the traffic is assumed to be targeted at an overlay. If there is no segment routing header or if there are no additional segments left, the traffic is assumed to be directly addressing a container instead. The key is set to (*segment[n]*, *segment[n - 1]*) for overlay traffic, or to (*destination*[5], *::*) otherwise. This key will be looked up in a hash map which maps the key to the container's interface and its MAC address, to which the packet will then be forwarded using `bpf_redirect`. The values in this map are provided by an external script, described below.

Applying policies to pure IPv6 ingress traffic requires only minimal packet modification. In the case of overlay traffic, the last segment is activated by updating the destination address and setting the segments left field to zero. The destination MAC address must always be updated, otherwise the frame will not be accepted by the network stack on the container side. However, encapsulated traffic requires more work because the outer IPv6 header and possible IPv6 extension headers must be removed as well.

To handle packet egress the eBPF program first looks up the sending interface index to determine its tenant ID and optional per-container segments. Next it looks up a prefix of the destination address to determine whether the address belongs to a known overlay. The prefix size for this lookup currently has a fixed size of 64 bits, but since kernel 4.11 an implementation of a longest prefix match trie can be used to support arbitrary prefix sizes.

If the destination prefix points to an overlay, the underlay address is determined by combining the resulting underlay prefix address, a fixed number of bits indicating this is a tenant SID, and the tentant ID. If there are no segments to be added as a result of this process, the packet is forwarded as-is. Otherwise room for the segment routing header

[5]Which is equivalent to *segment[n]*

must be added, the segments must be copied to the segment list, and the destination IP address must be updated to point to the first segment.

```
tc qdisc add dev eth0 clsact
tc filter add dev eth0 ingress prio 1 handle 1
    bpf da obj ingress.o sec do_ingress
tc qdisc add dev veth3CBDG1 clsact
tc filter add dev veth3CBDG1 ingress prio 1 handle 1
    bpf da obj egress.o sec do_egress
```

Figure 5: Commands to attach eBPF programs to interface ingress

Figure 5 shows the commands used to attach eBPF programs to an interface, by creating a *tc clsact* (classifier/action) hook and attaching the object to it. The *da* flag specifies that *direct packet access* is enabled.[6] In this mode the memory accesses of the packet data are automatically translated to the bytecode instructions that fetch this data from the socket buffer, instead of the application needing to manage data on the stack using `bpf_skb_load_bytes` and `bpf_skb_store_bytes` calls, allowing code to be written in a more 'natural' manner. This works under the condition that the kernel verifier can deduce that the access is within bounds of the socket buffer based on the presence of proper size checks in the code.

The eBPF programs use shared data structures called maps to communicate state with external applications. When using *tc* these maps are accessible over a special *bpffs* filesystem which can be mounted using `mount bpffs /sys/fs/bpf -t bpf`. The map will be located in `/sys/fs/bpf/tc/globals/$mapname`. A tool called `bpf-map` can be used to inspect these maps from the command line and we upstreamed a patch to modify these maps from e.g. a script[7], allowing us to dynamically modify container policies.

5.3.2.1 eBPF limitations

After experimenting with encapsulation it turned out that for the currently available Linux kernel version there is insufficient functionality exposed for removing or injecting IPv6 extension headers. Because of this, it is also not possible to decapsulate the outer IPv6 header in an IPv4-in-IPv6 scenario if the IPv6 packet has extension headers. The development team working on eBPF features in the Linux kernel acknowledged the lack of control over layer 3 extension headers and were also interested in adding such functionality.[8]

One of the potential functions is `bpf_skb_change_head`. This function adds some headroom at the start of the kernel socket buffer, but is intended to add layer 2 headers

[6]https://lwn.net/Articles/686677/
[7]https://github.com/cilium/bpf-map/pull/7
[8]https://github.com/cilium/cilium/issues/994

to packets for e.g. redirection. The socket buffer is set up such that the layer 2 and layer 3 headers are stored in separate segments, so this cannot be used to simply add a layer 2 header, copy the IPv6 header to the start of the packet, and then create a new segment routing header in the remaining headroom. It is also not possible to create an MPLS encapsulation this way because the socket buffer structure explicitly stores the Ethernet protocol type and will check the link layer header before forwarding the frame.

The second potential function is `bpf_skb_change_tail`, which can grow or trim the socket buffer size. This function has a performance penalty because it linearizes the entire socket buffer, preventing *generic segmentation offloading* from being used. Furthermore, it is also difficult to use due to the restrictive nature of BPF programs: the extremely limited stack size prevents copying full packet data to the stack, and there are no helper functions to directly move data around in the packet.

A further limitation is that checking of the HMAC extension is not possible because there is no support for cryptographic operations in eBPF. In general, verifying the presence of segment identifiers or extensions at non-fixed offsets is difficult to implement because of the difficulty of implementing dynamically sized loops in eBPF. Because of all the aforementioned limitations our design could not be fully implemented using just eBPF.

5.3.3 Userspace network handling

As the eBPF implementation currently does not meet all the requirements we set out with, an alternative approach was chosen which is based on Linux userspace packet handling. The design of the userspace application follows the same model as the eBPF design and can be used as a reference for the functionality that would be required in eBPF. The dynamic maps used in eBPF are replaced with static configuration.

There are several options for implementing both ingress and egress packet handling in userspace Linux. These datapaths may be slower than an eBPF-based approach, because every packet needs to be delivered to and processed by userspace resulting in extra copying and context switches, although optimizations can be applied such as sharing a ring buffers with the kernel.

One option is using *tun* and *tap* devices which enables a userspace program to receive layer 3 packets or layer 2 frames, respectively. Once received by the application, these datagrams can then be processed, or modified and sent out to an application-controlled interface using raw sockets. Ingress and egress traffic can be redirected to this interface using *iptables* in combination with *fwmark* by marking and redirecting traffic being forwarded from the container to the tapping interface, or directly using forwarding *ip rule* with an *in-interface* rule. One caveat of this approach is that egress forwarding cannot be done on a per-container basis because `veth` interfaces do not support forwarding. Instead they must be placed in a bridge from which traffic can then be forwarded. It is also not possible to replace the `veth` interface directly with a tun/tap interface because these interfaces cannot bridge Linux network namespaces.

Another option is to write an application based on the `PF_PACKET` packet sniffing API, which makes a copy of the packet and sends it to the application before kernel handling takes place. This approach will see both ingress and egress traffic, which must be filtered by matching the packet type in the application. The copying behavior means regular forwarding needs to be disabled when otherwise applicable routes are in place, otherwise packet will be forwarded unmodified by the kernel as well.

Similar to the eBPF programs these approaches are not transparent on layer 2, because forwarding the traffic over a raw socket requires filling in the link-layer headers. The destination MAC address must match the MAC address of the interface the traffic is being forwarded to, or the kernel will not handle the traffic. To some degree this can be bypassed when using point-to-point interfaces by using the broadcast MAC, if the traffic is not to be handled by the Linux kernel or if it does not have a segment routing header.

5.4 Overlays

The tools developed for this project based on the design proposed in 4.4 were used to implement the overlay scenarios described in sections 4.1 and 4.2. Ingress packet handling for overlay networks was implemented using both eBPF and in userspace. Because of technical limitations of eBPF described earlier ingress IPv4 decapsulation and egress functionality could not be fully implemented using eBPF.

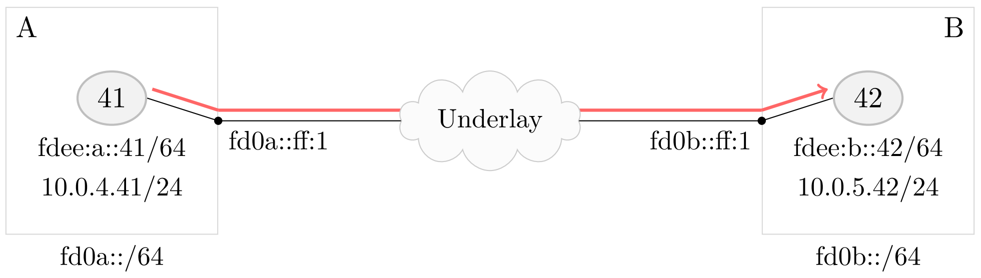

Figure 6: An example multi-tenant overlay network

An example multi-tenant overlay with two container hosts is shown in figure 6, based on the underlay in figure 3. For sake of brevity a single container from a single tenant is shown on each host. First a template was chosen for the overlay segment identifiers, in this case `fd0X::ff:Y` where X identifies the host and Y is a unique identifier for the tenant. The illustrated path represents container 41 belonging to tenant 1 at host *A* sending a packet to container 42 at host *B*. At container host *A* the segment list (fd0b::ff:1, fdee:b::42) will be inserted in traversal order. The underlay then routes traffic to container host B. At host *B* this segment list is used to map the traffic to the correct container. This overlay network also supports transporting IPv4 over the IPv6-only underlay. In this case host *A* encapsulates the IPv4 traffic with an outer IPv6 header and includes segment list (fd0b::ff:1, ::10.0.5.2).

5.5 Routing network function

As demonstration of deploying a network function in a container network a function was developed that can perform routing of opaque addresses. This scenario features an overlay network with addresses that are not directly routable, for which the container host has to add the routing network function SID to make successful packet delivery possible. When receiving a packet this network function resolves the actual destination host – based on the final segment – to where the container resides. It is able to route the packet to the final destination by adding segments to the segment list. The network function itself is backed by a simple mapping of addresses to container host SIDs.

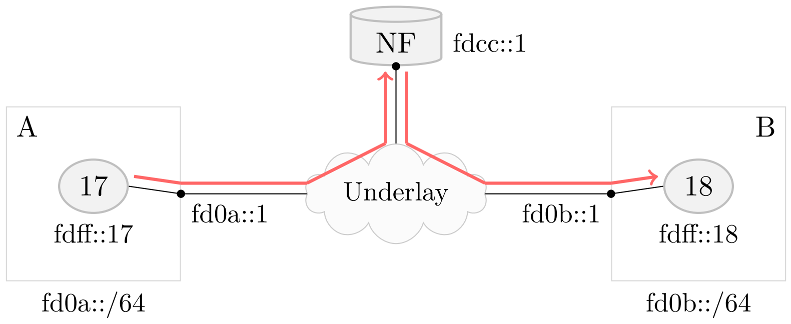

Figure 7: An example routing network function

A concrete example is shown in figure 7. In this scenario prefix fdff::/64 was chosen for opaque container addressing. When container 17 sends traffic to container 18 using its opaque address, the container platform will match this prefix and include the segment identifier of the routing network function so that the initial segment list becomes (fdcc::1, fdff::18), causing the traffic to be routed through the network function first. When the network function receives the packet, it will look up the location of the final segment and see that it can be reached at host *B* via fd0b::1. It updates the segment list to (fdcc::1, fd0b::1, fdff::18) and sends the packet back out again. Once the packet arrives at host *B* the container platform is able to deliver it to the container.

The implementation makes use of the same packet processing facilities as the userspace overlay applications. However, when deploying network functions on Linux care must be taken to avoid interacting with the Linux *seg6* implementation. For example, regular forwarding to a tun/tap interface will not work because the packets including a segment routing header addressed to one of the network interfaces causes the *seg6* processing to either act on the segment routing header or if it is disabled to reject the packet. Therefore the implementation is currently limited to the `PF_PACKET` packet sniffing API. Another workaround would be to only let Linux handle layer 2 forwarding, such that the interface is not configured with an IP address. This would require implementing IPv6 neighbor discovery in userspace so that the neighboring router can reach the network function.

6 Discussion

In this section we describe some of the potential limitations and developments of segment routing to container networks. The design and implementation of segment routing and SRv6 are still a work in progress, as both the standards themselves and the implementations have not been finalized yet.

6.1 Container control plane

While the proof of concept uses static configuration, container platforms typically implement a control plane to dynamically grow and shrink the container cluster. Container platforms that support clustering will already have well established communication protocols used to establish their clusters, e.g. based on shared key-value stores or direct information exchange.

In order to establish an overlay between hosts in the cluster, the hosts must exchange a set of parameters for that overlay. This includes an identifier for the overlay such as a tenant identifier, the segment identifier that must be used to route traffic to that host, and the addresses of the containers that are reachable on this overlay. Container address information should preferably be in an aggregated form for sake of efficiency.

A container network with complex addressing, such as containers with arbitrary layer 2 addresses spread throughout the network, can adapt existing approaches to support segment routing. For example, an existing approach which is used in VXLAN which uses MP-BGP can also be adopted to SRv6. In this case the exchange of *Virtual Tunnel Endpoints* used by VXLAN can be substituted with the exchange of overlay segment identifiers, which is a conceptually similar approach.

Traffic engineering can be part of the control plane as well, for example by exchanging traffic profiles between container hosts. Protocols to do path computation between end-hosts directly have not been proposed, instead existing approaches rely on a centralized path computation element based on PCEP. Alternatively, traffic engineering could be left open to be implemented by the core network itself. Here adequate marking of traffic may play an important role.

6.2 Further developments based on SRv6

It can be useful for hops in the network – such as routers or network functions – to add or replace segments in the segment routing header, for example to enforce security or routing policies. Truncation of the segment list at the network boundary could also be used to hide implementation details of the network, or simply to reduce the segment list overhead. The current specification prohibits direct modification of the segment list if the destination address is not within the domain of the modifying party, and whether such modification would be acceptable is an open point of discussion in the segment routing community.

Source routing based policies are inherently unidirectional. This is not necessarily an issue, but does prove to be a limitation where ingress traffic to a container may not have been passed through a (security) network function such as a firewall. Coupled with potential fail-open behavior, e.g. when new ingress paths become available or if the ingress point otherwise no longer applies an ingress policy, use of security functions need to be carefully designed to avoid unintended exposure, for example by enforcing the presence of security related SIDs when a packet is received by the container host. The ability for an end-host to request a path from an ingress point *to* that host is not currently described in the specification.

The addressing scheme proposed in section 4.2 which includes a predictable tenant SID as destination address can prove to be useful in allowing the underlying network to make (QoS) policy or traffic engineering decisions on a per-tenant basis, without having to be aware of the tenant's container addressing scheme. This is another potential reason to unconditionally use IPv6-in-IPv6 encapsulation, despite it not being strictly required, so that both outer source and destination addresses can fit in this scheme.

6.3 Improvements in eBPF

Support in eBPF for adding and removing arbitrary layer 3 extension headers was added July 3rd, 2017[9] to be introduced in Linux 4.13. The function `bpf_skb_adjust_room` adds the ability to add or remove a number of bytes directly after the IPv4 or IPv6 header. This missing function would allow us to implement adding and removing the segment routing header using eBPF. However, this function does not help if the SRH is (to be) placed at any position other than directly after the IPv6 header. Adding or removing segments to an existing segment routing header is also made difficult by this design, making it less useful for implementing network functions.

6.4 Security implications

The SRv6 concept shares similarities with the earlier ill-fated IPv6 routing header type zero (RH0), which was deprecated for its potential use in (reflected) denial of service attacks[2]. SRv6 attempts to solve these issues in multiple ways. Since container networks are in most cases exposed to external networks, the security of the container network against outside influence should be taking into consideration.

SRv6 can only have an impact on a network if the destination address of the packet is within that domain, as otherwise (by design) the routing header is never acted upon. At the network ingress points a security policy needs to be applied. For example, a source

[9]https://git.kernel.org/pub/scm/linux/kernel/git/davem/net-next.git/commit/?id=2be7e21

based ACL may be implemented (keeping in mind the potential for IP spoofing attacks) and the HMAC-based authentication of the SRH can be enforced. In any case, segment identifiers related to overlays must be protected from unauthorized access.

Contrary to the initial design of RH0 with SRv6 not all hosts need to support arbitrary routing to be enabled by default. Instead support for source routing is enabled by explicitly assigning behaviors to segment identifiers. Note that the Linux kernel implementation does *not* follow this model: enabling *seg6* on an interface – required in order not to drop traffic with the segment routing header *at all* – enables endpoint functionality described in section 5.3.1 on its local IPv6 addresses.

The segment routing standard does not offer any new transport encryption or authentication options with regards to protecting the other headers or payload of the packet. It is expected that end-to-end authentication and confidentially will be based on existing protocols such as TLS. The IPSec support built into IPv6 can also be used to provide end-to-end security, although because of the manipulation of the destination address field the authentication header may not appear to be valid while in flight.

7 Future work

The documentation of the Linux segment routing implementation states that a method that will allow applications to handle endpoint function behavior via a Netlink interface is planned. There is no further documentation or roadmap available as of yet. However, since such a method will likely be the intended way of using SRv6 on Linux it would be interesting to research the general design and application of this interface.

With SRv6 it is possible that the segment list remain present in the packet up to and including at the final destination. Because segment routing does not rely on path reservations, it might be interesting to use this segment list in reverse order for symmetric path signaling. Furthermore, the SRH has a flag that enables OAM messaging. This can be used for out-of-band path signaling as well, e.g. by including an extension with path parameters.

Based on the soon to be released functionality the eBPF-based approach to segment routing can be finalized. It would be interesting to test the performance of this implementation against e.g. the Linux kernel implementation and against the DPDK-based VPP. It would also be interesting to compare the performance of interpreted eBPF, JIT-compiled eBPF, and eBPF using the eXpress Data Path optimization.

8 Conclusion

In this paper we have shown that while segment routing in IPv6 is still in its (relatively) early development stages, it is possible to build experimental setups and evaluate its behavior in practice. The concepts and functionality offered by SRv6 integrate well with real-world use cases that benefit from software defined networking, such as container networks.

Building virtual overlays between container hosts and routing of services through network functions is easily achievable, even if the underlay network infrastructure does not itself actively participate in segment routing. Because the behavioral instructions are encoded into the packet, a noticeable reduction in the maximum transferable unit size should be taken into consideration when designing such a network and related applications, although such overhead is comparable to existing encapsulation mechanisms.

Host-driven SDN goals do not need to conflict with traffic engineering policies in effect on the network. A host in a segment routing domain can either apply traffic engineering policies directly, point the packet at a policy function, or allow ingress core routers to extend the segment list for traffic engineering. It is also possible to mix SRv6 with segment routing based on MPLS.

While the Linux kernel implementation cannot be used directly as building block for using SRv6 in container networks, the kernel offers adequate interfaces to allow userspace applications to experiment with new extensions to IPv6. When processing the segment routing header in userspace the in-kernel implementation can get in the way, but workarounds are available. The development of new interfaces such as eBPF greatly simplifies experimental network handling in a performant way, but is severely limited by the API exposed to these programs.

A Source code

All source code produced during this experimentation has been made publicly available under the GNU General Public License. For ecological reasons it is not included in this document, but instead can be found at https://bitbucket.org/uva-sne/segcon6/.

This source code includes the following:

- A script to create virtual topologies
- eBPF programs for ingress and egress (untested)
- Userspace tools based on tun/tap and `PF_PACKET`
- A network function template and routing example
- Linux kernel tracing scripts for eBPF and for the IPv6/*seg6* network stack

References

[1] Ahmed AbdelSalam et al. *Implementation of Virtual Network Function Chaining through Segment Routing in a Linux-based NFV Infrastructure.* Tech. rep. July 2017. URL: https://arxiv.org/abs/1702.05157.

[2] J. Abley, P. Savola, and G. Neville-Neil. *Deprecation of Type 0 Routing Headers in IPv6.* RFC 5095. RFC Editor, Dec. 2007.

[3] D. Awduche et al. *RSVP-TE: Extensions to RSVP for LSP Tunnels.* RFC 3209. RFC Editor, Dec. 2001.

[4] Pavel Benácek, Viktor Puš, and Hana Kubátová. *Automatic Generation of 100 Gbps Packet Parsers from P4 Description.* Nov. 2015. URL: https://h2rc.cse.sc.edu/2015/h2rc-p2.pdf.

[5] Daniel Borkmann. *Linux tc and eBPF.* Jan. 2016. URL: https://archive.fosdem.org/2016/schedule/event/ebpf/attachments/slides/1159/export/events/attachments/ebpf/slides/1159/ebpf.pdf.

[6] *Cilium.* URL: https://github.com/cilium/cilium.

[7] Luca Davoli et al. *Traffic Engineering with Segment Routing: SDN-based Architectural Design and Open Source Implementation.* Tech. rep. Dec. 2015. URL: https://arxiv.org/abs/1506.05941.

[8] *Docker.* URL: https://www.docker.com/.

[9] Adrian Farrel and Ron Bonica. *Segment Routing: Cutting Through the Hype and Finding the IETF's Innovative Nugget of Gold.* July 2017. URL: http://www.ietfjournal.org/segment-routing-cutting-through-the-hype-and-finding-the-ietfs-innovative-nugget-of-gold/.

[10] P. Ferguson and D. Senie. *Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing.* BCP 38. http://www.rfc-editor.org/rfc/rfc2827.txt. RFC Editor, May 2000. URL: http://www.rfc-editor.org/rfc/rfc2827.txt.

[11] Clarence Filsfils et al. *Segment Routing Architecture.* Internet-Draft draft-ietf-spring-segment-routing-11. IETF Secretariat, Feb. 2017. URL: http://www.ietf.org/internet-drafts/draft-ietf-spring-segment-routing-11.txt.

[12] Clarence Filsfils et al. *Segment Routing with MPLS data plane.* Internet-Draft draft-ietf-spring-segment-routing-mpls-06. IETF Secretariat, Jan. 2017. URL: https://www.ietf.org/internet-drafts/draft-ietf-spring-segment-routing-mpls-06.txt.

[13] Clarence Filsfils et al. *SRv6 Network Programming.* Internet-Draft draft-filsfils-spring-srv6-network-programming-00. IETF Secretariat, Mar. 2017. URL: https://www.ietf.org/internet-drafts/draft-filsfils-spring-srv6-network-programming-00.txt.

[14] Jesse Gross, Ilango Ganga, and T. Sridhar. *Geneve: Generic Network Virtualization Encapsulation*. Internet-Draft draft-ietf-nvo3-geneve-04. http://www.ietf.org/internet-drafts/draft-ietf-nvo3-geneve-04.txt. IETF Secretariat, Mar. 2017. URL: http://www.ietf.org/internet-drafts/draft-ietf-nvo3-geneve-04.txt.

[15] S. Hanks et al. *Generic Routing Encapsulation (GRE)*. RFC 1701. RFC Editor, Oct. 1994.

[16] A. H. M. Jakaria et al. “VFence: A Defense against Distributed Denial of Service Attacks Using Network Function Virtualization”. In: *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. June 2016, pp. 431–436. DOI: 10.1109/COMPSAC.2016.219.

[17] S. Jouet, R. Cziva, and D. P. Pezaros. “Arbitrary packet matching in OpenFlow”. In: *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*. July 2015, pp. 1–6. DOI: 10.1109/HPSR.2015.7483106.

[18] Jakub Kicinski and Nicolaas Viljoen. *eBPF Hardware Offload to SmartNICs: cls bpf and XDP*. Oct. 2016. URL: https://netdevconf.org/1.2/papers/eBPF_HW_OFFLOAD.pdf.

[19] *Linux Containers*. URL: https://linuxcontainers.org/.

[20] M. Mahalingam et al. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. RFC 7348. http://www.rfc-editor.org/rfc/rfc7348.txt. RFC Editor, Aug. 2014. URL: http://www.rfc-editor.org/rfc/rfc7348.txt.

[21] Stefano Previdi et al. *BGP Link-State extensions for Segment Routing*. Internet-Draft draft-ietf-idr-bgp-ls-segment-routing-ext-02. http://www.ietf.org/internet-drafts/draft-ietf-idr-bgp-ls-segment-routing-ext-02.txt. IETF Secretariat, June 2017. URL: http://www.ietf.org/internet-drafts/draft-ietf-idr-bgp-ls-segment-routing-ext-02.txt.

[22] Stefano Previdi et al. *IPv6 Segment Routing Header (SRH)*. Internet-Draft draft-ietf-6man-segment-routing-header-06. IETF Secretariat, Mar. 2017. URL: https://www.ietf.org/internet-drafts/draft-ietf-6man-segment-routing-header-06.txt.

[23] Stefano Previdi et al. *IS-IS Extensions for Segment Routing*. Internet-Draft draft-ietf-isis-segment-routing-extensions-09. http://www.ietf.org/internet-drafts/draft-ietf-isis-segment-routing-extensions-09.txt. IETF Secretariat, Oct. 2016. URL: http://www.ietf.org/internet-drafts/draft-ietf-isis-segment-routing-extensions-09.txt.

[24] S. Previdi et al. *Source Packet Routing in Networking (SPRING) Problem Statement and Requirements*. RFC 7855. RFC Editor, May 2016.

[25] Peter Psenak et al. *OSPFv3 Extensions for Segment Routing*. Internet-Draft draft-ietf-ospf-ospfv3-segment-routing-extensions-07. http://www.ietf.org/internet-drafts/draft-ietf-ospf-ospfv3-segment-routing-extensions-07.txt. IETF Secretariat, Oct. 2016. URL: http://www.ietf.org/internet-drafts/draft-ietf-ospf-ospfv3-segment-routing-extensions-07.txt.

[26] Alexei Starovoitov. *BPF in LLVM and kernel.* Aug. 2015. URL: https://linuxplumbersconf.org/2015/ocw//system/presentations/3249/original/bpf_llvm_2015aug19.pdf.

[27] JP. Vasseur and JL. Le Roux. *Path Computation Element (PCE) Communication Protocol (PCEP).* RFC 5440. http://www.rfc-editor.org/rfc/rfc5440.txt. RFC Editor, Mar. 2009. URL: http://www.rfc-editor.org/rfc/rfc5440.txt.

[28] *Weave.* URL: https://github.com/weaveworks/weave.

[29] Wikipedia. *Berkeley Packet Filter — Wikipedia, The Free Encyclopedia.* 2016. URL: https://en.wikipedia.org/w/index.php?title=Berkeley_Packet_Filter&oldid=716103041.