Master Thesis Autumn 23, Master of Science in Engineering – Computer Science
Eastern Switzerland University of Applied Sciences, Rapperswil

Leveraging eBPF for Intent-Driven Application-Centric End-to-End Segment Routing over IPv6

Julian Klaiber

***Advisor**: Prof. Laurent Metzger*
***Project Partner**: Bruce McDougall, Cisco Systems*

Abstract—**In today's digital era, the rapid evolution of applications significantly shapes how people work, communicate, and interact with technology. New hypes like artificial intelligence will become more and more critical. As these applications become increasingly complex and essential day-to-day, the underlying network infrastructure must evolve to meet their demanding requests. Traditional routing paradigms were once the backbone of digital communication and have stayed the same for decades. A fundamental shift of these paradigms is required to have more application-centric models. It is necessary to make changes that allow networks to adapt dynamically to the application's needs and scale accordingly to meet demand. Moreover, with the growing sophistication of applications comes an increase in network complexity. That complexity arises from the need to support a range of application behaviors - from bandwidth-intensive video streaming to latency-sensitive online gaming. That consequently leads to the network's more complex configuration, making management and optimization stressful. An intent-driven approach to network configuration handling emerges to counteract this complexity. Intent-driven networking outperforms traditional manual approaches by translating high-level business objectives into configurable operational objectives. That approach simplifies network management and reduces errors. The future of networking depends on its ability to adapt to the requirements of applications. This research focuses on this transformation, examining how integrating eBPF and SRv6 can accelerate the shift towards more adaptive, responsive, and application-centric network ecosystems. The study concludes with promising pathways for future research, including integrating emerging technologies like 5G and IoT. This research contributes to the evolving field of network technology, offering insights and solutions that pave the way for more responsive, efficient, and application-aligned networking in the future.**

Index Terms—**Segment Routing, SRv6, IPv6, eBPF, VPP, Linux, Programming, Dataplane, Golang**

I. Introduction

The following sections will offer background and current state of networking to provide a better understanding of this research. That will be followed by a clear statement of the problem as well as the objectives of the research. Finally, the text will conclude by pointing out the specific contributions of this research and where they are applicable.

A. Background

In modern networking, the current paradigm prioritizes routing decisions within the network. This approach signifies that the decision-making process regarding how data is routed (including path selection and prioritization of traffic) is managed internally by the network's infrastructure. These decisions are based on the network's configuration and protocols rather than being influenced or directed by external factors. That limits the development and deployment of applications. The current model does not empower application developers to influence how network traffic is routed to and from their applications or devices. Instead, this critical aspect is entirely delegated to network teams, which configure the routing policies on the routers.

This traditional model has some implications that make it challenging to handle the speed of technological innovations. First and foremost, it creates a barrier between application requirements and network behavior. Application developers, who best understand their applications' performance and reliability needs, need help to specify or influence the path that network traffic takes. That can lead to inefficiencies and suboptimal performance, as the network may need to be aligned with the application's specific needs.

Furthermore, this model leads to a multiplication of routing policies scattered across the network. Each application or device requiring specific traffic handling needs a unique policy. Managing these numerous policies can become a logistical nightmare, leading to increased complexity and a higher likelihood of misconfigurations. It also results in a static network configuration that is difficult to adapt as application requirements evolve.

The demand for more intelligent and flexible network routing is evident. A paradigm shift is required to move from a router-centric decision-making process to a more dynamic, application-aware concept. This shift involves enabling appli-

Acknowledgments: I would like to express my profound gratitude to Prof. Laurent Metzger for his guidance and supervision during my Master's program. Additionally, I sincerely thank Severin Dellsperger and Bruce McDougall, whose insightful discussions have significantly enriched my research experience.

cations and developers to be more active in defining how their traffic is managed and routed within the network. The goal is to create a network that is responsive to the demands of applications and adaptive to their changing needs.

Emerging technologies such as Extended Berkeley Packet Filter (eBPF) [1] and Segment Routing over IPv6 (SRv6) offer solutions to realize this vision. eBPF, for instance, provides a mechanism to execute custom code within the kernel, opening up possibilities for more granular and intelligent traffic handling. Segment Routing over IPv6 introduces a more flexible and scalable way to route packets, allowing paths to be determined at the ingress and enabling the source-based routing approach.

Combining and integrating these technologies can transform network routing, making it more application-centric. That would enable application developers to directly influence how their application traffic is handled, leading to optimized network performance and simplified network management. Developing and deploying such an intelligent routing concept is a technological advancement and a strategic imperative to meet the evolving demands of modern applications and networks.

In this context, the proposed research aims to explore and leverage eBPF for intent-driven, application-centric end-to-end segment routing over IPv6, aiming to bridge the gap between application requirements and network routing decisions. This approach promises a more cohesive, efficient, and responsive network, tailored to the specific needs of diverse applications and devices.

B. Problem Statement

As mentioned in I-A, a significant challenge exists in the current networking landscape: the disconnect between application requirements and network routing decisions. This challenge arrives from the traditional network routing paradigm, where routing is done on every router in the network. The network has no direct input or guidance on how to treat the application traffic. While Quality of Service (QoS) mechanisms exist and offer a way to steer traffic by prioritizing certain types of data, more is needed for a genuinely application-centric approach. QoS typically operates on broader traffic categories rather than the nuanced needs of individual applications. This results in a network that, despite having some level of traffic management, is still not fully aligned with the requirements of the applications. Due to this potential mismatch between network capabilities and application demands, critical issues can arise:

- **Lack of application-centric routing:** Current routing mechanisms need to consider the unique requirements of applications adequately. Application developers who understand the needs of their applications are not able to influence how the traffic to and from their applications is routed. That can lead to a mismatch between an application's needs and the network's delivery.
- **Complex policy management:** The network requires numerous routing policies to accommodate diverse application requirements. Defining each application's unique requirements on the necessary routers creates countless combinations. That increases the policy complexity, which also increases the possibility of misconfiguration.
- **Static and inflexible routing configuration:** Existing network routing configurations are often static, lacking the flexibility to adapt to dynamic application requirements or network changes. That can lead to paths that no longer meet application requirements.
- **Inefficiencies in network performance:** Without application-aware routing, network resources may be under or overutilized, and application performance may not be optimal. That can result in increased latency, reduced throughput, and overall suboptimal performance, which affects the user's experience.
- **Increased operational overhead:** The current model requires significant manual intervention by network teams to configure and maintain routing policies. That affects operational overhead but also slows down changes.

There is a need for a more intelligent, flexible, and application-aware network routing mechanism. Such a mechanism should empower application developers to influence routing decisions directly, ensuring network behavior is optimally aligned with application requirements.

C. Objective

The primary objective of this research is to develop an application that leverages an application-centric and intent-based routing approach, together with an end-to-end routing behavior. The application would bridge the gap between application requirements and network routing decisions, allowing for a more intelligent, flexible, and responsive network infrastructure.

- **Application-centric:** Develop an application that enforces application-specific requirements in the network. That will ensure network behavior is tailored to the demands of each defined application.
- **Intent-driven:** Implement a system that dynamically adapts network routing based on high-level operational intents or objectives. That approach enables the network to be more responsive and aligned with the goals of the operation teams.
- **End-to-end routing:** For optimized routing, achieve comprehensive control over the entire network path, from source to destination. That will lead to more efficient network resource utilization and an improved and simplified network backbone.

These objectives are centered around transforming traditional network routing paradigms to be more responsive, efficient, and aligned with the specific needs of today's applications.

D. Contribution

This research significantly advances networking by introducing an application that integrates intent-driven operations and application-centric design with an end-to-end routing

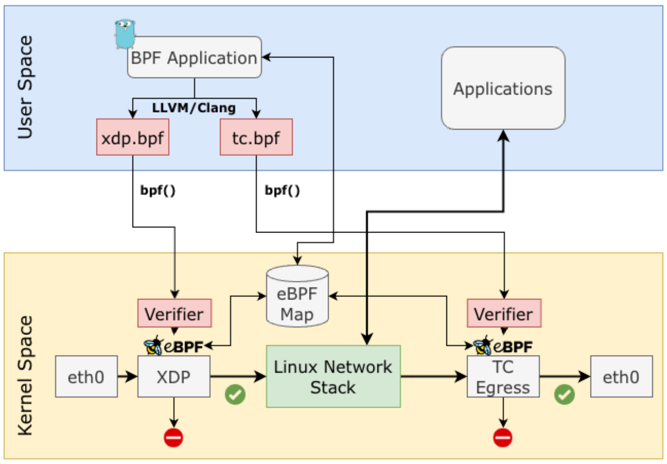


Fig. 1. eBPF Architecture

approach. The main contribution is that the routing decision functionality is brought directly to the end host.

At the core of this contribution is the shift towards an application-centric networking approach, where the application's requirements influence the routing decisions. This paradigm shift addresses the traditional disconnect between application requirements and network behavior, enhancing network performance and aligning user expectations with application demands.

Incorporating intent-driven networking principles represents a substantial jump in network management by translating actionable network configurations to high-level business and operational objectives. That allows for simple and easy configuration of the application's requirements.

Intent-driven networking and the application-centric approach are built on the foundation of enabling end-to-end routing communication with segment routing over IPv6. At the same time, end-to-end routing moves complexity away from the core network infrastructure, allowing for a simpler and more performant backbone.

Combining these elements within a single application significantly contributes to the networking field.

II. eBPF in a Nutshell

This section should provide an overview and deep dive into the most essential features of the Extended Berkeley Packet Filter (eBPF).

A. eBPF Overview

eBPF represents a significant evolution in how kernel-level packet processing is handled in modern operating systems. Initially designed for efficient packet filtering [2], eBPF has grown into a robust technology enabling programmable network packet processing, security monitoring, and performance analysis with minimal overhead on the system. Its core strength lies in the ability to run sandboxed code in the Linux kernel without changing kernel source code or loading kernel modules [3] [4]. eBPF is already in productive usage at significant players in the industry like Google [5], Facebook [6], Cloudflare [7], and many more. Applications like Cilium [8] or the open-source load balancer Katran [6] build their foundation with eBPF. It enables developers to write programs compiled into bytecode by the Linux kernel. This bytecode is executed in an isolated context within the kernel, ensuring safety and stability. eBPF programs can be attached to different hook points in the kernel, allowing them to be executed in response to specific events, such as network packets arriving or syscalls being made (see also Figure 1) [1].

In this research, we are concentrating on two specific hook points for packet processing. Booth hook points trigger the application when a packet arrives at the RX (receive) or TX (transmit) path. The first hook point that is considered is the eXpress Data Path (XDP) hook [9]. XDP is the earliest hook we can consider in packet processing. That means the hook is placed directly on the NIC driver or between the NIC driver and the Linux networking stack. Important to mention is that this happens before any memory allocation for the packet; it provides the raw packet data in the *xdp_buff* [10]. Contrary to the TC hook that already lays in the Linux networking stack and therefore is also allocating some amount of memory for the packet data structure, namely the *sk_buff*. TC can be used on either path, TX or RX, where XDP is only available on the RX path. Important to mention is also that the different locations of the hooks also directly influence the performance, where XDP at the earlier point delivers the most performance out of these two [11] [12].

Besides allowing the attachment of the eBPF program to different hook points, eBPF also has eBPF maps. The maps allow an eBPF program to share information with a user space application or store a state. Maps are an ideal way to transfer data and configurations from or to an eBPF program from a user space application. [13]

B. Cracking the Verifier

The verifier is the most critical component in the eBPF ecosystem and serves as a gatekeeper to ensure safety and stability in the kernel. Traditional kernel modules have unrestricted access to kernel memory and functions in contrast to eBPF programs that are restricted in their capabilities. The restrictions are essential as eBPF programs are dynamically loaded into the kernel at runtime, unlike kernel modules that are typically statically linked to the kernel. In contrast to eBPF programs, the kernel modules are a direct part of the kernel and, therefore, have the potential to crash the entire system when an error occurs. eBPF programs, on the other hand, are, per design, a safer alternative than kernel modules. They are running in a sandboxed environment within the kernel and are, therefore, unable to crash the entire system [14] [4] [3].

The eBPF verifier lays the foundation of the per-design runtime safety. By rigorously checking and analyzing the eBPF programs before execution. The verifier checks the program against strict safety rules, including illegal memory access, unbounded loops, and other unsafe operations that could destabilize the system. The verifier prevents potentially harmful eBPF programs from being attached to the kernel hooks at runtime when they do not satisfy the safety rules.

These safety rules are introducing several significant challenges in programming eBPF applications. It requires a deep understanding of the eBPF system, subsystem, and Linux kernel. Satisfying the eBPF verifier can be tedious because not only does the code functionality have to be correct, but the code also has to adhere to the verifier's safety rules. Many challenges can occur when working with dynamic and changing variables in the code and the limited stack size, also described in [15]. Meeting all these requirements can be time-consuming, technically demanding, and nerve-wracking.

C. The debugging Nightmare

Debugging eBPF programs can be challenging due to their unique execution environment within the kernel. Traditional debugging tools are often insufficient for eBPF code. Additionally, the per-runtime nature of eBPF programs makes debugging even more difficult. Most of the time, errors occur at runtime rather than during compilation. As a result, the debugging process can be very challenging because a general verifier log often does not provide enough information to locate the fault. Developers are left with using BPF tracepoints and leveraging BPF helper functions for logging to bring more visibility to the debugging process. All in all, debugging an eBPF application needs a lot of time and nerves.

III. The future of Networking

This section should provide an overview of Segment Routing over IPv6 (SRv6) and its features. It should also give insight into the concept of intent-driven networking.

A. Intent-Driven Networking

Intent-driven networking (IDN) represents a paradigm shift in managing and optimizing network configurations. There are several core concepts which have to be considered:

- **High-level policy definition:** IDN should allow a network administrator to define a policy on a high level. Instead of defining complicated rules on a router, the goal is to translate the high-level definition into executable configuration. For example, instead of manually defining a Quality of Service (QoS) policy for having high bandwidth, an administrator shall write: "A to B and back high-bandwidth." [16]
- **Dynamic network adaption:** In an IDN, the network should adapt its configuration automatically to changes.
- **Enhanced user experience and service quality:** By enabling high-level definitions for traffic and network control, the user experience is enhanced, and service quality is improved by reducing complexity and allowing non-experienced personnel to create intents for an application [17].

IDN can make the network more intelligent and adaptive to changes, making it capable of meeting the complex requirements of today's networks.

B. Integrating Segment Routing over IPv6 in Intent-Driven Networking

Building upon the principles described in III-A, Segment Routing over IPv6 (SRv6) [18] emerges as an enabler in providing these concepts. The source-based routing paradigm of SRv6, with its Segment ID (SID) list mechanism, already aligns with the intent-driven concept of having a distinct path control [19]. The SID list has to be abstracted to fulfill more concepts of IDN, especially the "high-level policy definition" concept. The abstraction represents a shift from the technical specifics of SRv6 configurations to intent-focused networking. The simplified expression of network objectives aligns with having an abstracted high-level policy. For example, an intent like "A to B and back high-bandwidth" would be translated to a specific SRv6 SID list that routes the traffic through the high-bandwidth path in the network. That allows an administrator to define a simple policy without knowing the exact segment list or having any know-how in network routing. In conclusion, the capabilities of SRv6 provide a robust framework for realizing the vision of IDN.

IV. Architecture

This section explores the architectural design. The architecture is centered around a client/server application and a central controller, focusing on DNS-based operations and dual operational modes for the client. An overview of the architecture can be found in Figure 2, with a more detailed explanation in the sections below.

A. Client/Server Application Design

The architecture employs a client/server model with distinct roles and functionalities.

1) Client Application: The application's client side is responsible for reading configuration information, which includes the client's intent regarding the application-centric network routing. This configuration informs how each packet should be encapsulated and decapsulated, ensuring it aligns with the specified intent. The client application also intercepts DNS replies to extract the destination IPv6 address. The client application operates in two modes:

- **Controller mode:** In this mode, the client utilizes high-level abstractions for intents, which are then queried from the controller. That allows for a more intuitive and simplified approach to specifying network routing behaviors.
- **Standalone mode:** This mode allows direct input of the SRv6 SID list, bypassing the controller. It is suitable for scenarios where the predefined intents are not required or immediate control over the SID list is preferred.

2) Server Application: On the server side, the focus is de- and encapsulating traffic. The application checks if the incoming packets are SRv6 encapsulated and, if so, processes the Segment Routing Header (SRH) data. The SRH data and the unique tuple of source IPv6 address and TCP/UDP port are stored to identify and appropriately encapsulate the outgoing traffic.

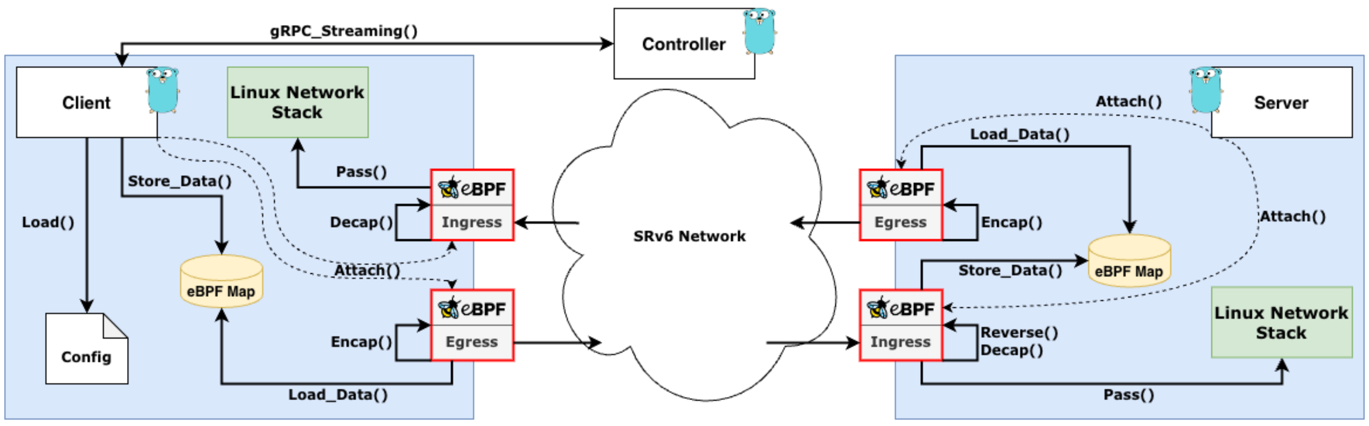


Fig. 2. Application Architecture

3) eBPF Integration for Dataplane Functionality: eBPF is used for all the dataplane functionalities of the client and server application. Two different eBPF applications are available for each client and server. There is a distinction between the ingress and egress applications.

- For outgoing traffic and, therefore, encapsulating logic, an eBPF application is attached to the Traffic Control (TC) egress hook. Both client and server egress eBPF applications are responsible for the correct traffic matching and SRv6 encapsulation.
- For the incoming traffic and, therefore, decapsulating the traffic when it is SRv6 traffic or intercepting DNS replies on the client, an eBPF application is attached to the XDP hook.

The eBPF applications are communicating to the userspace application over eBPF maps. The maps store the intents and tuples for matching outgoing traffic.

B. Central Controller Role

The central controller is a crucial component but not the primary focus of this work; for this work, a mock controller was used to simulate the behavior. Its primary function is to respond to the client's queries by providing the appropriate SID list based on the defined intents. The controller maintains an overarching network view, allowing it to calculate the SID list based on the required intents that the client requested. The controller reacts automatically to events in the network and will inform the client if there are changes in previously calculated paths. With this event-driven approach, the client will always have a correct SID list that he uses to encapsulate the corresponding traffic.

C. DNS-Based Operation

A crucial feature of this architecture is its DNS-based operation. When a domain name is specified for a connection to a server, the system intercepts the DNS replies to identify the destination IPv6 address; the destination address and the corresponding application port on the destination server are then stored as unique tuples. This information is critical for filtering and encapsulating the traffic according to the specified intents and the SID list. DNS-based operations are getting increasingly important, especially in the cloud-native-driven network world, where IP addresses constantly change, and we, therefore, have to use DNS as a layer of abstraction. [20]

V. Implementation

The following section will give more detailed insights into some specific implementation topics.

A. Development Environment

The development happened on a Cisco UCS server, providing an ideal platform for deploying a complete virtual network. The virtual network (seen in Figure 4) is entirely built with Linux namespaces and FDio's Vector Packet Processing (VPP) dataplane [21] [22]. VPP was chosen for its ability to fully integrate with Linux namespaces, the kernel, and support of the SRv6 End [23] behavior required for packet processing. The decision to use Linux namespaces was pivotal in creating completely isolated environments for development and testing. The namespaces allowed for having separate interfaces and DNS configurations.

B. eBPF Maps and Data Management

During the implementation of the eBPF program, managing the data from the user space application and the efficiency of queries were critical concerns. To address these concerns, map-in-maps and a structured approach to handling variable-sized data were introduced.

1) Map-in-Map Approach: eBPF allows the creation of so-called map-in-maps, where an outer map has the file descriptor to an inner map. This nesting approach provides a flexible and efficient way of managing complex data structures. An example of the map structure can be found in Figure 3.

2) eBPF Verifier and Variable Length Data: Dealing with variable-length data in eBPF can be challenging due to the verifier's restrictions. The verifier, designed to ensure the safety and stability of eBPF programs, often disapproves of variable-length loops or data structures due to their unpredictable behavior. Because SID lists are usually variable-sized, there has to be a way to circumvent this restriction. The `sidlist_data` struct was introduced, which can be seen in Listing 1. The struct includes a fixed-size array for the SID list and a variable `sidlist_size` that indicates the number of valid entries in the list.

```
struct sidlist_data {
    __u32 sidlist_size;
    struct in6_addr sidlist[MAX_ENTRIES];
};
```

Listing 1. Sidlist Data Struct

C. DNS Interception

Traffic matching (see V-D) is done with the tuple of IPv6 destination address and destination port. For DNS-based operations to be possible, triggering DNS resolution or intercepting the DNS replies is necessary. A distinction is made between whether the client application runs in the controller or standalone mode (see IV-A1).

In controller mode, the client user-space application must trigger the DNS resolution periodically to receive the IPv6 destination address, which is needed to create and send the corresponding queries to the controller (see V-F). The application maintains and updates its DNS cache even after the TTL (Time to Live) expires. That also ensures that if a DNS entry is changed, the new destination IPv6 address is recognized, and the controller can be queried with this new address.

In standalone mode, the eBPF application takes over intercepting DNS replies. The application does not initiate its own DNS requests but checks whether DNS replies arrive on the interface and then extracts the corresponding information required for traffic matching.

When the application starts, all domain names are saved in the `client_lookup_map` with a corresponding `domain_name_id` (see Figure 3). The `domain_name_id` represents a key for a `client_outer_map` entry. The requested domain name is extracted from the DNS reply and compared with the `client_lookup_map`. If a match occurs, the `domain_name_id` is returned, which is then stored in the `client_reverse_map` with the extracted IPv6 destination address. That enables traffic matching (see V-D) to only access the `client_reverse_map` to obtain the corresponding information for encapsulation.

D. Traffic Matching

Traffic matching is a crucial feature in the application, as it has an application-centric approach. This process involves matching traffic based on tuples of IPv6 destination addresses and destination ports in conjunction with DNS resolutions and queries on various eBPF maps (see V-C). The implementation utilizes several eBPF maps, each serving a distinct purpose. These maps include `client_inner_map`, `client_outer_map`, `client_reverse_map`, and `client_lookup_map` (see Figure 3). The user space application initially populates the `client_inner_map` and `client_outer_map` with the relevant SID lists and domain name IDs. Because these two maps are so-called map-in-maps (see V-B1), they cannot be instantiated directly over eBPF; this must be done over the user space application. When a new TCP or UDP connection is established, the eBPF program extracts the destination IPv6 address and destination port out of the packet to build the needed tuple. It can then retrieve the `domain_name_id` with the destination IPv6 address from the `client_reverse_map`. The `domain_name_id` can then be used to retrieve the correct `client_inner_map` from the `client_outer_map`. The `client_inner_map` consists of the needed `sidlist_data` struct (see Listing 1) as value and the destination port as key. With the destination port, the eBPF application can retrieve the struct and continue encapsulating the data (see V-E).

E. Encap/Decap Operations

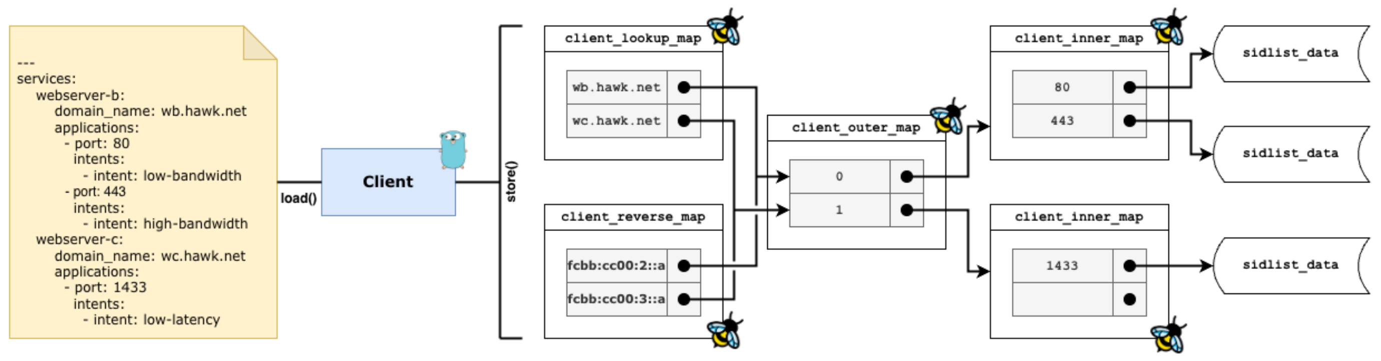
The following section should give an insight into the packet processing of the client and server. For simplicity, the TC and XDP eBPF programs are in the same algorithm snippet (see algorithms 1, and 2) divided into their functions.

1) Client-Side: As mentioned, there is a division between ingress and egress functions, as outlined in Algorithm 1.

- **Ingress processing:** When a packet arrives at the client-side ingress, the function checks if it is an IPv6 packet containing a Segment Routing Header (SRH). If it does, the packet boundaries are validated, and the SRH will be removed. After removing the SRH, the packet will be passed further to the Linux networking stack.
- **Egress processing:** For all packets leaving the client (egress), the function checks if the packet is IPv6. If so, it checks if a SID list is available for the destination IPv6 address and port tuple. The packet is encapsulated with an SRH and the corresponding SID list if a SID list is found. After encapsulating, the packet is sent out of the interface.

2) Server-Side: On the server side, as depicted in algorithm 2, packet processing also involves both ingress and egress functions but with slightly different operations than the client side.

- **Ingress processing:** Similar to the client side, the function checks if incoming packets are encapsulated with an SRH. If so, it first validates the boundaries. After validating it, it extracts and reverses the SID list and stores it under the appropriate tuple. The SRH will be removed, and the packet will be forwarded to the Linux networking stack.
- **Egress processing:** During egress, the server inspects the IPv6 packets to determine if they match any SID lists based on previously stored tuples. If a corresponding SID list is found, it encapsulates the packet with an



```yaml
---
services:
  webserver-b:
    domain_name: wb.hawk.net
    applications:
      - port: 80
        intents:
          - intent: low-bandwidth
      - port: 443
        intents:
          - intent: high-bandwidth
  webserver-c:
    domain_name: wc.hawk.net
    applications:
      - port: 1433
        intents:
          - intent: low-latency
```

Fig. 3. Map Data Structure Example (simplified)

Algorithm 1 Client Packet Processing

```
function INGRESS_PROCESSING(xdp_md)
    if packet is IPv6 and contains SRH then
        Validate SRH boundaries
        Remove SRH from the packet
        Continue with regular packet processing
    else
        Pass the packet as it is
    end if
end function
function EGRESS_PROCESSING(sk_buff)
    if packet is IPv6 then
        Retrieve SID list for dst IPv6 addr and port
        if SID list exists then
            Encapsulate packet with SRH
            Send packet
        else
            Pass the packet as it is
        end if
    end if
end function
```

Algorithm 2 Server Packet Processing

```
function INGRESS_PROCESSING(xdp_md)
    if packet is IPv6 and contains SRH then
        Validate SRH boundaries
        Store reversed SID list with tuple
        Remove SRH from the packet
        Process packet normally
    else
        Pass the packet as it is
    end if
end function
function EGRESS_PROCESSING(sk_buff)
    if packet is IPv6 then
        Check reverse mapping for SID list
        if SID list available then
            Encapsulate packet with corresponding SRH
            Send packet
        else
            Pass the packet as it is
        end if
    end if
end function
```

SRH, ensuring it follows the correct segment routing path. In cases where no SID list matches, the packet is sent without SRv6 encapsulation.

F. Message Format and Structure

The messaging between the client and controller is built upon a protocol buffer definition [24] comprising two primary message types: `PathRequest` and `PathResult` (see Listing 2).

The `PathRequest` message is pivotal in this setup. It includes the IPv6 source, destination address, and a list of `Intents`. Each `Intents` in this list specifies the desired characteristics or requirements for the network path. These `Intents` are flexible, allowing for various types and associated values.

- **Intents:** The `Intents` structure within the `PathRequest` is designed to be versatile, accommodating different network requirements. An `Intent` defines its type (high bandwidth, low latency, etc.) and carries a set of values.
- **Values:** Depending on the `Intent_Type`, values can either be numerical (e.g., minimum or maximum value boundaries) or strings (such as service names in service function chaining).

This design allows for the encapsulation of complex routing requirements.

- **Example:** For instance, in the case of a flexible algorithm [25] intent, the associated value would typically be a number specifying the flexible algorithm number. This specificity ensures that the network path is calculated according to the defined criteria.

The controller uses the `PathResult` message to respond to the `PathRequest`. It mirrors the structure of the `PathRequest` message, including the source and destination IPv6 addresses, the intents, and, most importantly, the calculated SID list for the specified intents.

Upon receiving a `PathResult` message, the client application assigns the provided SID list to the corresponding IPv6 destination and port tuple in the eBPF map. This process ensures that the routing of packets is dynamically adjusted as per the latest instructions from the controller (see Algorithm 3).

Algorithm 3 Process Path Result

```
Initialize portsToUpdate list
for each service in configuration do
    if address matches result destination address then
        for each application in service do
            if intents match result intents then
                Add port to portsToUpdate list
            end if
        end for
    end if
end for
Generate SID list data based on intentResponse
for each port in portsToUpdate do
    Get correct inner map
    Update inner map for the specific port
end for
```

G. Bidirectional Streaming

The client controller communication is facilitated using gRPC (Google Remote Procedure Calls) [26]. The connection is built with a bidirectional stream to achieve an event-driven approach. In traditional unidirectional streaming or request-response models, the client would need to periodically poll the controller to check for updates or changes in network paths. This approach can lead to delays and inefficiencies, especially in rapidly changing network environments. By allowing the controller to push updates to the client actively, the system ensures that the network remains optimally configured despite rapid changes and evolving application landscapes.

- **Startup procedure:** At the startup, the client application initiates the communication by sending `PathRequest` messages for various services defined in its configuration. The controller responds with `PathResult` messages, which the client uses to update the corresponding eBPF maps.
- **Event-driven updates:** When the controller detects network changes–such as path alterations due to a flex algorithm adjustment – it can immediately compute a new `PathResult` and stream this update to the client. Upon receiving a new `PathResult`, the client can update its eBPF maps, ensuring traffic is routed according to the latest SID list.

```
enum IntentType {
  INTENT_TYPE_UNSPECIFIED = 0;
  INTENT_TYPE_HIGH_BANDWIDTH = 1;
  INTENT_TYPE_LOW_BANDWIDTH = 2;
  INTENT_TYPE_LOW_LATENCY = 3;
  INTENT_TYPE_LOW_PACKET_LOSS = 4;
  INTENT_TYPE_LOW_JITTER = 5;
  INTENT_TYPE_FLEX_ALGO = 6;
  INTENT_TYPE_SFC = 7;
}

enum ValueType {
  VALUE_TYPE_UNSPECIFIED = 0;
  VALUE_TYPE_MIN_VALUE = 1;
  VALUE_TYPE_MAX_VALUE = 2;
  VALUE_TYPE_SFC = 3;
  VALUE_TYPE_FLEX_ALGO_NR = 4;
}

message Value {
  ValueType type = 1;
  optional int32 number_value = 2;
  optional string string_value = 3;
}

message Intent {
  IntentType type = 1;
  repeated Value values = 2;
}

message PathRequest {
  string ipv6_source_address = 1;
  string ipv6_destination_address = 2;
  repeated Intent intents = 3;
}

message PathResult {
  string ipv6_source_address = 1;
  string ipv6_destination_address = 2;
  repeated Intent intents = 3;
  repeated string ipv6_sid_addresses = 4;
}
```

Listing 2. Protocol Buffer Definition

VI. Traffic Flow Example

To understand how the application works, refer to the following example of a traffic flow. The example is based on the development network (see Figure 4).

A. Assumptions

Some assumptions must be made to have a good overview and make the example easier to understand.

- The traffic should flow from *host-a* (client) to *host-b* (server).
- A web server is running with port `80` on *host-b*.
- The DNS server replies with the IPv6 address `fcbb:cc00:2::a` for the domain name *wb.hawk.net* (host-b).
- The client application is not yet started in *host-a*.
- The server application is started on *host-b*.
- The *high-bandwith* path from *host-a* to *host-b* is over *site-a, vpp1, vpp2, vpp4, vpp6, site-b*.
- For simplicity, the TCP source port for the connection to the webserver is `65500`.

B. Tuple Matching

In this context, the concept of "tuple matching" is an abstraction for a series of interconnected queries within the

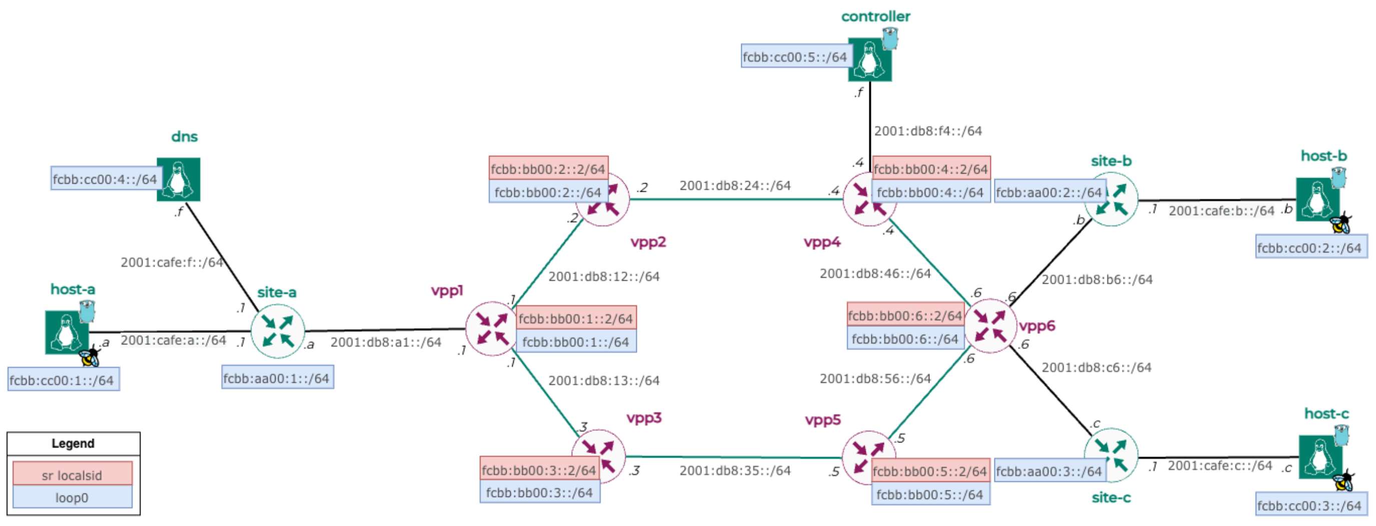


Fig. 4. Development Network with VPP

eBPF maps. When there is a reference to a "matching tuple", there are essentially the following steps involved:

1) The `client_reverse_map` is queried using the destination IPv6 address to retrieve the `domain_name_id`.
2) With this `domain_name_id`, it accesssses the `client_outer_map` to find the corresponding `client_inner_map`.
3) The `client_inner_map` is queried with the destination port to obtain the `sidlist_data` struct.

The result of these queries, particularly the `sidlist_data` struct, determines the appropriate data needed for the encapsulation.

C. Configuration

In Listing 3, the example configuration can be seen, which is given to the client application. The configuration states that there is one service *webserver-b* with the domain name *wb.hawk.net*, which has one application defined with port `80`. The intent for the application is *high-bandwidth*.

```
---
services:
  webserver-b:
    domain_name: wb.hawk.net
    applications:
      - port: 80
        intents:
          - intent: high-bandwidth
```

Listing 3. Example Traffic Flow Configuration

D. Client Start Procedure

When starting the client application on *host-a*, the following happens:

1) The client application reads the configuration and checks if there are intents defined. In this example, an intent *high-bandwidth* is defined for the service *wb.hawk.net* on port *80*.
2) When intents are defined, the application checks if there are IPv6 addresses defined or a domain name. In the example, a domain name is defined. The application stores the domain name *wb.hawk.net* into the `client_lookup_map` together with an assigned `domain_name_id`.
3) The application resolves the IPv6 address of the domain name. The DNS reply will be intercepted to extract the IPv6 destination address. The `client_reverse_map` will be updated with the IPv6 destination address `fcbb:cc00:2::a` and the corresponding `domain_name_id` requested via `client_lookup_map`.
4) An intent request will be created, which consists of the requested intent *high-bandwidth* together with the destination IPv6 address `fcbb:cc00:2::a` and source IPv6 address `fcbb:cc00:1::a`.
5) When the application receives the intent result from the controller, it will store the SID list in the `sidlist_data` struct, which is received via `client_reverse_map`, `client_outer_map`, and `client_inner_map`.

E. Controller Request and Reply

Because the controller was not part of this research, a static mock controller was used to simulate the behavior. The principles for the "correct" controller would be the following:

1) When receiving an intent request, the path from source to destination based on the internal metrics of the network would be calculated.
2) The corresponding SID list would be replied to the client. In this example, the SID list would be:

```
[0]  fcbb:bb00:6::2
[1]  fcbb:bb00:4::2
[2]  fcbb:bb00:2::2
[3]  fcbb:bb00:1::2
```

F. Client Outgoing Traffic Processing

As soon as a process on the client wants to connect to the web server on *host-b*, the eBPF egress application on the client takes over:

1) It checks if a matching tuple `(fcbb:cc00:2::a, 80)` is available in the eBPF map.
2) Since there is a matching tuple, the eBPF application takes the corresponding SID list of the tuple and encapsulates the packet with an SRH, including the SID list. The SID list will now contain the destination IPv6 address, resulting in:
```
[0]  fcbb:cc00:2::a
[1]  fcbb:bb00:6::2
[2]  fcbb:bb00:4::2
[3]  fcbb:bb00:2::2
[4]  fcbb:bb00:1::2
```
3) After encapsulating, the packet is sent out of the egress interface and takes the correct path with the help of SRv6.

G. Server Incoming Traffic Processing

On the server side, the following happens when traffic is incoming on the interface:

1) The application checks if the packet is encapsulated with SRv6.
2) Since it is an SRv6 packet, the application extracts the SID list from the SRH, the source IPv6 address, and the TCP or UDP source port. The source IPv6 address is `fcbb:cc00:1::a`, and the TCP source port is `65500`.
3) The SID list will be reversed while also exchanging the destination IPv6 address with the source IPv6 address, resulting in:
```
[0]  fcbb:cc00:1::a
[1]  fcbb:bb00:1::2
[2]  fcbb:bb00:2::2
[3]  fcbb:bb00:4::2
[4]  fcbb:bb00:6::2
```
4) The reversed SID list is stored with the unique tuple `(fcbb:cc00:1::a, 65500)` in an eBPF map.

H. Server Outgoing Traffic Processing

When there is outgoing traffic on the server, the following happens:

1) The application checks if there is a matching tuple `(fcbb:cc00:1::a, 65500)`
2) Since there is a matching tuple, the application will encapsulate the packet with an SRH and the corresponding reversed SID list.
3) After encapsulating, the packet is sent out of the egress interface.

I. Client Incoming Traffic Processing

When the client gets incoming traffic, there is a conjunction between DNS traffic and other TCP/UDP traffic. For this example, DNS traffic is not interesting, and the focus is on regular TCP/UDP traffic.

1) The application checks if the packet is encapsulated with SRv6.
2) Since it is an SRv6 packet, the application will remove the SRH and pass the packet further to the Linux networking stack.

VII. Results and Discussion

This section critically discusses the outcomes and examines specific issues. This research embarked on an ambitious plan to bridge the gap between application requirements and network routing decisions. The goal was to define application-centric requirements for the traffic to be routed accordingly. By leveraging the dataplane capabilities that eBPF brought and the advanced possibilities of SRv6, an application was developed that achieved the intended objectives. The application can read high-level intents, which then translates to operational objectives. With the intent and the application-based traffic matching, an application-centric framework is born, which can differentiate different application traffic and encapsulate that according to the intent. By introducing a client/server architecture, the application can handle inbound and outbound connections based on the intent and applications. With the implementation done in eBPF for the dataplane functions, the applications have no performance limitations or boundaries. All objectives have been completed; however, some discussions still need to occur, which can be found below.

A. Asynchronous Paths

Modern networks are often symmetric, meaning the path from source to destination is the same as in the opposite direction. That is usually perfectly fine; many applications need such a synchronous path. However, during the research, it became clear that reversing the SID list and, therefore, having a synchronous path is only sometimes the best option. The network conditions and requirements for the return path may differ significantly from the outbound path. For example, the low-latency return path may differ from the outbound low-latency path. So, having the opportunity to have asynchronous paths would not only allow even more application-centric traffic management but would also enable "real" intent-driven communication in both directions. Allowing the controller to calculate the intended path from the client to the server and vice versa requires the client to handle these two SID lists. The first SID list would be handled as described in V. The second SID list for the way back from the server to the client has to be handled separately.

There would be two options for this new challenge; the first would be to allow the server to communicate with the controller directly. That can lead to delay and latency because the server only knows what to request when an SRv6-encapsulated packet arrives on the ingress. That means the server would have to request the controller as soon as a packet arrives on the ingress, which can lead to problems when the server wants to reply to the client's request without having a response from the controller. That, in turn, will lead to either letting the packet pass without any encapsulation back, which is unacceptable or waiting for the controller's reply, which would severely impact the performance.

The second and better option would be to transmit the SID list as metadata within the client-server packet. When a packet arrives at the ingress, the server can prepare its maps and will be ready when the server replies to the client's request.

B. Delivering Metadata

The encapsulation must be adapted to deliver more information to the server to solve the problem described in VII-A. Two methods can be utilized: SRH TLVs (Type-Length-Value) [19] and the IPv6 DOH (Destination Options Header) [27]. Both methods offer distinct ways to transmit "metadata" to the server, such as the needed SID list for the return path. The SRH TLV is an extension of the SRH. It allows the inclusion of additional metadata within the SRH. The TLV format is flexible and can carry various types of data. SRH TLVs were originally designed to provide metadata for segment processing [19]. On the other hand, there is the IPv6 DOH, which is used to carry additional information. The IPv6 DOH is only examined at the packet's destination [27].

There are pros and cons for both of these options when comparing them. The TLV approach is more integrative in having all the segment routing specifics in one header. The problem with this approach is that every node in the path must handle the SRH TLV correctly. That is in contrast to the IPv6 DOH, where the header will be examined only at the destination node. When looking at other projects, like the Path Tracing project, where they switched from the TLV approach [28] to the DOH approach [29], using the DOH would be the better option for future improvement.

C. Enabling C-SID

While the traditional SID list is already defined as RFC in [19], improvements are ongoing to evolve it. The so-called C-SID [30] defines a new approach to compressing the SID list. This approach will be the de-facto standard in future deployments in the backbones of big players like Swisscom, Bell Canada, and others [31]. That leads to the need to bring the functionality of C-SID into the presented application. However, while C-SID improves performance [32], fewer bits are used for defining the compressed SID list; there are new challenges. Simplified, the compression algorithm (described in [30]) will shift bits away while being processed by a Segment Routing enabled router. That leads to missing the complete path information at the destination node, breaking the approach of reversing the SID list described in V-E.

The compressed SID list must be delivered as metadata in the packet to add this functionality to the application. That leads back to the approach described in VII-B, which could be used for that. When implementing the C-SID functionality, there would be no way around implementing the approach defined in VII-B, except there would be another compression algorithm that would not shift bits away.

VIII. Conclusion and Future Work

This section should reflect on the journey to enhance networking with eBPF and SRv6. The research demonstrated the potential of these technologies to create a more responsive and flexible network infrastructure tailored to the needs of the applications and users. SRv6's source-based routing approach enables intent-driven and application-centric networking, along with the powerful possibilities of eBPF to create dataplane functionality on Linux. The development of new technologies and the changing requirements of users are also forcing the network to evolve continuously. This work has its share in this development but has to be further developed.

A. Future Research Directions

While this research provides a good foundation for bringing end-to-end SRv6 communication in combination with an application-centric and intent-driven traffic-shaping approach to the end hosts, there are several promising avenues for future research.

- **Central controller:** Developing the central controller (as described in IV-B) is crucial to research further. The controller should always provide the clients with an up-to-date path for the specific application intents.
- **IoT integration:** Implementing SRv6 and eBPF on IoT devices can be beneficial to control traffic flow. However, further research is needed on integrating eBPF on IoT devices.
- **Integration with emerging technologies:** The integration of the SRv6 with emerging technologies is already ongoing [33]. Future studies could investigate how SRv6 and eBPF, as a combination, can enhance performance and efficiency in rapidly growing technologies like 5G, IoT, and edge computing.
- **Scalability and performance optimization:** Continued research is necessary to give more insights into the performance and scalability improvements that end-to-end SRv6 brings to the network.
- **Return path optimization:** To further improve the intent-driven objective, there is a need to implement the thoughts of the discussion presented in VII-A.
- **Implementation of SRv6 C-SID:** Investigating the implementation of SRv6 C-SID (as described in VII-C) is one of the most significant future research directions. As defined, C-SID will become more prevalent in areas like 5G due to the compression it provides and the benefits it brings [34]. Therefore, implementing it would prepare the application for future deployments.
- **Automated management tools:** Managing the requirements of IoT devices can be overwhelming when done manually. An application that can automate defining the application requirements embedded in IoT devices can benefit the future.
- **Cloud integration:** In today's world, most applications live in the cloud. Testing and improving the presented eBPF applications for cloud architectures is necessary.

Being able to connect, for example, servers hosted in the cloud to local clients would not only improve the performance but also the adaptability.

By pursuing the abovementioned avenues, the journey is to push the boundaries of what is possible in networking, ensuring that future networks are more efficient, capable, and aligned with the changing needs of applications and users.

References

[1] "BPF and XDP Reference Guide," https://docs.cilium.io/en/latest/bpf/#bpf-and-xdp-reference-guide, [Accessed 06-12-2023].

[2] A. S. Jay Schulist, Daniel Borkmann, "Linux socket filtering aka berkeley packet filter (bpf)," https://www.kernel.org/doc/Documentation/networking/filter.txt, [Accessed 06-12-2023].

[3] M. H. N. Mohamed, X. Wang, and B. Ravindran, "Understanding the security of linux ebpf subsystem," in *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 87–92. [Online]. Available: https://doi.org/10.1145/3609510.3609822

[4] Y. Li, W. Niu, Y. Zhu, J. Gong, B. Li, and X. Zhang, "Fuzzing logical bugs in ebpf verifier with bound-violation indicator," in *ICC 2023 - IEEE International Conference on Communications*, 2023, pp. 753–758.

[5] "Buzzer - an ebpf fuzzer toolchaion," https://github.com/google/buzzer.

[6] "Katran: A high performance layer 4 load balancer," https://github.com/facebookincubator/katran, [Accessed 06-12-2023].

[7] "How We Used eBPF to Build Programmable Packet Filtering in Magic Firewall," https://blog.cloudflare.com/programmable-packet-filtering-with-magic-firewall/.

[8] "Cilium: eBPF-based Networking, Observability, and Security," https://cilium.io/, [Accessed 06-12-2023].

[9] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 54–66. [Online]. Available: https://doi.org/10.1145/3281411.3281443

[10] A. Mayer, L. Bracciale, P. Lungaroni, P. Loreti, S. Salsano, and G. Bianchi, "ebpf programming made easy with eclat," in *2022 18th International Conference on Network and Service Management (CNSM)*, 2022, pp. 28–36.

[11] H. Liu, "tc/BPF and XDP/BPF," https://liuhangbin.netlify.app/post/ebpf-and-xdp/, [Accessed 06-12-2023].

[12] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Comput. Surv.*, vol. 53, no. 1, feb 2020. [Online]. Available: https://doi.org/10.1145/3371038

[13] "eBPF introduction," https://ebpf.io/.

[14] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv, "Simple and precise static analysis of untrusted linux kernel extensions," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 1069–1084. [Online]. Available: https://doi.org/10.1145/3314221.3314590

[15] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network services with ebpf: Experience and lessons learned," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, 2018, pp. 1–8.

[16] L. Pang, C. Yang, D. Chen, Y. Song, and M. Guizani, "A survey on intent-driven networks," *IEEE Access*, vol. 8, pp. 22 862–22 873, 2020.

[17] J. Zhang, J. Guo, C. Yang, X. Mi, L. Jiao, X. Zhu, L. Cao, and R. Li, "A conflict resolution scheme in intent-driven network," in *2021 IEEE/CIC International Conference on Communications in China (ICCC)*, 2021, pp. 23–28.

[18] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois, "The segment routing architecture," in *2015 IEEE Global Communications Conference (GLOBECOM)*, 2015, pp. 1–6.

[19] C. Filsfils, D. Dukes, S. Previdi, J. Leddy, S. Matsushima, and D. Voyer, "IPv6 Segment Routing Header (SRH)," Mar. 2020. [Online]. Available: https://www.rfc-editor.org/info/rfc8754

[20] K. Beevers, "The evolving role of dns in today's internet infrastructure," *Data Center Knowledge*, 2023. [Online]. Available: https://www.datacenterknowledge.com

[21] "FDio - The Universal Dataplane," https://fd.io/, [Accessed 18-12-2023].

[22] "Vpp - vector packet processing," https://github.com/FDio/vpp, [Accessed 22-11-2023].

[23] C. Filsfils, P. Camarillo, J. Leddy, D. Voyer, S. Matsushima, and Z. Li, "Segment Routing over IPv6 (SRv6) Network Programming," RFC 8986, Feb. 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc8986

[24] "Protocol Buffers," https://protobuf.dev/, [Accessed 22-12-2023].

[25] P. Psenak, S. Hegde, C. Filsfils, K. Talaulikar, and A. Gulko, "IGP Flexible Algorithm," RFC 9350, Feb. 2023. [Online]. Available: https://www.rfc-editor.org/info/rfc9350

[26] "gRPC," https://grpc.io/, [Accessed 22-12-2023].

[27] B. Hinden and D. S. E. Deering, "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460, Dec. 1998. [Online]. Available: https://www.rfc-editor.org/info/rfc2460

[28] C. Filsfils, A. Abdelsalam, P. Camarillo, M. Yufit, T. Graf, Y. Su, S. Matsushima, M. Valentine, and A. Dhamija, "Path Tracing in SRv6 networks," Internet Engineering Task Force, Internet-Draft draft-filsfils-spring-path-tracing-04, Aug. 2023, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-filsfils-spring-path-tracing/04/

[29] ——, "Path Tracing in SRv6 networks," Internet Engineering Task Force, Internet-Draft draft-filsfils-ippm-path-tracing-00, Dec. 2023, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-filsfils-ippm-path-tracing/00/

[30] W. Cheng, C. Filsfils, Z. Li, B. Decraene, and F. Clad, "Compressed SRv6 Segment List Encoding," Internet Engineering Task Force, Internet-Draft draft-ietf-spring-srv6-srh-compression-10, Dec. 2023, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-spring-srv6-srh-compression/10/

[31] J. Horn, "Introduction to srv6 usid," Presentation at Cisco Live, Swisscom Schweiz AG, Amsterdam, Jun. 2023. [Online]. Available: https://docplayer.net/235902219-Introduction-to-srv6-usid.html

[32] A. Tulumello, A. Mayer, M. Bonola, P. Lungaroni, C. Scarpitta, S. Salsano, A. Abdelsalam, P. Camarillo, D. Dukes, F. Clad, and C. Filsfils, "Micro sids: A solution for efficient representation of segment ids in srv6 networks," *IEEE Transactions on Network and Service Management*, vol. 20, no. 1, pp. 774–786, 2023.

[33] T. Torii, "Srv6 deployment at softbank," SoftBank Corp, Jun. 2023. [Online]. Available: https://www.segment-routing.net/conferences/2023-01-25-srv6-deployment-at-softbank/

[34] C. Li, J. Mao, S. Peng, Y. Xia, Z. Hu, and Z. Li, "Application-aware g-srv6 network enabling 5g services," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2021, pp. 1–2.