Busoni: Policy Composition and Northbound Interface for IPv6 Segment Routing Networks

Osamah L. Barakat*, Pier Luigi Ventre†, Stefano Salsano†, Xiaoming Fu*
*University of Goettingen, Germany, †University of Rome Tor Vergata, Italy

***Abstract*—Segment Routing is a source routing based architecture that provides an opportunity to include a list of instructions called *segments* in the packet headers. The segments may allow the inclusion of detours for responding to Traffic Engineering needs or Service Function Chaining implementations. Even though there is an increasing interest towards enhancing and adopting Segment Routing, the administrators are still burdened with the task of manually write and maintain the segment lists. Such type of management presents several challenges ranging from error-prone configurations to increased response time for network updates. In this paper, we present a Segment Routing management framework named *Busoni*, which automates and simplifies the process of segments lists management. Additionally, we also provide programming tools to compose and manage Segment Routing policies that operate efficiently, even under multi-tenancy environments. Using different use cases, we show the programming capabilities offered by our framework.**

***Index Terms*—Network Function Virtualization, Northbound Interface, Policy, Segment Routing, SFC, SRv6, Software Defined Networking, VNF**

I. Introduction

Segment Routing (SR) [1] was proposed the first time in the late 2014 mainly with the goal of overcoming some scalability issues and limitations that had been identified in the traffic engineered Multi Protocol Label Switching (MPLS-TE) solutions used for IP backbones. SR is a variation of source routing where instructions, commonly known as *segments*, are attached to packet headers to implement detours to the default shortest path. The survey in [2] provides a thorough description of SR technology. SR could be implemented on top of either MPLS or IPv6 (i.e., SR-MPLS and SRv6) dataplane, where segments identifiers (SIDs) are respectively encoded as MPLS labels and IPv6 addresses.

The SRv6 implementation has drawn a lot of attention to researchers from academia and industry, as witnessed recently by the publication of several research activities [3], [4] and [5] (a comprehensive survey related to Segment Routing can be found at [2]). For these reasons, we are focusing our attention on the SRv6 dataplane. Moreover, the IETF draft [6] introduced the concept of encoding network commands (i.e., SRv6 behavior) as IPv6 addresses in the segments list. This means when a network node receives a packet encapsulated with the SR header, and the destination address matches an associated behavior provided by this node, it will execute this defined action. Therefore, whenever a network administrator wants to implement a network program (e.g., traffic engineering), she/he needs to inject a segments list that represents her/his program in the packet's header.

Even with all of these programming capabilities enabled by SRv6, network administrators still face the difficulty of manually constructing segments lists that fulfill their intents and policies. To the best of our knowledge, there is only one proposal that partially automates segments list management [5], where authors proposed to utilize the DNS service in the enterprise network to transfer segments lists between end users and the controller. However, the proposal does not react to networks updates and overrides the forwarding leveraging DNS service instead of using the service IP address, which would make it not applicable in several real contexts.

In other related SRv6 works [4], [7]–[10], segments lists were composed manually as topologies used in the evaluation tended to be small. However, in real operated network topologies, manual composition presents various challenges in the context of composing network policies. One of the many challenges is the errors prone manual segments list composition. It becomes even more challenging when the identifiers are changed due to a migration of some network functions or any other network dynamic events, and the need to respond to such updates is time sensitive. Another challenge with manual management is finding a correct parameter to pass in the SRv6 command. Moreover, possible conflicts between SRv6 behaviors could exist due to behavior misuse such as applying decapsulation followed by encapsulation. All of these challenges motivate the need for an automated framework that efficiently manages SR policies. Existing solutions [11], [12] that address these challenges in SDN work only with OpenFlow based networks.

In this paper, we present *Busoni*, a framework to compose and manage network policies on top of SRv6 networks. Busoni provides the needed programming functionalities for network administrators as a northbound interface on top of an SR controller. The contributions of this work are:

- **Automate segments list management:** Busoni exploits the benefits of the network controllers and utilizes a data store to keep track of the SIDs announced in the network. This feature allows the administrators to focus on network management goals rather than focusing on physical details of segments and their location.
- **SR policy management:** Network administrators can choose to utilize the predefined policies provided by Busoni or build a new policy on top of it. Busoni

offers different tools to compose SR policies ranging from packet matching rules to functions that attach SRv6 behaviors to segments lists.
- **Responding to network dynamics:** Busoni updates any affected policy whenever there is a network failure or updates in the network topology. This feature complements SR built-in reaction functions and keeps the installed policies resilience to dynamic events.
- **Multi-tenancy support:** Carrier grade networks or cloud service providers deliver overlay services to end-users. Busoni supports multi-tenancy as it embeds tenants IDs in any related SRv6 commands.

The rest of the paper is organized as follows: Section II presents the architecture of the framework. Then, we describe use cases in Section III and draw conclusions in Section IV.

II. Northbound Interface for SRv6

Busoni is a northbound interface for SRv6 networks. A typical network environment, where Busoni operates, is similar to the SDN [13] reference architecture, which includes network controllers, interfaces to communicate with end-users (i.e., northbound) and data plane (i.e., southbound). It is crucial to expose network topology information to Busoni to run network management-related tasks (e.g., finding paths).

The networks controller should be able to collect the topological information from the SRv6 data plane using available southbound interfaces (e.g., OSPFv3 or BGP-LS). The controller also has to push the commands or segments lists to the edge routers leveraging the available protocols [4] (e.g., gRPC or SSH). Using a specific protocol (e.g., OSPFv3) to run one of the corresponding tasks does not affect how Busoni processes and installs policies. Busoni interacts with the network controller without knowing how the controller learns the network topology. As shown in the Figure 1, there are three main subsystems in Busoni, *SRtypes*, *Path Computation*, and *Database Middleware*. They interact with each other to provide the full-fledged functionality of Busoni.

The first component is *SRtypes*, which is the entry point to the framework. It contains the main policy class which users would use to build their policies. It also includes the basic types needed by Busoni to hold policy's components (e.g., SRv6 Behaviors) including the class `Match`, which is used to define matching criteria for incoming packets. *SRtypes* interacts with the other two subsystems to perform its functions. It uses *Path Computation* to calculate a possible shortest path according to policy's conditions and communicates with *Database Middleware* when it needs to save or retrieve any policy information to/from the datastore.

Path computation first checks what type of path is requested (e.g., SFC) and calls the appropriate functions accordingly. Whenever it needs to save the computed path to the datastore, it communicates with *Database Middleware*. Busoni allows four different variations of path finding queries: simple path, *QoS only*, *SFC only*, and path with both QoS and SFC requirements. In the simple path case, Busoni would find the shortest path between all ingress and egress points, and it could give the same path as the IGP routing protocol. The second path finding query would evaluate all possible paths which fulfill the QoS requirements and retrieve the path with a minimum cost. However, in the third case where there are some network functions to be included in the path, finding the shortest path would be an NP-Hard problem [14]. Therefore, Busoni uses an heuristic algorithm proposed in [15], the ASR algorithm, where the final path is composed of shortest paths between waypoints (e.g., routers hosting virtual network functions). In the latter path query type, where QoS and SFC should be considered while finding the path between ingress and egress points, Busoni combines both queries described in the second and third cases.

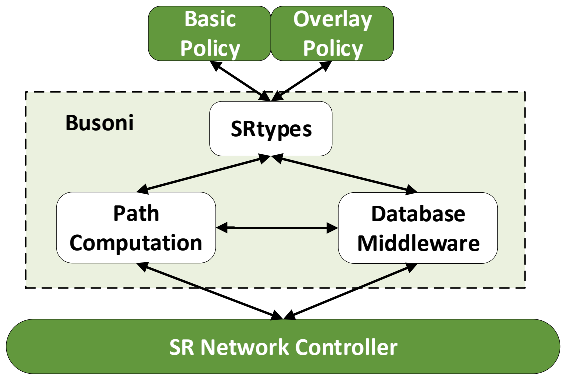

Figure 1: Major software subsystems of Busoni

Finally, the *Database Middleware* holds all functions and event handling methods that need to interact with the datastore directly. It operates as an interface to ease database interaction with other components. It also listens to any dynamic event and responds by calling the corresponding handler, and whenever it needs to update a policy, it communicates with *SRtypes* to launch finding path function for the update procedure.

A. API for policies composition

Busoni provides a standard policy class `SRpolicy` which contains the necessary functions needed to compile submitted policies. End-users inherit this class and mainly extend the `eval` function which is called automatically when users instantiate an object from their defined class. Before executing the `eval` function, Busoni needs to first find a corresponding path that represents the requested policy (e.g., adhere to QoS or go through network functions). Then, it will have to encode the path using available SIDs.

In Listing 1, we see that `PolicySR` abstraction takes many arguments that help in defining customized policies. The first argument *match* defines the matching criteria for incoming packets classification. For example, both source and destination could be determined using a range of IP addresses. `Match` class is flexible and provides through defined keywords (e.g., dstIP, dstport) a powerful tool to define custom matching rules like exclude a specific IP address. Users would use native python libraries to define their criteria without a need to learn any specific syntax. The second argument holds the quality of service (QoS) specifications defined by the user's policy. It

Listing 1: The construction function of class `PolicySR`

```
PolicySR(match, qos=None, nfList=None, isOrdered=True,
    matchOnSrc = False, id=0)
```

could relate to any QoS metrics (e.g., bandwidth or latency), where the user has the flexibility to define what is needed in the policy. In the implementation of Busoni, we used a score metric where high score value reflects low latency and vice versa. Any other metrics or a combination of them would be used with some query updates to maintain which condition is preferred, low or high value. It is also an optional argument, and thereby it is not mandatory that policy provides some QoS specifications.

After that, there are some arguments related to service function chaining and whether they should be visited in order or not. An NFV management framework should provide beforehand which network functions are running in the network and feed the network controller with functions related information such as functions name and SIDs. Data-plane routers on their side, using the IGP protocol, broadcast network functions they host (virtualized or stand-alone). The last arguments determine if routers should match incoming packets against the source IP address or not, tenant ID if any (VPN user) for special routing table matching when packets exit the network domain, and a policy id which is used internally for dynamic updates.

Users, after passing the above inputs, need to declare any special handling needed for their packets. To understand what "*special*" means in this context, we elaborate in Section II-B how the basic class compiles the policy requirements to generate related segment commands, including the *special* handling requests.

B. Busoni in action

To generate a correct list of segments that represents what an user wants from the network, Busoni has to perform some functions. These functions begin with finding a path then encode it as segments list (i.e., SIDs) and conclude with performing special routines defined by the user (e.g., adding some SRv6 behaviors). Busoni starts working just when the basic class `PolicySR` is inherited, and its constructor function is called. It first sends a request to the *Path Computation* subsystem. If there are no routes satisfying the policy's requirements, Busoni would inform the user. In this case, the user has the freedom either to adjust policy's requirements or propose changes to the network topology. Busoni cannot accept all submitted policies as it considers available resources.

The second step after finding the path is to encode it using segments. In this step, Busoni calculates the minimum number of segments needed to represent the whole path. Even though most of proposals [16]–[19] address the problem for MPLS-SR, minseg algorithm [19] considers adjacency segment [1] and SRv6, thereby fits with our framework. The main idea of the algorithm is to check if the shortest path between two nodes belonging to the current path, will traverse any middle points in the same path. The minseg algorithm was modified to fit in our network model by ignoring network functions' relationships and use only routers' to calculate the correct shortest path between routers.

After that, when the segment list is available, Busoni calls the `eval` function which contains any custom actions that should be executed before sending the segment list to ingress routers. For example, end users could choose a behavior, from available SRv6 behaviors, which are supported by data plane routers, to be added and concatenated to the segments list. A practical example we can mention here is the multi-tenancy management where users would attach `T.Encaps` and `END.DT6` at the beginning and at end of the segments list respectively to get VPN service. Such addition to the segments list would trigger a flag to let *Database Middleware* subsystem to generate proper southbound interface commands and ensure that policy is executed correctly in the data plane.

In the last step, Busoni will store the policy information in the database. This information contains matching criteria composed by the user, the calculated path, the optimized segment list, any path requirements specified by the user (either QoS or SFC related), and the policy ID. Maintaining this information is essential to allow any future updates that could be triggered as a response from network dynamics.

C. Datastore

To keep data integrity, track installed policies, and respond to network events, Busoni uses as data store a graph database [20] to model and save network environment data [21]. This option allows Busoni to recover from an outage and reload the database and synchronize any network updates. In this database model, nodes represent main network components either physical (e.g., routers) or virtual (e.g., policies), and edges represent the connection between the nodes.

Whenever a path is calculated between two points, it is stored to be used later as one of the policy information. Moreover, when Busoni saves a policy in the database, it makes sure that all network functions and routers that are part of the policy path are connected to the policy node. Thereby, if any event regarding these components is raised, Busoni would find it easy to update the policy according to the raised event.

III. Use cases

In this section, we show the functionality of our framework, Busoni, using three realistic use cases: SFC, QoS and VNF migration. All scenarios are based on a simple network topology depicted in Figure 2. In the topology, there are two types of network functions firewall (FW) and deep packet inspection (DPI), each one hosted on a separate router. There are also two network sites: site A and site B. We assign bandwidth values for the links in mb/s, which will be used later to demonstrate a QoS based policy.

A. Basic policy with SFC

Basic Policy is any policy that does not have any multi-tenancy requirements and where a network admin would like to steer the traffic between site A and B through two network

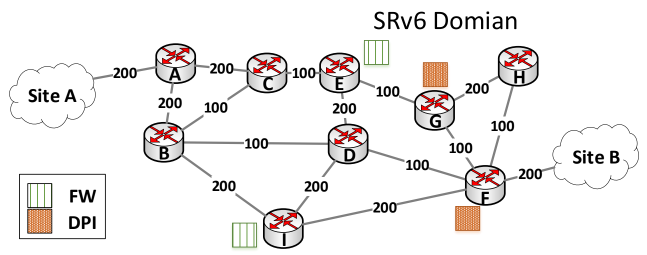

Figure 2: SRv6 enabled domain

functions (e.g., firewall and deep packet inspection). The starting point in Busoni is to define a class holding the policy description. After the class definition, the network admin should start by defining the match object to specify the source and destination addresses and then she/he would instantiate an object from the class and pass the network function names (FW, DPI). Busoni will choose the hosting router that makes the total route *better* according to what we discussed earlier about path computation in Section II. There is no need to call or define further functions as the path gets calculated and installed automatically after instantiating an object from the policy class. Considering the scenario in Figure 2, the segment list will be $\texttt{FW}_{\texttt{E}}$`,`$\texttt{DPI}_{\texttt{G}}$`,Site B`.

B. *Overlay with QoS policy*

Let us now consider a network admin that would like to implement a quality of service policy where packets between the two sites must use only links with a minimum bandwidth of 200 Mb/s. In this use case, we assume also that the traffic belongs to one tenant. Therefore, the SRv6 behaviors `T.encaps` and `End.DT6` are needed at ingress and egress nodes, respectively. Setting up the decapsulation and the tenant routing table at the egress points should be done during the VPN installation phase. Following the same steps we described earlier, the network admin would first define the policy class (Overlay Policy) Then, the network admin will use the `eval` function to define `T.encaps` and `End.DT6`. The former will be inserted in the top of the segments list and the latter, with the tenant ID as a parameter, will be attached at the end of this list. The path in this case according to the input topology would be `A,B,I,F,Site B` and therefore the segments are `T.encaps,I,End.DT6(102)`$_{\texttt{F}}$.
`A,B,D,`$\texttt{FW}_{\texttt{D}}$`,F,`$\texttt{DPI}_{\texttt{F}}$`,Site B` and hence the segments list that represents the updated path is $\texttt{FW}_{\texttt{D}}$`,`$\texttt{DPI}_{\texttt{F}}$`, Site B`.

C. *Responding to a VNF migration*

Let's consider the use case described in Section III-A as starting point, the network function FW hosted in router E is migrated to router D. The controller is notified of this event and hence will trigger Busoni to respond. First, Busoni will have to find affected policies and retrieve their match conditions and their information. At this point, Busoni will discover that there is an already installed policy using the function that was moved from router E. Then, it deletes the policy from the database and starts the update procedure immediately by calculating a new path that satisfies SFC and the classification conditions. The path would be

IV. Conclusion

In this paper, we presented a policy composer and management framework for SRv6 networks. We have showcased the capabilities of the framework and the tools provided using three realistic use cases: SFC, QoS and VNF migration.

References

[1] Filsfils *et al.*, "The Segment Routing Architecture," in *2015 IEEE Global Communications Conference (GLOBECOM)*, Dec. 2015, pp. 1–6.
[2] Ventre *et al.*, "Segment routing: A comprehensive survey of research activities, standardization efforts and implementation results," *arXiv preprint arXiv:1904.03471*, 2019.
[3] Abdelsalam *et al.*, "Performance of IPv6 Segment Routing in Linux Kernel," in *1st Workshop on Segment Routing and Service Function Chaining (SR+SFC 2018) at CNSM 2018, Rome, Italy*, 2018.
[4] Ventre *et al.*, "SDN Architecture and Southbound APIs for IPv6 Segment Routing Enabled Wide Area Networks," *IEEE Trans. Netw. Service Manag.*, vol. 15, no. 4, pp. 1378–1392, Dec. 2018.
[5] Lebrun *et al.*, "Software Resolved Networks: Rethinking Enterprise Networks with IPv6 Segment Routing," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. ACM, 2018, pp. 6:1–6:14.
[6] Filsfils *et al.*, "SRv6 Network Programming," IETF Secretariat, Internet-Draft draft-filsfils-spring-srv6-network-programming-06, Oct. 2018.
[7] Abdelsalam *et al.*, "Implementation of virtual network function chaining through segment routing in a linux-based NFV infrastructure," in *2017 IEEE Conference on Network Softwarization, NetSoft*, 2017, pp. 1–5.
[8] Desmouceaux *et al.*, "6lb: Scalable and Application-Aware Load Balancing with Segment Routing," *IEEE/ACM Trans. Netw.*, vol. 26, no. 2, pp. 819–834, Apr. 2018.
[9] Aubry *et al.*, "Robustly Disjoint Paths with Segment Routing," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18, 2018, pp. 204–216.
[10] Xhonneux *et al.*, "Leveraging eBPF for Programmable Network Functions with IPv6 Segment Routing," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18, 2018, pp. 67–72.
[11] Reich *et al.*, "Modular SDN Programming with Pyretic," *USENIX*, vol. 38, pp. 40–47, 2013.
[12] Foster *et al.*, "Frenetic: A network programming language," *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, pp. 279–291, 2011.
[13] Kreutz *et al.*, "Software-Defined Networking: A Comprehensive Survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.
[14] Deo *et al.*, "Shortest-path algorithms: Taxonomy and annotation," *Networks*, vol. 14, no. 2, pp. 275–323, Jun. 1984.
[15] Dwaraki *et al.*, "Adaptive Service-Chain Routing for Virtual Network Functions in Software-Defined Networks," in *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, ser. HotMIddlebox '16, 2016, pp. 32–37.
[16] Lazzeri *et al.*, "Efficient label encoding in segment-routing enabled optical networks," in *2015 International Conference on Optical Network Design and Modeling (ONDM)*, May 2015, pp. 34–38.
[17] Davoli *et al.*, "Traffic Engineering with Segment Routing: SDN-Based Architectural Design and Open Source Implementation," in *2015 Fourth European Workshop on Software Defined Networks*, Sep. 2015, pp. 111–112.
[18] Cianfrani *et al.*, "Translating Traffic Engineering outcome into Segment Routing paths: The Encoding problem," in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Apr. 2016, pp. 245–250.
[19] Aubry *et al.*, "SCMon: Leveraging segment routing to improve network monitoring," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, Apr. 2016, pp. 1–9.
[20] Angles *et al.*, "Survey of Graph Database Models," *ACM Computer Survey*, vol. 40, no. 1, pp. 1–39, Feb. 2008.
[21] Barakat *et al.*, "Gavel: A fast and easy-to-use plain data representation for software-defined networks," *IEEE Transactions on Network and Service Management*, pp. 1–12, 2019.