Dynamic Service Programming with Path Preprocessing

Julian Klaiber*, Severin Dellsperger*, Laurent Metzger*, Ahmed Abdelsalam†, Francois Clad†
*Institute for Networked Solutions, Eastern Switzerland University of Applied Sciences
{julian.klaiber, severin.dellsperger, laurent.metzger}@ost.ch
†Cisco Systems
{ahabdels, fclad}@cisco.com

***Abstract*—Network services are essential in modern networks. They are a crucial part of today's network operation and ensure customizable, reliable, and secure communication. This being said, they also have a noteworthy drawback: they are consumed in a static manner. As a consequence, the management of network services is too complex and thus expensive and error-prone. Furthermore, this static processing cannot react to network changes: a service outage, a link failure, or any other network events can result in connectivity loss for customers and has to be usually resolved manually. This paper proposes a practical solution for dynamic, event-triggered, and fast path calculations that program services and permit a so-called service chain. To allow programming services directly in the backbone of a service provider network is solving the static service consumption in service provider networks. The solution is based on the functionalities of Segment Routing over IPv6 (SRv6 in short), which implements the source-based routing paradigm with the native IPv6 encapsulation. Our focus was set on finding the best reliable and fast way to calculate the best path through the given service instances. Because service provider networks are constantly growing, an algorithm has to be found and implemented so that the growth of the network has no impact on the calculation performance. A complete cloud-native development approach has been taken to create the application as elastic and fault-tolerant as possible. Thanks to the cloud-native approach, the application can be portable in a different environment, in an on-premise datacenter, or directly in the public cloud.**

***Index Terms*—Segment Routing, SRv6, Service Programming, Scalability, Performance**

I. Introduction

In the recent years, the IT network domain has changed fundamentally. New approaches and technologies has been introduced, which has open the door to innovations. It paves the way for the development of modern and dynamic networks that close the gap between networks, applications, and end-users. It permits creating applications that work closely with the underlying network. Creating a network that really fulfils customer needs instead of simply providing basic connectivity. That's where network services like firewall systems, intrusion detection/prevention systems or load-balancers come into play. They have become indispensable and are firmly anchored in computer networks but they are consumed in a static manner. Service Programming is one of that innovation in future networks and addresses the challenges of static service consumption. The network is dynamically configured based on the intent of the operator and the network services are included into the network path and process customer traffic according to their needs. The network services can be placed anywhere in the network - the service programming application will find the best services according to the constraint applied by the operator. Hence, networks with integrated service programming become more intelligent, flexible and cost-efficient by allowing easily the consumption of the best services by every application.

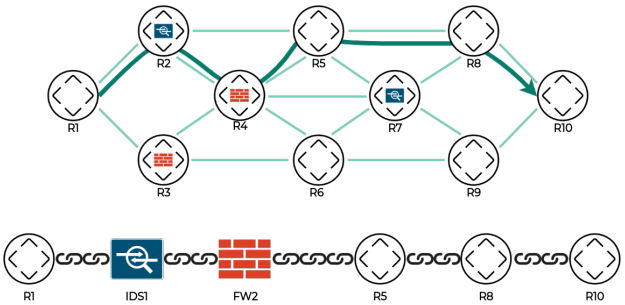


Fig. 1. Service Chain Example

Such a service programming application should produce a so-called service chain (see fig. 1) through which the traffic should flow. The application should steer the traffic into a network path that follows the computed service chain. The application should track the changes in the network and adapt the service chain to topology changes automatically (if necessary). That is the only way to ensure that traffic always takes the most suitable route via the correct services. [1]

The term Service Programming was first mentioned in [2]. Service Programming is a solution to enforce traffic to be steered and processed through network services on the path from source to destination. Because Service Programming is more intelligent, dynamic, and therefore more enhanced than simple service chaining, it can be seen as its successor. Service Programming aims to introduce a technique to program network paths that incorporates network services like a modern programming language. Therefore, this very dynamic approach addresses the customer needs of the future and aims to deliver a method that can adapt to network events. [3] [2] [4]

978-3-903176-36-2 ©2021 IFIP

A. Objective

This paper propose a Segment Routing Service Programming application which solves the challenges with static service consumption. The application enforces traffic engineering in the network and ensure that traffic flows over the programmed path that has been calculated based on the constraints defined by a user in the GUI of the application. The packets should not only be steered on the best path from source to destination but also sent via the most suitable services in the network. The appropriate services should process the packets and execute the necessary actions according to their configuration. In order to calculate and program the policies created by the application, the application obtains the network topology and Segment Routing information via BGP-LS [5].

So that packets can be processed by services, SR-aware services are in use and those services are assigned with appropriate Segment information. More information about the SR-aware services can be found in section I-C.

A user of the Segment Routing Service Programming application is able to create SR service policies directly in the GUI of the application. The application provides all necessary information for the user to define the path and the path objectives. The user selects the following information in order to create a valid policy:

- SR Policy headend information (Node, VRF, Network)
- Endpoint information (Node, VRF, Network)
- Optimization metric / SR-Algorithm information [6] [7]
- Service functions (number of service functions, order of the service function)

The application calculates the best path and allows the user to deploy the created SR Policy in the network.

In order to steer the traffic onto the calculated path, BGP-based steering is used [8]. The application has to guarantee that the following tasks are included, so that the mechanics works:

1) The application has to ensure that the appropriate routes are tagged with a color value and that this information is distributed in the Segment Routing domain. The application has to change configuration aspects on the endpoints to address that matter.
2) The application has to ensure that the path is programmed and enforced. Therefore, the application creates Segment Routing Policies with an explicit candidate path and the appropriate color information. This way the traffic can be directed to the services with their Segment information.

B. Service Programming and Segment Routing

Segment Routing can combine service programming and other use-cases such as underlay and overlay using single encapsulation. Other solutions require new encapsulations and protocols for each use case. The single encapsulation leads to a very efficient and elegant solution in terms of SR service programming. Since all the instructions are carried in the packet header, SR provides a simple solution with no per-flow state in the intermediate routers. The whole approach is

Fig. 2. Calculation Sub-Paths

based on the Network Programming concept defined in RFC 8986 [9], which allows to specify the packet processing in the Segment List in the IPv6 header [10]. In an SR network, each service function is associated with a segment and can thus be included as part of an SR path and benefit from the powerful traffic engineering concepts of Segment Routing [8]. In a nutshell, the following criteria must be available for a successful Service programming :

- The packet must be steered to the service. This functionality is already present.
- The service must understand the segment and know which instruction has to be executed. More details about this current state can be found in section I-C.

C. SR-aware Services

In order to integrate a service into a Segment Routing path, the service has to be associated with a segments. [11] The exact functionality and implementation go beyond this paper. However, it is worth mentioning that SR Service programming differentiates between two categories of services: SR-aware and SR-unaware. SR-unaware services cannot handle the different functionalities of SRv6 natively; for example, a proxy has to be used so that the traffic can be routed through this service. Therefore, our focus is on SR-aware services that fully allow the utilization of the SRv6 network programming capabilities. In order to implement an end-to-end service programming, the following existing SR-aware services have been deployed in the proof of concept:

- **SERA** stands for SEgment Routing aware firewall and extends the Linux iptables functionalities to handle Segment Routing packets. [12]
- **SR-aware Snort** is an extended version of the well-known Intrusion Detection System (IDS)/Intrusion Protection System (IPS) snort. [13]

II. Calculation

The calculation of the most suitable paths is a key part of the application. It should not only deliver a correct result, but it should also do it within an acceptable time frame. The difficulty of the calculation lays in the structure of a complete path. A complete path contains several sub-paths depending on the number of service instances in the path. So the calculation has to analyse all sub-paths to decide if the whole path is the most suitable one. If, for example, the path has only one service instance in it, it has to consider two sub-paths: The first path goes from the start node to the service. The second sub-path runs from the service node to the destination node. A similar example is illustrated in fig. 2.

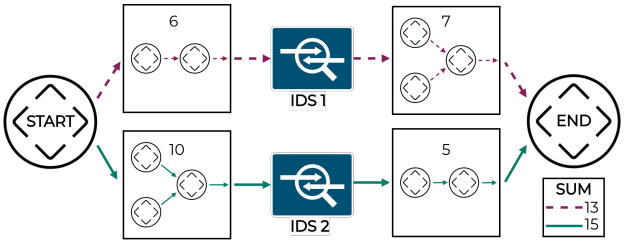
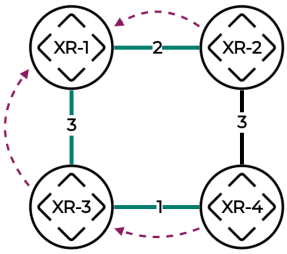


Fig. 3. First Calculation Approach



Router	Predecessor	Shortest Distance
XR-1	XR-1	0
XR-2	XR-1	2
XR-3	XR-1	3
XR-4	XR-3	4

Fig. 4. Preprocessed Values

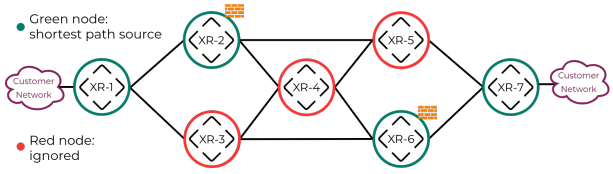


Fig. 5. Preprocessing Optimization

A. First approach

In the first approach, a whole shortest path with the help of Dijkstra's algorithm for each sub-path was calculated. All the paths were considered and calculated. The application also calculated paths that were not appropriate because they lead over services, which are not the most suitable ones. Consequently, these non-optimal paths were finally not considered, but the time and effort to calculate them are not negligible. The process was already optimized by cancelling a path calculation if the path already had a higher total cost than the current best one. However, it was overhead to calculate the whole path information continuously.

Figure 3 serves as an example for this suboptimal behavior. The previous implementation first calculated all Equal-Cost-Multi-Paths (ECMP) for the dashed path and saved the result. Next, it calculated all ECMPs for the green path compared it to the current best path and noticed that the path was worse. Therefore, the second calculation was fully processed, even if it was not considered as suitable in the end. This was a major drawback which should be improved.

B. Preprocessing

We have introduced a valuable and specific preprocessing. The preprocessing helps to improve the calculation efficiency. The idea behind is straightforward: Before a calculation is triggered, information relevant to the calculation should be collected, that could be reused later by the calculations and thus enhance the calculation.

The first step is to identify which calculation-relevant characteristics should be calculated in the preprocessing. Usually, without preprocessing, Dijkstra's shortest path algorithm calculates the shortest path between two nodes. By inspecting the algorithm, this process can be optimized: The preprocessing has to deliver a map containing the information of the shortest distance of a node and its preprocessor. As soon as the mapping has been calculated, the algorithm can use this information to calculate the shortest path efficiently. The shortest distance can be observed directly in the mapping table. The path can be observed by following the preprocessor entries until the preprocessor is the start node. Figure 4 shows an example of this procedure.

After the preprocessed values could be identified, an approach had to be found to execute the preprocessing. This process was related with research about possible algorithms. There are algorithms, which calculate the all-pairs shortest paths. So, these algorithms calculate the shortest path from each vertex to each other vertex in a graph. Johnson and Floyd-Warshall are such algorithms, that could be considered for the preprocessing. However, it has been shown that these algorithms do not have a smaller time complexity than execute different shortest-path calculations with Dijkstra's algorithm. [14]

Furthermore, an all-pairs shortest path algorithm is a calculation overhead because not every node could be a potential start node in a calculation. Especially in the particular use case of a service programming application, it was figured out that the approach with Dijkstra's algorithm is faster than an all-pair shortest path algorithm. As already mentioned, the calculation of the best service chain path has to be divided into sub-parts (compare fig. 2). A start node has to be a Provider-Edge (PE) router or a node with a service trailed in these sub-parts. Therefore, the preprocessing could be optimized with this insight: a preprocessing has only to be calculated for each potential start node. Because the number of these nodes is smaller than all vertices in the graph in a common provider network, the calculation with the Dijkstra algorithm is more performant than calculating the all-pair shortest paths.

The preprocessing with an optimization is illustrated in the fig. 5. Without that improvement, the application would calculate all shortest paths for all routers in the shown network graph, whereas the optimized solution only calculates the necessary information for the possible start nodes. The optimization avoids the unnecessary calculation of the shortest distance information for the intermediate nodes *XR-3, XR-4, XR-5* which are not possible start nodes in the shown network.

The result is the implementation of a preprocessing, which calculates the shortest distance and the preprocessor information for each node of interest with the help of Dijkstra's algorithm. Like mentioned before, possible starter nodes are regarded as these nodes of interest. Starter nodes in the application are PE routers, which have at least one customer

network attached, or nodes, which have at least one service connected.

Obviously, the preprocessing has to be triggered as soon as the application is started and the initial loading of the network information is done. Furthermore, there can be topology changes in the network that would require a new preprocessing to deliver an updated, correct result. For that reason, a new message which informs that information has been modified is introduced. The so-called *finish message* is sent when the processing of the BGP-LS updates [5] detect changed information and indicates that the polling interation has finished (see listing 1).

```
{
  "type": "info",
  "status": "finished",
  "message": "polling iteration finished",
  "data": {}
}
```

Listing 1. Finish Message

As soon as this message is retrieved, the backend service can check if new preprocessing is necessary. The decision if a preprocessing is necessary depends on the changes since the last *finish message*. If any changes are included that have influenced the network graph maintained in the application, a new preprocessing execution is necessary. Otherwise, the preprocessed information is still correct and does not need to be changed.

An example of an update that triggers an adjustment in the graph is a link addition, whereas for example a change of the router name does not change the graph behind the application. As soon as such a significant change has happened, a flag is saved in the cache, indicating that a new preprocessing is necessary. The big advantage of saving the flag in the cache is that it is possible to detect changes in the network even in the event of the crash of a backend container. This flag can be checked after the *finish message*. If it is set, the preprocessing has to be executed. This mechanism ensures that the preprocessing is only executed on demand. This on demand execution of the preprocessing represents a further optimisation that is part of the application.

C. Calculation Improvement

The calculation is optimized with the help of the introduced preprocessing method. The workflow of a calculation is visualized in fig. 6.

As soon as a calculation is requested, the different input parameters are validated. The request will be declined if the input is incorrect. An example of incorrect input would be a destination VRF (Virtual Routing and Forwarding) with no export tag in common with the source VRF which means that there is no IP connectivity.

If the input parameter is valid, the next step is to create all possible combinations of the services. A combinatorial approach is implemented with the help of the Cartesian Product, which returns all possible service combinations.

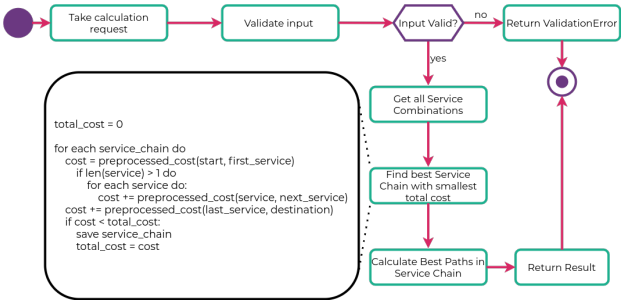


Fig. 6. Calculation Workflow

Once all the service combinations are created, the actual calculation can be started. The first step is to get the best service chain with the lowest possible total costs. This step is fundamental because the preprocessed smallest cost information can be used to get the service chain that is the best one. It has to be mentioned that if several equal service chains exist, the service chain which was found first is considered the best one. The best service chain with its total cost has already been found without any shortest path calculation thanks to the intelligent preprocessing strategy.

The only missing part is the exact hop-by-hop path, which is needed to display the complete path, from the source to the destination through the best service combination. The subsequent step is therefore to get the best path information out of the service chain. The path can be calculated with the help of the shortest path calculation and Dijkstra's algorithm. The number of shortest path calculation to be executed depends on the amount of included services. Minimum two shortest path calculations have to be executed (with one service) (see fig. 2). This process could also be optimised thanks to the predecessor information, which was collected previously. Hence, the actual paths do not have to be calculated but have to be looked up in the predecessor map. As soon as this step is finished, the final result can be returned.

III. Dynamic Adjustments and Verification

The policy verification is a vital part of the application. It ensures that the correct policy is enforced at any time in the network and that in case of a network change, the policy is going to be checked and modified if necessary. The goal is to verify the policies and ensure that the policies have the same status in the network and the application according to the latest network situation. This verification has to be performed after the network has converged (similar to the preprocessing).

After the *finish message* was received, the preprocessing has to be made to ensure that the calculation data is up-to-date. After the preprocessing is done, the verification of the policies has to be executed. A mechanism was introduced to decide which verification-relevant parts have changed since the last *finish message*. As soon as a relevant update message is processed, the information is updated. Subsequently, a verifier module has been written that checks if there have been any changes of interest. It works in a similar way as the check

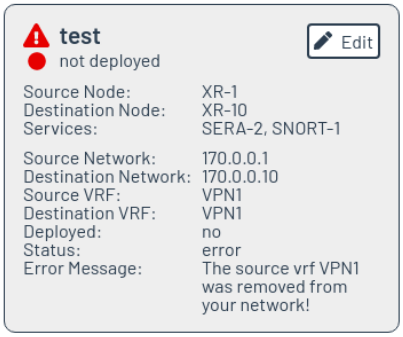


Fig. 7. Policy Error after network change

performed to verify if a preprocessing is necessary. If a router name has changed, there is no need to verify the policies. However, if a VRF has been removed from a router, this action is likely to influence existing policies and have to be checked entirely by the verification process.

A. Validation

After a network change has happened, the existing policies have to be checked and an evaluation has to take place to check if the changes influence their validity. Let's consider a policy that is in a valid state. After a network change, this policy is now in an incorrect state. The verifier is going to detect this problematic state and react accordingly, meaning the policy has to be put in an error state. This error state is a task that is part of the application and will be triggered automatically if there have been any changes leading to error policies. What the verifier exactly checks: if the source, destination and service information are still correct after the topology change.

A reverse check is also built into the application. This reverse check is triggered as soon as a network change has occurred which adds elements. This investigation is responsible for checking if the policy can be put into a valid state again. So, if the procedure can classify the whole policy data as correct, the policy can be re-enabled and put into a valid state again. An example of such a situation is if a router has rebooted. Once the router loses its connection, the application is informed, and the node is marked as down in the application. The policy verifier check will then detect automatically all policies that have installed this exact node as a source or destination node. Consequently, these policies will be in an error state until the node is enabled again. When the node is back online, the policies are back in a valid state if all other elements are also validated.

B. Policy Recalculation

Besides the policy validation, which ensures the correct state of all the policies at any time, it is also essential to always hold the correct result. This task is also included in the verification process.

After the validation process is completed, the application detects automatically if there are any policies to recalculate. So, if any major changes happen in the network, the application will react to these and recalculate automatically the affected policies. This has several positive effects. The first positive point of the intuitive recalculation is that it ensures that the result of each policy is always correct. Hence, this means the user can always observe in the GUI of the application the up-to-date policy with the latest result information - including network path, total cost, and the correct Segment ID (SID) list.

The notification about better paths is another advantage of this recalculation logic. After a topology change, the user is notified in the GUI about better path options. This status reveals that a new service combination is found, which has a smaller total cost than the current one. The user can now observe the changes and decide to accept the better path options. This case indicates that the old service chain is not the best anymore, but it is still working. Such a situation could happen if, for example, a new service is added to the network, which reduces the total cost of the policy.

Thanks to this implemented logic, it is also possible to survive network outages. The application finds automatically a possible path if there is any. If the most suitable firewall system is disconnected, the next best firewall system is found automatically. The user has, therefore, always a working result, if there is any.

This last step has a further improvement implemented in the application's logic: if the policy was deployed in the network and an outage impacts the service, the better policy will be calculated and redeployed automatically. This behaviour is necessary to ensure that traffic always gets the treatment, which the customer wants. It has to be mentioned that the different services should have the same configuration if the customer would like to have that feature. That could be established with a kind of "service function grouping" in which the services would share the same configuration.

IV. Architecture

Since Segment Routing is used in large and very large networks, especially in provider networks, the application must handle extremely large topologies and many users. For all that reasons, an application maintaining dynamic service programming policies have to be scalable and highly available, which explains the use of a cloud-native architecture with the deployment of the application on a Kubernetes cluster. That allows the application to scale horizontally and offers natively the possibility to activate autoscaling [15] functions [16].

The selected architecture is depicted in fig. 8. All the components and their tasks are briefly described below.

- **Storage System:** The storage system consists of a Redis cluster and a PostgreSQL database. The combination of cache and database has the advantage that the data stored in the cache is available extremely quick. Data that needs to be persisted or is sensitive is stored on the relational database. This combination has proven to be a very stable and fast solution. [17]
- **Messaging System:** The messaging system is the foundation for ensuring that the application can be deployed

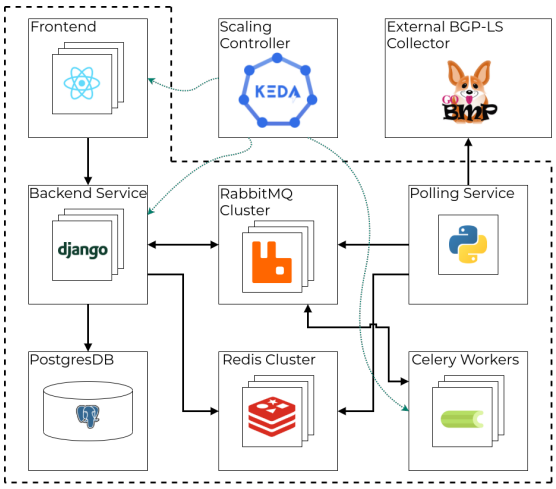


Fig. 8. Architecture of the service programming application

in a stable and scalable manner. The messaging system is used for complete communication between the different services. A RabbitMQ cluster is used to ensure that this central element of the application is always available. Since messages are now always stored in a queue, they can no longer be lost on the way between services, which makes the entire communication much more stable and trustworthy. The ability to consume messages in a controlled manner also provides the basis for scaling the application. [18]

- **Polling Service:** In order to continuously provide the whole application with the latest topology data, a dedicated service had to be developed that fetches and processes the latest data from an external BGP-LS collector [5].
- **Backend Service:** The backend is the most critical service in the whole application. As the brain of the application, it is responsible for all data delivery via an Application Programming Interface (API). For this purpose, a standardized REST API was implemented, making it as easy as possible for the frontend service to query and change data.
- **Frontend Service:** The frontend is responsible for the presentation of the data and functions provided by the backend. The frontend provides the end-user with a graphical user interface that allows them to use all the backend functionalities in a structured and straightforward way. The frontend communicates via the REST API, which is provided by the backend.
- **Workers:** The workers are responsible for the complete deployment to the different routers in the network. The workers enable the backend to commission deployment jobs asynchronously. Due to this asynchrony, the backend is not blocked, and the job is entirely outsourced to these workers. All deployment jobs are placed directly from the backend into the messaging system and picked up by the workers.

The complete deployment of the application is done using a dynamic Helm chart [19] which allows the complete application to be deployed on any Kubernetes [20] environment.

V. Conclusion

In this paper, a Segment Routing Service Programming application in form of a proof of concept is presented. The application allows the customer to manage the different policies from a central location. Constant manual adjustment of the various policies belongs to the past, thanks to automatic recalculation and redeployment. With the ability to dynamically route traffic through the various services, such as a firewall or an intrusion detection/prevention system, services can now be utilized better and centrally deployed in the network. The application can be seamlessly integrated into a cloud environment due to its cloud-native structure and can scale with the size of the network without any problems.

We believe that the Segment Routing Service Programming application has laid the foundation for service handling in modern networks. Nevertheless, the expandability seems almost infinite. There are so many characteristics and features that could be introduced in the future. The necessities of the different potential customers are almost unlimited. Still, there are not enough SRv6-aware services supported yet. The field of services, which can handle the Segment Routing technology have to be extended further. Although there are still some open questions to clarify, one thing is sure: Service Programming is the future.

References

[1] S. Dellsperger and J. Klaiber, “Service chaining path calculation,” 2020. [Online]. Available: https://eprints.ost.ch/id/eprint/912/

[2] F. Clad, X. Xu, C. Filsfils, D. Bernier, C. Li, B. Decraene, S. Ma, C. Yadlapalli, W. Henderickx, and S. Salsano, “Service Programming with Segment Routing,” Internet Engineering Task Force, Internet-Draft draft-ietf-spring-sr-service-programming-04, Mar. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-spring-sr-service-programming-04

[3] C. Filsfils, K. Michielsen, and K. Talaulikar, *Segment Routing*. Independently published, 2016.

[4] A. Abdelsalam, “Service function chaining with segment routing,” Ph.D. dissertation, Università di Roma Tor Vergata, 2020.

[5] gobmp. [Online]. Available: https://github.com/sbezverk/gobmp

[6] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, “Segment Routing Architecture,” RFC 8402, Jul. 2018. [Online]. Available: https://rfc-editor.org/rfc/rfc8402.txt

[7] P. Psenak, S. Hegde, C. Filsfils, K. Talaulikar, and A. Gulko, “IGP Flexible Algorithm,” Internet Engineering Task Force, Internet-Draft draft-ietf-lsr-flex-algo-17, Jul. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-lsr-flex-algo-17

[8] C. Filsfils, K. Talaulikar, D. Voyer, A. Bogdanov, and P. Mattes, “Segment Routing Policy Architecture,” Internet Engineering Task Force, Internet-Draft draft-ietf-spring-segment-routing-policy-13, May 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-spring-segment-routing-policy-13

[9] 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://rfc-editor.org/rfc/rfc8986.txt

[10] C. Filsfils, D. Dukes, S. Previdi, J. Leddy, S. Matsushima, and D. Voyer, “IPv6 Segment Routing Header (SRH),” RFC 8754, Mar. 2020. [Online]. Available: https://rfc-editor.org/rfc/rfc8754.txt

[11] F. Clad, X. Xu, C. Filsfils, D. Bernier, C. Li, B. Decraene, S. Ma, C. Yadlapalli, W. Henderickx, and S. Salsano, "Service programming with segment routing," *Internet Engineering Task Force, Internet-Draft draft-xuclad-spring-sr-serviceprogramming*, 2018.
[12] A. Abdelsalam, S. Salsano, F. Clad, P. Camarillo, and C. Filsfils, "Sera: Segment routing aware firewall for service function chaining scenarios," in *2018 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE, 2018, pp. 46–54.
[13] ——, "Sr-snort: Ipv6 segment routing aware ids/ips," in *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2018, pp. 1–2.
[14] H. Abu-Ryash and A. Tamimi, "Comparison studies for different shortest path algorithms," *International Journal of Computers and Applications*, vol. 14, no. 8, pp. 5979–5986, 2015.
[15] Keda. [Online]. Available: https://keda.sh/
[16] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-native applications," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, 2017.
[17] Redis caching system. [Online]. Available: http://redis.io/
[18] Rabbitmq message broker. [Online]. Available: https://rabbitmq.com/
[19] Helm kubernetes package manager. [Online]. Available: http://helm.sh/
[20] Kubernetes. [Online]. Available: http://kubernetes.io