Enabling Scalable SFCs in Kubernetes with eBPF-based Cross-Connections

Enabling Scalable SFCs in Kubernetes with eBPF-based Cross-Connections

Francesco Monaco, Giuseppe Ognibene, Federico Parola, Fulvio Risso
Department of Computer and Control Engineering
Politecnico di Torino, Italy
name.surname@polito.it

***Abstract*—Service Function Chains (SFCs) are composed of an ordered set of Network Functions (NFs) that provide network services to the handled traffic. However, traffic is highly variable over time, thus telco operators need to deploy scalable chains that can quickly and easily adapt to the load fluctuations. Although Kubernetes has already brought benefits in terms of increased scalability and flexibility to general-purpose applications, it is not natively suitable for network workloads since it lacks some functionalities required by network services. This paper presents a simple, cloud-native architecture that integrates SFCs in Kubernetes, with the aim of seamlessly leveraging cloud-native features such as horizontal autoscaling. The solution is based on flexible cross-connections, namely logical links that connect adjacent network functions, which can promptly adapt the distribution of the network traffic to the existing network functions in case of scale in/out events affecting the number of NF instances. The architecture has been validated with an open-source proof-of-concept implementation based on dedicated Kubernetes operators and an eBPF load balancer, demonstrating the feasibility and the efficiency of the proposed approach.**

I. Introduction

With the ongoing softwarization of the network and the advent of Network Function Virtualization (NFV), Network Functions (NFs) have been increasingly deployed as software rather than hardware appliances. The recent trend towards the increasing use of microservices instead of huge, monolithic functions, led to the necessity to define a way to create chains of NFs, forming Service Function Chains (SFCs), which are characterized by an ordered set of NFs traversed by the traffic in a precise sequence. Since the amount of traffic traversing the chain is highly variable, SFCs must be able to dynamically adapt to the current network traffic load, hence optimizing the use of resources and improving the quality of the offered service. With the advent of cloud-native NFs [1], new possibilities have opened up for the implementation of more flexible SFCs that can benefit from cloud-native environments. In particular, cloud-native infrastructures such as Kubernetes include the logic to provide automatic scalability of the running services within their core functionalities. However, although Kubernetes is currently the de facto standard orchestrator for general-purpose applications, it lacks some functionalities required by NF workloads, such as the possibility of defining service chains with precise network topologies and configurations, and the possibility to have pods with multiple network interfaces. Furthermore, the *Service* abstraction provided by Kubernetes, which leverages horizontal scaling and provides load balancing for applications, is not suitable for NFs since typically they are not the final recipient of the network traffic.

To address the aforementioned limitations of Kubernetes while bringing several of its advanced features also to the world of SFC services, we propose a model that integrates SFCs in Kubernetes, with the explicit goal of maximizing the reuse of existing Kubernetes features (in particular, horizontal pod autoscaling), while current SFC technologies tend to propose dedicated mechanisms for SFCs. This simplifies the creation of cloud-native SFCs and, with respect to automated scaling, it leads to a higher efficiency thanks to the capability to dynamically adapt to the actual traffic load. In fact, the proposed solution enables each NF to be independently scaled in an arbitrary number of replicas, hence providing flexible cross-connections between adjacent NFs instances in the chain.

This paper is structured as follows. Section II presents the current state-of-the-art. Section III details our model for a scalable cloud-native SFC, while Section IV provides a brief insight of a proof-of-concept implementation. Finally, the experimental evaluation is presented in Section V, while Section VI concludes the paper.

II. Related Work

Several approaches for the provisioning of SFCs based on Software Defined Networking (SDN) and NFV technologies can be found in the literature, which has been comprehensively analyzed in [2]. Recently, the growth of cloud-native technologies for NFs and the success of Kubernetes as orchestration platform paved the way towards new techniques for SFCs provisioning, such as in [3] and [4].

A solution based on the Network Service Mesh (NSM) framework [5] was proposed in [3]. NSM allows individual workloads to securely connect to Network Services, independently of where they are running. A Network Service can be composed of a chain of Endpoints, which actually implement the NFs. NSM provides and manages all the necessary interconnections mechanism to let the traffic pass from the client workloads to the endpoints of the requested Network Services. When a client workload requests a particular Network Service, NSM creates the necessary interfaces on the client and on the endpoints, and configures the underneath forwarding mechanisms in order to steer the traffic across the chain. However, NSM does not enable any efficient load balancing among NF Endpoint replicas. For this reason, the authors included

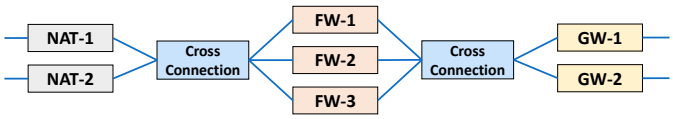

Fig. 1: An example of a scalable SFC composed of a NAT, a Firewall and a Gateway.

a network-aware load balancing system in order to leverage all the instances of a particular Endpoint.

A framework based on Contiv/VPP [6] that integrates SFCs in Kubernetes was proposed in [4]. It consists in a Kubernetes Container Network Interface (CNI) plugin that uses FD.io VPP to provide network connectivity between pods. Contiv-VPP supports pods with multiple custom interfaces and enables chaining between pods. Since Contiv-VPP enables only SFCs that are composed of single NFs instances, scaling requires the replication of the whole chain, with a further load balancer that distributes the traffic among the different paths.

Unlike these two solutions that require either (*i*) the creation of new interfaces and links for each client session or (*ii*) the replication of the whole SFC, our solution allows to define NF cross-connections only once, when the chain is initialized, and leverages the native Kubernetes autoscaling at the NF-level granularity, improving the efficiency and simplicity the solution. This was achieved introducing the new concept of flexible-cross connections, which connect replicas of adjacent NF and are dynamically adapted in case of scaling events.

III. System Design

A. Goals

Since Kubernetes does not provide any abstraction to support SFCs, our solution (*i*) introduces a proper model for SFCs, and (*ii*) defines the logic required to support multiple and independent NFs replicas in the chains, while leveraging existing Kubernetes features to the maximum extent. In details, our work addresses the following four goals.

Declarative definition of SFCs. SFCs must be created with a simple declarative description that includes only (logical) NFs and their interconnecting links, without any further low-level detail, such as the number of NF replicas in each stage of the chain, or how the traffic is distributed among existing replicas.

Automatic support for multiple replicas of a NF. Each NF in the chain may be executed with an arbitrary number of replicas (e.g., Figure 1). This enables the creation of multiple paths across NFs of the chain for optimal workload distribution, which demands for a clever traffic distribution mechanism that is implemented transparently by our solution.

Automatic adaptation of the chain to the variation of replicas. Each NF in the chain must be able to scale (e.g., in/out) automatically, mostly leveraging existing Kubernetes mechanisms (e.g., horizontal autoscaling). This requires our solution to dynamically adapt the cross-connections between consecutive NFs to keep them aligned with the number of instances present at any given moment. This ensures that new replicas are used and no traffic is forwarded to deleted replicas,

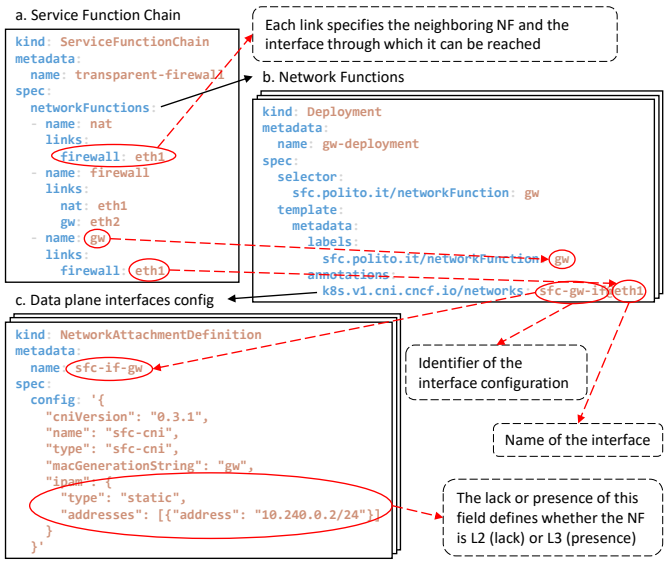

Fig. 2: Description of a simple SFC, referred to the chain of Figure 1.

which can be achieved through consistent hash mechanisms to keep established sessions to the existing replicas.

Support for L2/L3 NFs. The solution must support both Layer 3 NFs, which have a MAC/IP address on their interfaces (e.g., a router), and NFs operating as bump-in-the-wire transparent middleboxes, processing all traffic flowing through them regardless of L2 addressing (e.g., a transparent firewall).

This work leverages existing Kubernetes mechanisms to decide when and how to scale NF replicas; hence, how to allocate resources and where to schedule each pod. However, pure standard Kubernetes algorithms might not always take the optimal decision for SFC workloads; as illustrative examples, the default autoscaling criteria is based on CPU load, which is not appropriate for NFs running according to the busy-polling model; the default scheduling policy may not recognize the necessity to run two consecutive NFs on the same server to minimize I/O traffic, and more. The solution consists in creating additional Kubernetes components that, for example, export new metrics to drive the autoscaling mechanisms (e.g., based on the amount of traffic instead of the CPU used), or additional policies (e.g., affinities) to influence the decisions of the scheduler. Nevertheless, our approach enables reusing a large amount of tested code already present in Kubernetes, which greatly reduces the number of problems that need to be considered when running a SFC.

B. Modeling SFCs

We modeled a SFC as a list of nodes, each one specifying a NF, and a logical connection for each data plane interface of the NF (Figure 2a) that defines the next NF that can be reached through the interface. Each NF (Figure 2b) is characterized by one or more data plane interfaces, each one with its own network configuration (e.g., MAC/IP addresses, etc.; Figure 2c). One additional interface could be natively handled by Kubernetes and attached to the main CNI plug-

in, which can be used for management purposes. Since a running NF can include one or more replicas, interfaces of each replica have exactly the same network configuration (including IP/MAC address, if present), hence making each replica just a new pool of computing resources that is (from the data plane point of view) indistinguishable from other ones.

This high-level connection model enables also the support for asymmetric NFs, e.g., a NAT, which apply a different processing to traffic according to its direction.

C. Modeling flexible cross-connections

To enable the connectivity between multiple replicas of adjacent NFs, we need a flexible *cross-connection* that connects adjacent groups of NF replicas according to the corresponding logical connections specified in the SFC models. Each instance of the cross-connection is two-sided and bidirectional, i.e., it can only manage traffic between two consecutive NFs, with packets that can flow in both directions. The number of handled replicas may be asymmetrical on the two sides and also variable over time.

Flexible cross-connections, compared to simple point-to-point links, have to manage a higher level of complexity deriving from the larger number of possible connections between NF groups. Furthermore, they provide traffic forwarding capabilities and all the necessary logic to distribute the traffic among all the NF replicas. The logic that determines how the traffic is distributed is based on network sessions (e.g., TCP/UDP 5-tuple), coupled with consistent hashing mechanisms to ensure that all the packets belonging to the same sessions are always forwarded to the same NF instances. This association between session flows and NF replicas occurs for each step of the SFC. As a result, the traffic of each session traverses a path composed of exactly one replica for each NF. All the above characteristics are transparent to NFs, which are not aware of the presence of any intermediate cross-connection, nor have any knowledge about the preceding/following NF replicas.

IV. Implementation Overview

This section presents an open-source[1] PoC implementation of our model. Since the system was conceived to be integrated with Kubernetes, some of the prototype components have been designed to leverage the extensibility features of the aforementioned platform. This is the case of the *SFC CNI plugin*, which allows to configure the data plane interfaces of NF pods, and of the *SFC Operator* which acts as a manager for SFCs resources in the cluster, allowing to instantiate chains given their logical model. The other fundamental component is the *eBPF Load Balancer*, which implements the flexible cross-connections between NFs.

A. SFC CNI Plugin

The SFC CNI plugin allows to configure the L2/L3 data plane interfaces on NF pods, and has been designed to operate in conjunction with Multus CNI [7]. Multus is a meta-plugin that allows the use of multiple CNI plugins, to support multiple NICs in Kubernetes. By default, Kubernetes allows to have only one NIC for each pod, which is connected to the main cluster network. However, one NIC is usually not enough for NFs, since they typically need at least two additional interfaces (ingress and egress).

The SFC CNI plugin configures `veth` pairs that connect the pods with the network namespace of the host on which they are scheduled. For what concerns IP addressing, the system requires the use of the *static* IPAM plugin in order to assign the same addresses to all the replicas of a NF. Similarly, also the MAC address assigned by this plugin is required to be the same for groups of replicas. In addition, the SFC CNI plugin stores the details about interfaces configuration on the corresponding Kubernetes pod resource, in order to make this information available to the SFC Operator when it has to connect the pod to the eBPF-based load balancers.

B. eBPF Load Balancer

Our flexible cross-connections are based on eBPF [8], which has been chosen for its proven capability to create efficient network functions [9]; in fact, it can process the traffic on its natural path inside the Linux network stack, with clear benefits in terms of performance and transparency. The eBPF load balancer handles the traffic between two NFs, irrespective of the actual number of replicas. Hence, it can implement an N-to-M cross-connection between all `veth` interfaces associated to two consecutive NFs in the SFC, which terminate in the *host* network namespace.

The eBPF load balancer operates at the Traffic Control (TC) level in the Linux networking stack, leveraging the hook points of the `veth` on the host side. This allows the packets coming from the pods to be immediately processed as soon as they arrive at the ingress interface of the host, hence without touching in any way the container where the NF is running. We rely on some internal eBPF maps to keep all the information about connected NFs and handled sessions. In particular, they leverage two array maps containing the indexes of the NF interfaces connected on each side. The session table, which is a hash table, associates each handled TCP/UDP 5-tuple with two interface indexes: one is the index of the interface from which the first packet of the session was received and the other one is the index of the egress interface selected by the load balancing logic for that session.

This selection is made in two phases: (*i*) an hash function is applied to the session 5-tuple and (*ii*) the obtained value is used as an index to select an entry from the array map of the egress interface indexes. A more elaborate consistent hashing mechanism could be used to limit the shuffling of sessions that can not be stored in the session table when the latter is full, but this optimization is left as a future work. Furthermore, an additional hash map is used as a small ARP cache to facilitate the management of the ARP protocol across the chain. In fact, since the IP/MAC addressing is the same for each group of replicas, the ARP request/replies are always identical. Responses can therefore be cached so that, after the

[1] https://github.com/fmonaco96/sfc-k8s

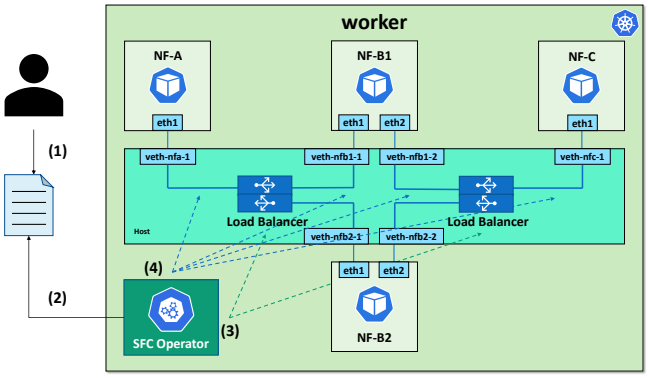

Fig. 3: Chain creation process

first reply, load balancers are able to respond immediately with the cached reply, without having to propagate the request across the chain.

C. SFC Operator

The SFC Operator acts as a manager for the system implementing the *Operator Pattern* of Kubernetes. It operates as a controller for `ServiceFunctionChain` and `LoadBalancer` custom resources, which represent the SFCs and eBPF Load Balancers instances in a cluster and are stored in `etcd`. This operator runs as privileged DaemonSet in the hosts' network namespace, due to the necessity to manipulate the network stack and attach the eBPF Load Balancers to `veth` interfaces. Moreover, it watches the Kubernetes resource representing the NF pods in order to immediately detect if there is a new replica or an existing one is going to be deleted.

The SFC Operator has been implemented using Kopf [10], a framework that allows to write Kubernetes Operators in Python. This allowed us to leverage the BCC toolkit [11] to handle the eBPF Load Balancer instances. In this way, it was possible to create a single program that could deal with the management of events concerning SFC cluster resources, but also with the practical aspects concerning the configuration of the data plane of eBPF Load Balancers. When the SFC Operator notices that there is a new SFC resource in the cluster, it creates the load balancers across the chain, compiling and injecting the corresponding eBPF code in the Linux kernel. It then configures the TC hook of the NFs interfaces so that all the generated traffic is handled by the load balancing logic, providing a logical link between NFs and load balancers. During the operation of the chain, when it detects an event on NF replicas, it updates the configuration of adjacent eBPF Load Balancers so that they are always aligned to work with the instances present at the moment.

D. SFC Lifecycle

The lifecycle of a SFC is determined by its corresponding `ServiceFunctionChain` resource. These resources allow users to create SFCs in a fully declarative way without the need of any other further detail, leaving the practical operations to the SFC Operator. Figure 3 shows the main actions performed by the SFC Operator when it has to manage the creation of a new chain. As soon as a user applies a manifest of a SFC to the Kubernetes API Server, (1) the SFC operator watches the related event and (2) parses the provided description in order to obtain the chain structure. During this phase, for each couple of adjacent NFs, it creates and pushes to the API server a `LoadBalancer` manifest and, at the same time, through another controller, it proceeds with the actual instantiation of the load balancers injecting their eBPF code in the kernel (3). After their successful creation, the SFC Operator proceeds with the creation of the links between NF pods `veth` interfaces and adjacent load balancers (4), configuring the TC hook of the pods `veth` interfaces so that all incoming packets are processed by the load balancing logic. Moreover, it pushes the interfaces indexes in the load balancer data structures so that they know to which NFs they are linked.

As concerns the deletion, when a user deletes the `ServiceFunctionChain` resource from the cluster, the SFC operator reacts removing the eBPF Load Balancers injected in the kernel and cleaning up all the TC hooks of `veth` interfaces attached to the NF pods.

E. NF scaling

In order to properly handle scaling and more generally the creation and the deletion of NF pods, the system reacts to two specific events during the lifecycle of the pod. The first one is the beginning of pod execution and the second one is the start of the pod deletion procedure. When the SFC Operator notices that a new NF pod has entered the running phase, it immediately proceeds with the operations needed to include it in the chain. First, it searches for all the eBPF Load Balancers it must be linked to, then it proceeds with the actual connection of the NF pod interfaces as described previously. On the other hand, as soon as the SFC Operator notices that a NF pod is going to be deleted, it removes from load balancers all the information related to that replica so that it will not be selected by the load balancing logic anymore. It is important to highlight that the above clean-up operations must occur before actually shutting down the pod in order to avoid dropping packets. This can be achieved by postponing the actual NF pod termination so that the SFC Operator has enough time to complete the cleanup, which can be done by leveraging the `PreStop` hook in the containers lifecycle, that allows the execution of a command before starting the termination phase.

V. Evaluation

This section presents a preliminary evaluation of the proposed architecture in terms of performance and reaction times. First, we compare our SFC scaling approach that operates on individual NFs against the alternative approach based on the scaling of the whole chain. Second, we evaluate the performance of the eBPF Load Balancers in terms of their throughput. Third, we measure the reaction time of the overall system at the occurrence of scaling events.

A. NF scaling efficiency

In our solution, the scaling of NFs is based on the autoscaling features of Kubernetes, whose logic operates on each

single NF deployment. This enables SFCs that can scale only the component that is stressed or underused, allowing better resource usage compared to other approaches that scale the whole chain. To highlight this aspect, we measured the amount of resources requested by an SFC when it scales out only a stressed NF as opposed to when it scales out the entire chain. The tested SFC was composed of three NFs: a firewall, a simple pass-through traffic policer, and a gateway. In terms of computational resources they requested[2] 200, 150, 100 milliCPU respectively for each instance. The firewall is the NF that is assumed to scale.

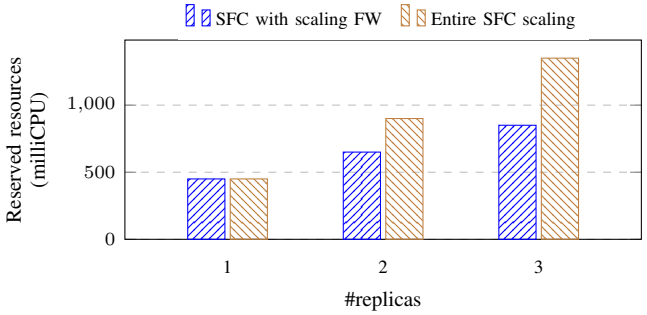

Fig. 4: Requested milliCPU as the number of scaled instance increase.

Figure 4 shows the results of the test in terms of milliCPU *requested* by the entire chain for the two approaches as the number of replicated instances increase. In case of interrupt-based NFs, the *requested* CPU does not imply it is actually consumed, hence it could be re-assigned to other services demanding more CPU power. However, this is possible only in case the over-allocation policy is allowed, which is usually discouraged in case of strong service guarantees. Vice versa, with NFs based on a busy-polling model (e.g., in case of DPDK-based NFs), the amount of CPU actually consumed is equal to the reserved value, independently of the amount of traffic to be processed, hence leading to an unnecessary waste of resources. Hence, this result confirms that the scaling performed on individual NFs allows to greatly decrease the resource utilization, avoiding unnecessary over-allocations or resources wasted by busy-polling NFs.

B. eBPF Load Balancer performance

In the traditional approach in which the whole chain is scaled, NFs can be directly connected through a point-to-point link (e.g., `veth`). Instead, in our architecture the fan-in/fan-out of each NF can be potentially greater than one, hence requiring the presence of a load balancer between NFs, which may introduce additional performance penalties. This section evaluates this additional cost of the eBPF load balancer, compared to other two interconnection mechanisms provided by Linux kernel: Virtual Ethernet (`veth`) and the standard *Linux bridge* software. In this test, physical connections between NF pods are created as follows:

- Virtual ethernet: the `veth` is used as a direct link between the NFs. One end of the `veth` is placed in the network namespace of the first NF and the other end is placed in the network namespace of the second NF.
- eBPF Load Balancer: each NF is connected to the host network namespace through a `veth` and the load balancer cross-connects the `veth` ends on the host side.
- Linux bridge: similar to the previous case, but in this case the `veth` head ending in the host network namespace are connected through a Linux bridge.

Tests address two different scenarios: (*i*) a chain composed of two pods, a client and a server; (*ii*) a chain composed of three pods, a client, a transparent firewall and a server. In these tests, each NF is deployed with a single replica. In both scenarios all the pods of the chain are executed on the same Kubernetes node, with the physical interconnections implemented through the above technologies. The throughput of each chain has been measured with `iperf3`, using TCP traffic with standard parameters.

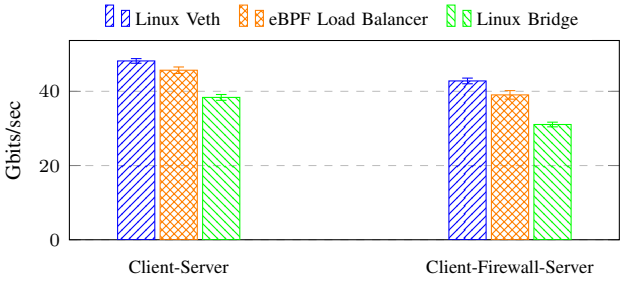

Fig. 5: Performance comparison between scenarios' sub-cases.

Results in Figure 5 show that the additional overhead introduced by the eBPF load balancer is very limited compared to the optimal case (`veth`), and much better than any alternative technologies, such as Linux bridge. This is due to the efficiency provided by the eBPF technology, which operates directly in-kernel, thus avoiding expensive context switching. Hence, despite the proof-of-concept (session-based) load-balancing logic, our implementation proved to be definitely faster than the standard Linux bridge.

C. Reaction Time

In order to evaluate the reaction time of the system, we measured the time required from a change of state of a replica to be noticed by the operator, and also how long it takes to the operator to configure/clean-up the interfaces and update the load balancing rules affected by the event. For the tests, a generic NF with two network interfaces has been used. For this reason, the add or remove operations on the interfaces are performed twice for each event. The time taken to update the load balancers eBPF maps is not shown in the results because its contribution is so low that can be considered negligible.

The first test evaluates the time elapsed from starting a pod (representing a new replica) until it is fully included in the

[2]In Kubernetes, *requests* determine the amount of resources reserved for a pod, which may be different from the consumed resource, which is usually smaller than the value above.

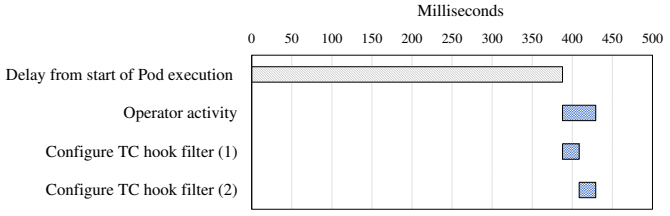

Fig. 6: Reaction time composition for a new replica event.

chain by the operator. Results in Figure 6 show that most of the delay is due to the propagation of the event related to the status of the running pod in the cluster, which falls under the responsibility of the standard Kubernetes logic, while the time required by our operator is much smaller. More in depth, the graph shows that, within the operator, the most time-consuming operations are the configuration of the two TC hook filters (before/after the NF) that connect the current NF to the rest of the chain through the eBPF load balancing logic.

The reverse case is the scale-in operation, which corresponds to the removal of one replica. In this case, the operator reacts as soon as it detects that a pod is going to be deleted. In this test we measured the time taken by the SFC to converge to the new state, starting from the instant in which the pod begins its graceful termination to when it is successfully removed from the chain and all the cross-connections are properly updated. The pod used in the tests was forced to delay the beginning of the termination procedure by 0.5 seconds.

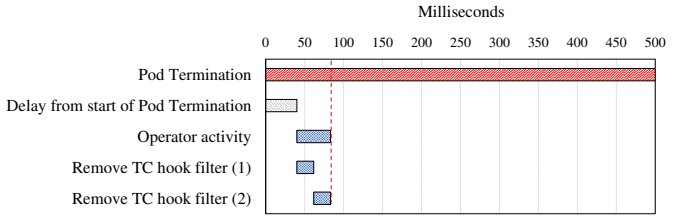

Fig. 7: Reaction time composition for a terminating replica event.

Figure 7 shows that the time spent by the operator is similar to the previous test. In this case the main contribution is given by the removal of TC hook filter from the `veth` interfaces. Even in this case the operator reacts after a slight delay as evidenced by the grey bar in the picture, due also in this case to the time to propagate the deletion event in the cluster. The dashed line highlights the moment in which the replica is no longer part of the chain, showing that the SFC converges much faster than the time required by Kubernetes to terminate the pod. This is a desired result, as the operator must complete his operations before the pod terminates in order to prevent traffic from being forwarded to a replica that no longer exists.

Finally, additional tests (not shown here for the sake of brevity) confirm that no packets are lost in the reconvergence process, hence making this operation loss-free. However, some traffic sessions are handled by a different replica than before, with possible problems in case of stateful NFs. A clever mechanism that avoids session redirections, which requires a smarter load balancer and a more sophisticated pod termination logic, is left to our future work.

VI. Conclusions

This paper presents a possible approach for highly scalable SFCs in a cloud-native environment such as Kubernetes. In particular, we propose a model to represent the concept of scalable SFCs and a PoC implementation to integrate them into the aforementioned platform. This work demonstrates the possibility to support SFCs with an arbitrary number of NF replicas that can change continuously over time, thus enabling the creation of chains that can dynamically adapt to the traffic load, with support for different L2/L3 network models.

Our experimental evaluation has shown that our solution achieves better results in terms of overall resource consumption compared to other approaches based on the scaling of whole SFCs. Tests on the performance of the eBPF-based cross-connections have shown that the maximum throughput is comparable with simpler direct links, emphasizing the low overhead deriving from the additional load balancing logic.

As part of our future work, we foresee the possibility to extend the load balancing implementation with a more advanced weighted logic based on the effective load on NFs, which could further optimize the use of NFs resources and improve the average quality of services offered by SFCs.

Acknowledgment

Federico Parola acknowledges the support from TIM S.p.A. through the PhD scholarship.

References

[1] T. U. Group. (2020) Cloud native thinking for telecommunications. [Online]. Available: https://github.com/cncf/telecom-user-group/blob/master/whitepaper/cloud_native_thinking_for_telecommunications.md

[2] K. Kaur, V. Mangat, and K. Kumar, "A comprehensive survey of service function chain provisioning approaches in sdn and nfv architecture," *Computer Science Review*, vol. 38, p. 100298, 2020.

[3] B. Dab, I. Fajjari, M. Rohon, C. Auboin, and A. Diquélou, "Cloud-native service function chaining for 5g based on network service mesh," in *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, 2020, pp. 1–7.

[4] A. Bouridah, I. Fajjari, N. Aitsaadi, and H. Belhadef, "Optimized scalable sfc traffic steering scheme for cloud native based applications," in *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*, 2021, pp. 1–6.

[5] T. N. S. M. Authors. Network service mesh. [Online]. Available: https://networkservicemesh.io/

[6] T. C.-V. Authors. Contiv-vpp. [Online]. Available: https://contivpp.io/

[7] T. M. Authors. Multus. [Online]. Available: http://multus-cni.io/

[8] M. Fleming. (2017) A thorough introduction to ebpf. [Online]. Available: https://lwn.net/Articles/740157/

[9] 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.

[10] K. Authors. Kubernetes operator pythonic framework (kopf). [Online]. Available: https://github.com/nolar/kopf

[11] T. B. Authors. Bpf compiler collection (bcc). [Online]. Available: https://www.iovisor.org/technology/bcc