A Service-Agnostic Software Framework for Fast and Efficient In-Kernel Network Services

A Service-Agnostic Software Framework for Fast and Efficient In-Kernel Network Services

Sebastiano Miano*, Matteo Bertrone*, Fulvio Risso*,
Mauricio Vásquez Bernal*
*Department of Computer and Control Engineering
Politecnico di Torino, 10129, Italy
name.surname@polito.it

Yunsong Lu†, Jianwen Pi, Aasif Shaikh
†Futurewei Technologies, Inc.
Santa Clara, California
yunsong.lu@futurewei.com, jianwpi@gmail.com,
acloudiator@gmail.com

***Abstract*—This paper presents Polycube, an open-source software framework based on eBPF, that enables the creation of arbitrary and complex network function chains. Each function can include an efficient in-kernel data plane and a flexible user-space control plane with strong characteristics of isolation, persistence (e.g., across server reboots) and composability. In addition, a generic model for the control and management plane of each network function simplifies the manageability and accelerates the development of new network services. We validate the framework by creating different network services and benchmarking their performance in a complex scenario, namely a network provider for Kubernetes. Results show that Polycube programs are about 20x shorter than equivalent programs implemented with vanilla-eBPF.**

***Index Terms*—eBPF, XDP, Linux, NFV, Service Chaining**

I. Introduction

Network Functions Virtualization (NFV) enables network services to be transformed in pure software images that are executed on standard servers. This technology guarantees lower costs thanks to the reduction of the number of physical appliances [1] and to the possibility to rely on (cheap) commodity hardware. At the same time, it enables more agile services thanks to the click-and-play nature of the software.

The most common approach to NFV is through a set of (chained) VMs or containers, connected by means of a virtual switch [2]. This often includes heterogeneous applications, built from different vendors, with diverse characteristics e.g., in terms of configuration protocols and life-cycle management, which complicates day-by-day operations [3]. This heterogeneity impairs also on the possibility to achieve higher throughput through cross-VNF optimizations, as each application operates in isolation. For instance, even simple approaches, such as zero-copy or shared memory between cascading functions (e.g., [3]–[6]), are often very difficult to deploy in practice. Furthermore, advanced features such as service decomposition [7], fault-tolerance [8], high availability [9], [10], should be provided separately per each VNF, complicating the design of the control plane and making the system more complex [11]. Finally, the computational requirement for the above VNFs is often huge, due to the number of different components involved (e.g., hypervisors, VMs with their guest operating systems, vSwitch, etc.) not to mention the cost in moving a packet during its journey, due to the many components traversed and the several transitions between kernel and user space.

In this paper we present a novel software architecture that simplifies the creation, deployment and management of in-kernel network functions. In particular, it offers *(i)* the possibility to implement complex functions on the data plane, *(ii)* enables the creation of arbitrary service chains, hence simplifying the creation of complex services through the composition of many elementary components and *(iii)* offers a service-agnostic interface that decouples the control and management logic (which is generic and valid for all services) from the actual service logic, hence enabling the dynamic and seamless deployment of arbitrary network services. In summary, this paper makes the following contributions:

- We show the design and architecture of Polycube (section III).
- We provide a description of the APIs and abstractions provided to the developers to simplify the development of new services (section IV).
- We demonstrate the practical benefits and of the Polycube programming model with a complex application, namely a network provider plugin for Kubernetes (section V).

Polycube source code, documentation and implementations of the various network services are available at [12].

II. Context and Motivation

We now provide a set of desired properties for a network application and motivate the choice of using eBPF as underlying technology; then, we describe the challenges that need to be addressed by Polycube to enable this new design.

A. Desired properties for NFs

In-kernel packet processing. Current options to build NFs rely on either kernel bypass approaches, implementing all the functionality in user-space, or in-kernel processing where packets are entirely handled within the kernel context. Although the former approaches bring unquestionable performance improvements, they *(i)* take the ownership of one (or more) CPU cores, thus permanently stealing precious CPU cycles to other tasks [13] and *(ii)* require to re-implement the entire network stack in userspace [14].

Specialization. Typical kernel implementation result limited and general to reduce the number of changes required when adding a new feature [2]. A desirable property for a NF would be to let the user defining the behavior of the program, making it specialized for the current applications and workloads.

Easy development process. To introduce greater programmability and additional functionality inside the kernel, current approaches rely on custom kernel modules (e.g., OvS), which may, however, result difficult to maintain and distribute [2], [15]. A kernel application should instead follow the same development process of userspace applications, which may improve the delivering of new functionality and diversify the existing implementations.

Integration with other kernel subsystems. With the recent changes in the data center workloads, applications are now running on the host operating system that is then shared between different applications. These application often rely on existing kernel functionality to accomplish their tasks. It is then important that kernel network application can easily interact with other kernel-level data structures (e.g., FIB or neighbor table) and leverage kernel functionalities (e.g., TSO, skb metadata, etc.).

B. Extended Berkley Packet Filter (eBPF)

The extended Berkeley Packet Filter (eBPF) [16], [17], recently introduced in the Linux kernel, can be used to create network functions while guaranteeing the above desired properties. First of all, eBPF can process a packet entirely in kernel space, without context switches or packet copies between kernel and user space [13]. Second, it leverages a set of features that are already present in a modern Linux kernel, without requiring additional kernel modules that are difficult to create and maintain. Third, the possibility to compile and inject the code at runtime paves the way to a context-based customization of each network function. Last but not least, eBPF programs can cooperate with the kernel TCP/IP stack, possibly complementing existing networking features.

C. Challenges

Creating network functions with eBPF is sometimes complex given the lack of a common framework that provides useful abstractions to developers to solve common problems or known limitations [18]. In fact, even though eBPF allows more complex data plane processing than OpenFlow [19], it is not Turing-complete. As a consequence, it cannot support truly arbitrary processing, making the implementation of some common services (e.g., ARP handling in a router) challenging. Furthermore, no abstractions current exist to implement the (complex) control plane of a service, hence forcing developers to dedicate a considerable amount of time to handle common control plane operations (e.g., user-kernel interaction). This motivates us to rethink how those network functions are designed and managed. We envision a novel software architecture, called Polycube, wherein eBPF based network services are consolidated to run on the same host, and managed in a logically centralized manner.

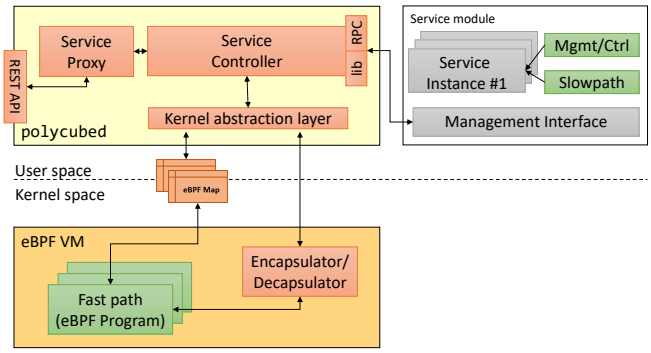

Fig. 1: High-level architecture of the system

III. Architecture Overview

This Section introduces first the main ideas that inspired the design of Polycube; then it will present the resulting software architecture and the most significant implementation details.

A. Unified point of control

All Polycube network functions feature a unified point of control, which enables the configuration of high-level directives such as the desired service topology. In addition, it facilitates the provisioning of cross-network function optimizations that could not be applied with separately managed services. Polycube supports this model through a single, service-agnostic, userspace daemon, called `polycubed`, which is in charge of interacting with the different network function instances. Each different type of virtual function is called `Cube`, which are similar to *plugins* that can be installed and launched at runtime. A new Cube can be easily added to the framework by a specific registration phase, in which the service sends the information required for its identification within the framework, such as the service type (i.e., local or remote, Section III-B3) or the minimum kernel version. When the service is registered, different instances of it can be created by contacting `polycubed`, which acts mainly as a proxy; it receives a request from a northbound `REST` interface and forwards it to the proper service instance, returning back the answer to the user.

B. Structure of Polycube services

Each Polycube service is made up of a *control plane* and a *data plane*. The *data plane* is responsible for per-packet processing and forwarding, while the *control and management plane* is in charge of service configuration and non-dataplane tasks (e.g., routing protocols). Although this separation between the control and data plane is common in many network functions architectures, Polycube provides a clear separation between these components; each service is composed of a set of standard parts that make it easier for the programmers to implement the desired behavior, while Polycube takes care of creating all the surrounding glue logic,

handling all the interactions and communications between the different components.

1) Data plane: The data plane design of a Polycube service is characterized of a *fast path*, namely the eBPF code that is injected into the kernel, and a *slow path*, which handles packets that cannot be fully processed in the kernel or that would require additional operations, slowing down the processing of the other packets.

Fast path. The data plane portion of a network service is executed *per packet*, with the consequent necessity to keep its cost as small as possible. When fired, the fast path retrieves the packet and its associated meta-data from the receive queues, then it executes the injected eBPF instructions. Typical operations are usually very fast, such as packet parsing, lookups in memory (e.g., to classify the packet), and map updates, such as storing data in memory (e.g., statistics), for further processing. When those operations are carried out, the fast path returns a forwarding decision for that particular packet or send it to the slow path for further processing.

Slow path. Although eBPF offers the possibility to perform some complex and arbitrary actions on packets, it suffers from some well-known limitations due to his restricted virtual machine, which however are necessary to guarantee the integrity of the system. Those limitations may impair the flexibility of the network function, which *(i)* may not be able to perform complex actions directly in the eBPF fast path or *(ii)* could slow down its execution, adding more instructions in the fast path to handle exceptional cases. To overcome those limitations, Polycube introduces an additional data plane component that is no longer limited by the eBPF virtual machine and it can hence execute arbitrary code. The *slow path* module is executed in userspace and interacts with the eBPF fast path using a set of components provided by the framework. The eBPF fast path program can redirect packets (with custom meta-data) to the slow path, similar to Packet-In messages in OpenFlow. Similarly, the slow path can send packets back to the fast path; in this case, Polycube provides the possibility to inject the packet into the *ingress* queue of the network function port, simulating the reception of a new packet from the network, or into the *egress* queue, hence pushing the packet out of the network function.

2) Control and management plane: The control plane of a virtual network function is the place where out-of-band tasks, needed to control the data plane and to react to possible complex events (e.g., Routing Protocols, Spanning Tree), are implemented. It is the point of entry for external players (e.g. service orchestrator, user CLI) that need to access service's resources, modify (e.g., for configuration) or read service parameters (e.g., reading statistics) and receive notifications from the service fast path or slow path. Polycube defines a specific *control and management* module that performs the previously described functions. It exposes a set of `REST` APIs used to perform the typical CRUD (create-read-update-delete) operations on the service itself; these APIs are automatically generated by the framework starting from the service description, removing this additional implementation overhead to the programmer. To interact with the service, an external player has to contact `polycubed`, which checks the service to which the request is directed to and dispatches it to the corresponding service control path, which in turn serves the request modifying its internal state or reflecting the changes to the service data path instance.

3) Remote vs Local services: The separation between the data and control plane allows to execute the two components separately, not necessarily on the same server. While the former is running in the server, the latter can be executed either *locally* or *remotely*. Polycube may support both *local* services, installed as local applications on the same server of `polycubed` and whose interaction is through direct calls, and *remote* services, possibly running on a different machine and communicating with `polycubed` through RPC mechanism. When a new service is plugged into the framework, it communicates to `polycubed` if it is local or remote. In the first case, the path to the service executable (i.e., a dynamic library file) is specified and `polycubed` loads the library at runtime, forwarding the requests to the service control path as a normal function call. In the second case the service is registered by providing the remote RPC endpoint; in that case all subsequent requests for that service will be redirected through the RPC channel. Polycube provides a *management interface* that allows to control any service data plane, regardless of the service type and structure, hence being agnostic to the control plane location. It allows to get access to any registered service in the same way from its `REST` interface, facilitating the service developer who does not have to deal with the low-level details of the communication with the daemon.

C. Management and Control

The capability to add (or remove) a network function dynamically (even from a remote server) into `polycubed` provides several advantages such as the possibility to update an existing service, adding functionality without modifying the network functions currently deployed and running. To support this model, the Polycube core (i.e., `polycubed`) has been designed to be completely independent from the type of network function that is installed; it has no idea of how the network function is composed internally or what are its functionalities, `polycubed` only takes care of forwarding the request to the proper service instance. This approach does not require changes to `polycubed` whenever changes to the individual service are needed; when the service is being updated, it is unplugged from the framework, updated and plugged-in again without affecting existing services. On the other hand, it complicates the service design, which has to define the interface to the outside (i.e., the `REST` APIs). To simplify this process, Polycube uses `YANG` [20] models, each one describing a specific service, to automatically synthesize the `REST` interface of the service.

Model-driven service abstraction. The YANG data modeling language allows to *(i)* model the structure of the data and the functionalities provided by the Polycube service, *(ii)* define the semantic of the service data and their relationship and *(iii)* express their syntax, which will be used to interact with the service itself. When a new service is registered, polycubed reads the provided YANG model and generates an internal representation of the service data together with a specific path mapping table used to access those data from outside. Whenever a new request for that service arrives, polycubed validates it (e.g., checking the correct format of an IP address, ports in a given range) according to the information specified in the YANG model, without having to rely on the service itself for those "ancillary" tasks.

D. Functional Service Decomposition

The data plane architecture of Polycube supports the composition of micro-functional blocks in more complex data path structures. In particular, the data plane of a single Polycube service can be created by stitching multiple eBPF programs together that are controlled and injected separately from the control plane using the Polycube service-independent APIs. This set of eBPF micro-blocks, called *micro-cubes (μCubes)*, are part of a single Cube instance and are handled by an unique control plane and slow path module, as opposed to different Cube instances that have their own controllers. This modular design introduces the necessity to specify an order of execution of the μCubes inside the NF; in fact, when a packet reaches a Cube composed of different micro-blocks, Polycube has to know the first module to execute, which in turn will trigger the execution of the others μCubes within an arbitrary order based on its internal logic. To do this, Polycube introduces the concept of MASTER and SLAVE μCubes. The MASTER μCube, which is unique within the Cube itself, represents the entry point of the entire service and its execution is triggered upon the reception of a packet in a port. Subsequently, there are multiple SLAVE μCubes, whose execution is triggered only through a direct call from another μCube. Having a different set of eBPF programs, each one performing a specific function, is useful in particular for two reasons. First, it allows the developer to handle each feature separately, enabling the creation of loosely coupled services with different functionalities (e.g., packet parsing, classification, field modification) to be dynamically composed and replaced; each single μCube can be substituted at runtime with a different version or can be directly removed from the chain if its features are not needed anymore. Second, it can be useful to overcome some well-know eBPF limitations such as the maximum size of an eBPF program or the inability to create unbounded loops in the code.

IV. APIs and Abstractions

Polycube provides a set of high-level APIs and abstractions to the developers to simplify the writing of a new service, both from the control plane and the data plane point of view. It bases upon the BPF Compiler Collection (BCC) [21], extending the above abstraction with additional helper functions targeted to networking services. For example, it adds useful abstractions to manage special packets, to cope with special processing that may complicate (and slow down) the fast path, or to react to special events such as timeouts. Table I shows some of the main helper functions introduced by Polycube at different levels of the network service code, i.e., the eBPF fast-path, the slow path and the control and management plane.

Transparent port handling. A Polycube network service instance is composed by a set of virtual ports that are uniquely identified through a *name* and an *index* inside the service itself. Every port of the service can be attached to a Linux netdevice or to another service port by means of the *peer* parameter. When the fast path of the service decides to redirect the packet to a specific output port it can use the pcn_pkt_redirect() function to send the packet to the next hop whether it is a net-device or another Polycube service. Although the implementations for the above two types of next hops are quite different, Polycube hides this difference by providing a generic helper that receives the virtual index of the output port and, if the port is connected to a netdevice, redirects the packet to the attached netdevice, otherwise jumps directly to the next Polycube network function in the chain.

Fast-slow path interaction. If the packet currently processed in the eBPF service fast path requires additional inspections or further processing, it can be sent to the slow path module of the service by means of the pcn_pkt_redirect_-controller() helper. This function receives as parameters the reason why the packet has to be sent to the slow path and, optionally, additional meta-data fields. Polycube hides the implementation details of the communication between the eBPF fast path program and the service slow path; it sends the packet to an eBPF control module (the *Decapsulator* and *Encapsulator* shown in Figure 1), that will copy the packet and its meta-data to userspace, where they will be received by the Polycube daemon, which will call the packet_in() function of the associated service's slow path. On the other side, when the slow (or control) path decides to send a packet out on a given port, it can use the send_packet_out() function, indicating the output port and the direction where to send the packet, which can be the INGRESS or EGRESS queue of the chosen port.

Debug mechanism. Polycube provides a debug helper that can be used in both fast and slow/control path to print debug messages. Although this feature is quite common for userspace programs, it is not the same for the eBPF programs, which are executed in the kernel context. Similar to kernel modules, eBPF programs use the bpf_trace_printk() function to print debug messages; once the program is loaded, the verifier checks whether program is calling this function and allocates additional buffers, which may slow-down the processing of the function. Polycube uses a more efficient mechanism through the pcn_log() helper; when called, this helper uses a *perf ring buffer* to send debug messages to polycubed, which

TABLE I: Helper functions provided by Polycube at different level of the NF.

Level	Helper function	Arguments	Description
Fast path (eBPF)	`pcn_pkt_redirect`	md, out_port	Redirect a pkt to an VNF interface (either physical or virtual)
Fast path (eBPF)	`pcn_pkt_controller`	reason	Send a pkt to the slow-path with a given reason
Fast path (eBPF)	`pcn_pkt_controller_md`	md, reason	Send a notification to userspace with a specific reason
Fast path (eBPF)	`call_ingress_program`	index	Call the μCube at a given index attached to the ingress pipeline
Fast path (eBPF)	`call_egress_program`	index	Call the μCube at a given index attached to the egress pipeline
Slow path (user)	`packet_in`	pkt, md, reason	Callback executed when a notification is sent to userspace
Slow path (user)	`send_packet_out`	pkt, dir	Send a packet out to the ingress or egress pipeline
Control plane	`reload`	code, idx	Reload the μCube at a given index with the new code
Fast/slow path	`pcn_log`	level, txt	Print debug messages with a given verbosity level

redirects them to the current log file, as for the slow and control path. This allows also to introduce additional network-specific custom modifiers (e.g., `%I`, `%M`, `%P`) that can be used to print IP, MAC addresses or port numbers in a human readable format. Finally, using different log levels, `polycubed` is able to dynamically remove all the references to the debug messages under the specified log level, reloading the service fast path to reflect the changes.

Table abstractions. To store the network function state across different runs of the same program or to pass configuration data from the control path to the fast path, a Polycube service uses eBPF tables, which are defined into the service fast path and are created when the program is loaded. Every eBPF table has a scope into the system, which expresses the possibility to read and/or modify the table content from another eBPF program. Polycube introduces the possibility to define `PRIVATE` tables, which are only accessible from the same μCube where they have been declared and `PUBLIC` tables, which are instead accessible from every μCube running in the machine. In addition, since Polycube supports the possibility to compose the network function data path as a collection of μCubes (i.e., simple eBPF programs), we added the concept of `SHARED` tables, where a table can in fact be shared between a given set of μCubes. In this case, when the table is instantiated, it is possible to specify the *namespace* within which this table will be shared.

V. Evaluation

In this section we first evaluate the overhead imposed by Polycube programming model when compared to baseline programs written using the vanilla eBPF (section V-A). Then, we evaluate a use case that show both the performance advantages of eBPF-based NFs and how Polycube enables the creation of complex network services by chaining different modules together (section V-B).

A. Baseline performance

To measure the baseline performance and the overhead introduced by the Polycube abstraction model to a single NF, we implemented the same operations performed by the `xdp_redirect` application, available under the Linux samples, as a standalone NF inside the Polycube framework (i.e., `pcn-simplefwd`). The application receives traffic from a given interface and, after swapping the source and destination L2 addresses of the packet, redirects it to a second interface. We installed the application into a server equipped with an Intel Xeon Gold 5120 14-cores CPU @2.20GHz (hyper-threading disabled) 19.25 MB of L3 cache and two 32GB RAM modules. We used Pktgen-DPDK to generate 64-bytes UDP packets from another server and to count the received packets; in fact, each server has a dual-port Intel XL710 40Gbps NIC, directly connected to the corresponding one of the other server. Both servers run Ubuntu 18.04.1 LTS, with the DUT running kernel 5.1.3 and the eBPF JIT flag enabled.

TABLE II: Comparison between vanilla-eBPF applications and a Polycube network function.

Application	Through.	LoC (FP)	LoC (S/CP)
xdp_redirect	4.7Mpps	64	176
tc_redirect	1.48Mpps	53	56
pcn-simplefw (XDP)	4.0Mpps	17	0
pcn-simplefw (TC)	1.31Mpps	17	0

Results. Table II shows a comparison between two very simple vanilla eBPF applications and a Polycube NF that performs the same operations, attached to either XDP or Traffic Control (TC) hooks. As we can notice, Polycube introduces a small (fixed) overhead to every NF, which is required to provide the abstractions mentioned before (e.g., virtual ports); in fact, this requires additional processing before and after calling the fast path of the NF, which is totally hidden to the NF developer. As result, the number of LoC for both the fast-path (FP) and the slow and control path (S/CP) is considerably reduced, allowing the developer to focus on the core logic of the program and leaving the common tasks and the possible optimizations to the Polycube daemon. Note also that the sample vanilla applications that we are taking into account are extremely simple; for more complex applications, a developer using vanilla-eBPF has to implement, for example, the entire fast-slow path interaction, which requires a non-negligible amount of effort.

B. Use case

In this subsection, we present a real world use case that can be implemented within Polycube and the type of performance improvements that we can expect.

1) K8s Network Provider: To demonstrate the capability of Polycube to facilitate the creation of complex applications

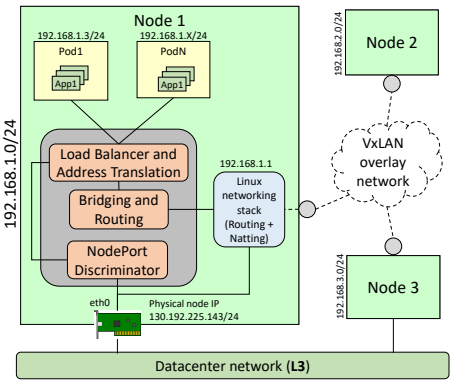

Fig. 2: Architecture of the Polycube K8s plugin.

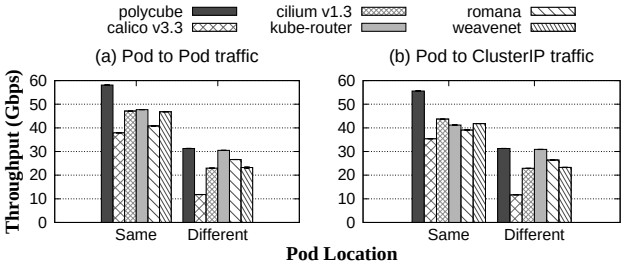

Fig. 3: Performance of different k8s network providers.

created by chaining different network functions together, we implemented an CNI plugin for Kubernetes [22], one of the most important open source orchestration system for containerized applications. Figure 2 shows the resulting architecture that is composed of different independent Polycube services that are chained together to support the main operations required by k8s network plugin interface. Tests were carried out on a 3-node cluster, a master and two workers with Linux kernel v4.15, Intel Xeon CPU E3-1245v5 @3.50GHz with dual-port Intel XL710 40Gbps NIC cards connected point-to-point. TCP throughput was measured with `iperf3` with default parameters; the server was always running in a Pod, while the client was either in a physical machine or in another Pod depending on the test.

Results. In Figure 3 we asses the performance of our Polycube network provider comparing it with the other existing solutions. In particular, we consider the Pod-to-Pod[1] connectivity and the Pod-to-ClusterIP[2] connectivity. Results show that the Polycube k8s plugin reaches 15-20% higher throughput than other solutions in the case server and client are on the same node. When pods are on different nodes, the advantage of the plugin becomes less evident, but still better than other solutions.

Note: Although our k8s plugin achieves better performance than the others in the two cases under consideration, it is not comparable in terms of functionality with the existing solutions, which are both more stable and complete. The purpose here is to demonstrate the generality of the Polycube programming model and the performance benefits that can be obtained from eBPF-based NFs.

VI. Conclusions

This paper presents the design and architecture of Polycube, a framework to build network services that can be dynamically instantiated and loaded into the Linux kernel at runtime. Polycube uses a model-driven abstraction to express the NF behavior and syntax, and provides additionally a set of abstractions and APIs to simplify the development of chained network functions. We demonstrate Polycube in practice, by demonstrating the performance benefits when creating complex network services such as a network provider for Kubernetes.

[1]A Pod is the smallest manageable unit in a k8s cluster and is composed of a group of one or more containers sharing the same network.

[2]A ClusterIP is a type of service that is only accessible within a Kubernetes cluster through a *virtual* IP. When a Pod communicates with this *virtual* IP, the request can be mapped to an arbitrary Pod running within the same physical host or into another one.

Acknowledgment

We would like to thank the many people who contributed to this work and the anonymous reviewers for their thoughtful feedbacks. Our thanks also to the European Commission (project ASTRID, Grant Agreement no. 786922), which partially founded this project.

References

[1] ETSI. (2017, feb) Network functions virtualization. [Online]. Available: http://www.etsi.org/technologies-clusters/technologies/nfv

[2] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15. USENIX Association, 2015, pp. 117–130.

[3] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: a framework for nfv applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 121–136.

[4] M. V. Bernal, I. Cerrato, F. Risso, and D. Verbeiren, "A transparent highway for inter-virtual network function communication with open vswitch," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 603–604.

[5] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: high performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, 2015.

[6] L. Rizzo and G. Lettieri, "Vale, a switched ethernet for virtual machines," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012, pp. 61–72.

[7] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[8] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo *et al.*, "Rollback-recovery for middleboxes," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 227–240.

[9] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 163–174.

[10] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using sdn," in *ACM SIGCOMM computer communication review*, vol. 43, no. 4. ACM, 2013, pp. 27–38.

[11] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul, "Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 19–24.

[12] P. Authors. (2019, January) Polycube: ebpf/xdp-based software framework for fast network services running in the linux kernel. [Online; last-retrieved 22-July-2019]. [Online]. Available: https://github.com/polycube-network/polycube

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

[14] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: a highly scalable user-level {TCP} stack for multicore systems," in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014, pp. 489–502.

[15] Z. Ahmed, M. H. Alizai, and A. A. Syed, "Inkev: In-kernel distributed network virtualization for dcn," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 3, 2016.

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

[17] C. Authors. (2018, jul) Bpf and xdp reference guide. [Online]. Available: https://cilium.readthedocs.io/en/latest/bpf/

[18] S. Miano, M. Bertrone, F. Risso, M. Vásquez Bernal, and M. Tumolo, "Creating complex network service with ebpf: Experience and lessons learned," in *Proceedings of the IEEE High Performance Switching and Routing (HPSR)*. ACM, 2018, pp. 1–8.

[19] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[20] M. Bjorklund, "The yang 1.1 data modeling language," 2016.

[21] IOVisor. Bpf compiler collection (bcc). [Online]. Available: https://github.com/iovisor/bcc/

[22] G. Inc. (2019, July) Kubernetes: Production-grade container orchestration. [Online; last-retrieved 22-July-2019]. [Online]. Available: https://kubernetes.io/