A Micro-service Approach for Cloud-Native Network Services

Sebastiano Miano, Fulvio Risso
Dept. of Control and Computer Engineering
Politecnico di Torino, Italy

CCS CONCEPTS

• **Networks → Network architectures**; **Cloud computing**; *Programmable networks*; *Network management*.

ACM Reference Format:
Sebastiano Miano, Fulvio Risso. 2020. A Micro-service Approach for Cloud-Native Network Services. In *Proceedings of Symposium on SDN Research 2020 (SOSR '20).* ACM, New York, NY, USA, 3 pages.

1 INTRODUCTION

Recently, there has been a visible shift in the paradigms used to develop and deploy (previously monolithic) server applications in favor of micro-services. Cloud-native technologies are used to develop applications built with services packaged in containers, deployed as micro-services and managed on elastic infrastructure through agile DevOps processes and continuous delivery workflows. This new paradigm has brought a visible change in the type and requirements of network functionalities deployed across the data center given the new type of workloads and applications running on the servers. Cloud-native platforms, like Kubernetes, relies on different network providers (a.k.a., network plugins) to implement the underlying data plane functionalities and transparently steer packets between the micro-services.

Current alternatives to build such software network functions rely mostly on kernel bypass approaches, implementing all the network functionality in user-space in a "busy waiting" loop. Although these approaches bring unquestionable performance improvements, they may not be suitable for this kind of paradigm given their intrinsic characteristics. In fact, they *(i)* require the exclusive allocation of resources (i.e., CPU cores) to achieve very good performance; this is perfectly fine when we have a single dedicated machine for the networking purposes but it becomes overwhelming when this cost has to be payed for every server in the cluster, since they permanently steal precious CPU cycles to other application tasks. Moreover, they *(ii)* require to re-implement the entire network stack in userspace, hence losing all the well-tested configuration, deployment and management tools developed over the years within the operating system.

As consequence, most of existing cloud-native network providers today still rely on functionalities and tools embedded into the operating system network stack. The drawbacks of this approach are also evident; first of all, kernel network applications are notoriously slow and inefficient given their generality, which impairs the possibility to specialize the software network function depending on workloads or the type of application that is running on top of it. Secondly, software network functions that live in the kernel have also proven hard to evolve due the complexity of the code and the difficulties in maintaining, up-streaming or modifying the kernel code (or the respective kernel modules).

In this demo, we show **Polycube**, an overarching coherent software architecture that solves the previous mentioned problems by applying the micro-service paradigm to the world of in-kernel network functions, enabling the creation of efficient, modular and dynamically reconfigurable networking components, available with vanilla Linux. Each Polycube service can be dynamically plugged into the framework and configured through a set of `REST` APIs that are used to perform the typical CRUD (create-read-update-delete) operations on the service itself. The corresponding demo video is available at **https://youtu.be/gW2uByayYxY**.

2 ARCHITECTURE

Each Polycube service is composed of a *control plane*, which is executed in userspace and is in charge of the service configuration and other non-dataplane tasks (e.g., routing protocols), and a *data plane*, which is executed in the kernel context and triggered every time a new packet is received in the given part of the Linux networking stack where the program is attached to. The data plane exploits the recently added eBPF [1] subsystem of the Linux kernel to enable custom user-defined programs to be executed in the kernel and attached to different points of the TCP/IP stack (e.g., XDP [2]), guaranteeing the safety (i.e., eBPF programs cannot crash the kernel) trough an in-kernel verifier and allowing to inject those programs in the kernel at runtime, without having to install additional kernel modules or restarting the machine.

A Polycube service chain involves of a set of network function instances that are connected to each other through virtual ports, which are in turn peered with a Linux networking device or another NF instance. In the standard model, eBPF programs do not have the concept of port from which traffic is received or sent out. Polycube enables this abstraction by adding a set of additional eBPF components, which are hidden to the developers.

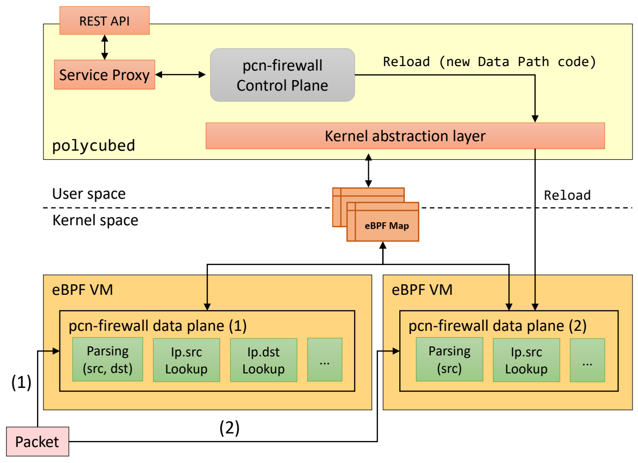

Figure 1: Re-configurable pipeline of the Polycube `pcn-firewall` service.

We will show how Polycube services can be chained together providing a highly customized path in the kernel that is optimized for the specific application context (e.g., a Kubernetes CNI plugin), while still being able to communicate and interact with the rest of the TCP/IP stack, hence avoiding to re-implement or throw away existing functionalities embedded within the operating system.

3 DEMONSTRATION

3.1 Agile Service Development

Internally, the data plane architecture of every Polycube service can be composed of different micro-functional blocks, which are instantiated as a set of different eBPF programs connected together through *tail calls*[1]. This set of different micro components, handled by a unique control plane, is particularly useful because it allows the developer to handle each feature separately, enabling the creation of loosely coupled services with different functionalities such as packet parsing, classification or field modification.

Figure 1 illustrates the internal structure of the simplified version of the Polycube `pcn-firewall` service, which is able to filter packets depending on the source and destination IP. The first version (1) of the data plane, which is instantiated by the corresponding control plane component when the service is created, is composed of several micro-blocks; a parser module that extracts the fields from the packets and two additional blocks that check if the source and destination IP of the packets need to be filtered. At runtime, the service administrator may change the type of rules deployed in the firewall service, for instance by eliminating all the rules matching on the destination IP. At this point, the component matching on the destination IP becomes useless, together with the parsing module that does not need to extract also that field from the packet. The control plane of the firewall NF then removes and substitutes the above-mentioned program and reloads a new version of the data path (2); Polycube will take care of safely perform this reloading, thus avoiding any service disruption.

This capability to dynamically compose and replace those micro-blocks within the NF allows the creation, at runtime, of an optimal version of the original NF data plane, which is customized depending on the application needs.

[1]Tail calls are an eBPF mechanism that allows to perform a "long jump" from one eBPF program to another.

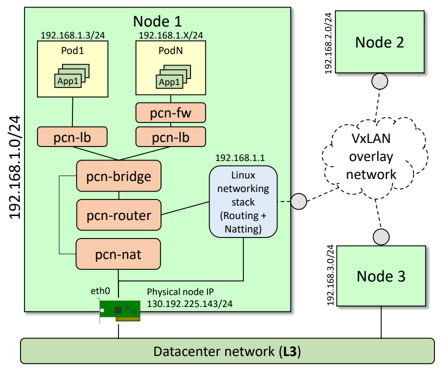

Figure 2: Architecture of the Polycube K8s CNI plugin

3.2 Kubernetes CNI Plugin

In the second part of the demo, we demonstrate the capability of Polycube to enable the creation of complex applications created by chaining different network functions together, which were already available as standalone components. A novel K8s-specific control plane integrates the above components to provide exactly the network service required by this orchestrator, namely pod-to-pod[2] and pod-to-service communication, providing also support for special K8s services (namely, ClusterIP and NodePort), and security policies. Figure 2 shows the resulting architecture, which is composed of different independent Polycube services that are chained together to support the main operations required by K8s network plugin interface. Our implementation, albeit prototypal, required a very limited amount of work and supports all the required communication patterns through a simple concatenation of existing Polycube NFs, which can be dynamically deployed, modified or removed depending on the application needs and can easily interact with the rest of the TCP/IP stack to exploit existing functionalities (e.g., tunneling).

[2]A K8s pod is a group of containers that are deployed on the same host.

REFERENCES

[1] Cilium Authors. 2018. *BPF and XDP Reference Guide*. https://cilium.readthedocs.io/en/latest/bpf/

[2] Høiland-Jørgensen T. et al. 2018. 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*. ACM.