An Overview of Cloud-Native Networks Design and Testing

Zhaobo Zhang, Xinli Gu
Silicon Valley Network Technology Lab.
Futurewei Technologies Inc.
Santa Clara, CA, USA
e-mail: zzhang1@futurewei.com, xgu@futurewei.com

***Abstract*—Cloud-native patterns have reshaped application development over the past decade. With the benefits of agility, resiliency, and scalability, the network domain starts embracing the cloud-native patterns to accelerate its evolution. Containerization becomes another solution of network function virtualization. Leveraging existing network services and the mature container orchestration platform, cloud-native networks attract wide attention, however the performance and scalability challenges in design and testing arise as the architecture advances. This paper presents an overview of cloud-native networks, the design and testing challenges and the development activities from open-source communities towards overcoming those issues. Performance optimization and hardware and software co-design are critical for the future success of cloud-native networks.**

Keywords—Cloud-native; Cloud-native network functions; container; Kubernetes; continuous testing; performance testing.

I. INTRODUCTION

Since Amazon first launched cloud computing platforms, delivering compute and storage resources through the Internet in 2006, on-demand and scalable cloud infrastructure has overwhelmingly reshaped the development of software and business [1]. Application architecture shifts from monoliths to microservices. Combining microservices with containerization and Continuous Integration and Continuous Delivery (CI/CD), the cloud-native concept emerged around 2010. As one of the pioneers, Netflix redesigned their systems in a cloud-native way and migrated all the services and data to the cloud through a seven-year journey, which facilitates rapid product release, new resource-hungry features and ever-growing volumes of data [2].

With proven success, cloud-native becomes a modern way of developing software. In 2015, Cloud Native Computing Foundation (CNCF) [3], a Linux Foundation project, was founded to advance container technology and align industry practice around its evolution. Since then, the cloud-native technologies and tools have thrived and taken great strides. Kubernetes [4], a container orchestration platform for automated container deployment, scaling and management is the first CNCF project. The plugin-based design and high extensibility build its success and make it to be the most adopted container orchestration system. Along with orchestration, a configurable infrastructure layer called service mesh is designed to ensure the security, resiliency, and observability of the communications between services. These two key components pave the way for container deployment and runtime management and significantly accelerate the cloud-native patterns adoption. In addition, CNCF launched many other projects covering different perspectives, including continuous integration and delivery, container runtime, cloud-native network, etc.

In the 5G and cloud era, communication service providers seek solutions to advance networks to meet ever-changing customer needs, optimize network utilization, and support new application scenarios, e.g., augmented reality, virtual reality, Internet of things. Cloud-native principles are meant to increase the velocity of the business. With API enabled design, CI/CD and Development and Operations (DevOps) practices, the cloud-native technologies improve the service agility and time-to-market. Therefore, network equipment vendors and communication service providers start adopting cloud-native architecture, containerizing network functions, more importantly, leveraging open-source cloud-native tools to modernize networks, e.g., orchestration, automation, monitoring. Together with application, network development joins the cloud-native journey. Milestones are illustrated in Figure 1.

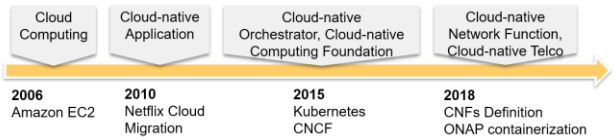

Figure 1. Cloud-native Journey from Applications to Networks.

This paper aims to provide an overview of the current landscape of cloud-native networks, with focus on contributions from open-source communities. The definition and reference architecture of cloud-native networks are first introduced in Section II. The challenges and network specific requirements are discussed in Section III. Good design practice and guidance are summarized in Section IV. Testing flow and performance testing are presented in Section V. Conclusions are presented at the end.

II. CLOUD-NATIVE NETWORKS

Network architecture has evolved from individual physical machines for each Physical Network Functions (PNFs), to Virtual Network Functions (VNFs) running on VMware or OpenStack, to what the CNCF sees as the next wave of Cloud-native Network Functions (CNFs) running on Kubernetes. CNFs are like VNFs, but they run on lighter weight containers, simpler to upgrade, easier to secure, and cheaper to operate.

Cloud-native networks can be understood from two perspectives. The first is networks are built with cloud-native principles, which means network functions are containerized, with both control and data plane composed of microservices. The other is that networks are to provide connectivity and security to cloud-native applications in a cloud environment. Therefore, the network itself and the workloads that it serves both are considerably evolved. However, considering of the existing infrastructure, the compatibility with the PNFs and VNFs is needed in some scenarios.

As Kubernetes is the most adopted container orchestration platform, the cloud-native networks discussed in the remainder of the paper are in the context of Kubernetes. Kubernetes networking is based on a plugin model, which is open to third-party implementations. A network plugin needs to provide connectivity and reachability in pod networking. A pod is a group of containers that are deployed together on the same host in Kubernetes. Each pod has a unique and dynamic IP. All the pods in a cluster are connected through a flat network. Project Container Networking Interface (CNI) defines the standards on how network plugins should look, and how container runtime should invoke them [5]. It also provides a set of basic plugins as reference.

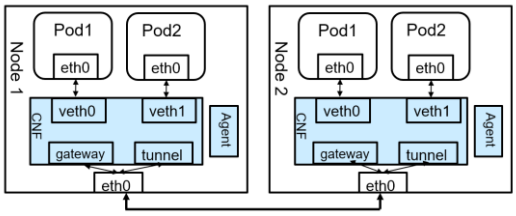

Figure 2. Container Networking Block Diagram.

In Figure 2, a simplified network block diagram is shown to illustrate the communication between pods across two diffident nodes. A CNF module first builds the connectivity between the pods and host network, and then it creates the overlay network between hosts based on different protocols, e.g., VXLAN or IPIP. This CNF can be implemented either in a kernel bypass manner to improve performance or with Linux kernel networking stack for the sake of simplicity. Together with this CNF, an agent pod is typically used for routes and network policy configuration. Project Flannel [6] and Calico [7] are the two commonly used CNI solutions.

III. NETWORK SPECIFIC DESIGN

Network workloads, responsible for low-level traffic forwarding, are different from the generic application workloads running in the cloud. A containerized network function may require multiple interfaces, faster data pipeline, comprehensive network policies, etc. In this section, the network specific requirements and solutions are discussed.

A. *Multiple Networks Attachment*

When Kubernetes initiate a pod, only one interface is created by default. In order to provide multiple interfaces, a CNCF network plumbing working group was formed, and a meta-plugin solution was proposed to create multiple network interfaces and manage multi-network policy. An illustration is shown in Figure 3. Compared to one standard CNI, multiple CNI plugins can be chained to form a meta-plugin. Then, multiple networks can be attached to a single pod. Project Multus [8] and CNI-Genie [9] provide reference implementations.

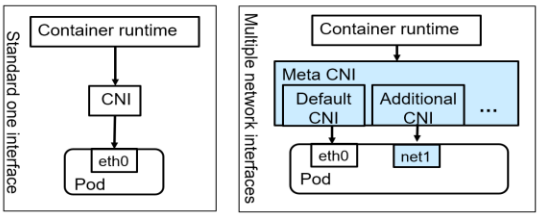

Figure 3. Standard vs Multiple Network Interfaces Attachment.

B. *Host Networking Performance Improvement*

CNI often leverages Linux host networking to implement network functions and policy. For example, iptables, a user-space utility program, is used to configure the IP packet filter rules. The filters are organized in different tables of chains to treat packets with specific rules. However, it becomes a bottleneck when large numbers of pods are under orchestration, since each host needs updates if any pod changes in the cluster.

An alternative of using iptables is implementing the function with extended Berkeley Packet Filter (eBPF), a Linux kernel technology, which compiles user programs to bytecode and attached to the kernel to be more performant [10]. eBPF enables the dynamic insertion of security, visibility, and networking control logic to the kernel. The flow is illustrated in Figure 4. The ability to run user-supplied programs inside the Linux kernel makes eBPF a powerful tool in terms of performance and convenience. Project Cilium is an eBPF-based CNI [11]. Detailed workflow and performance improvement can be found on Cilium's blog [12].

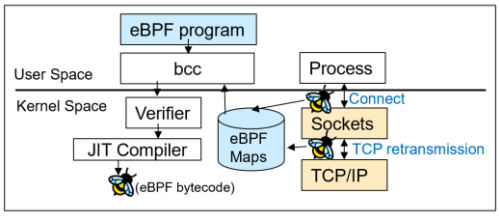

Figure 4. The Flow of eBPF Program Inserted to Linux Kernel.

C. *Data Plane Acceleration*

When a packet goes from user space to kernel space, an expensive copy occurs. To avoid the copy overhead, DPDK is widely used to process packets in user space and directly interact with network hardware bypassing the Linux kernel [13][14]. A data path comparison is shown in Figure 5. To further improve the performance, a high-performance virtual switch, e.g., Open vSwitch (OVS) can be added too. Project Antrea implemented OVS based CNI. With offloading the OVS function to supported Network Interface Card (NIC),

the network bandwidth is increased by more than 3 times [15]. In order to provide high performance computing and networking in hyperscale data centers, hardware acceleration moves beyond CPUs and turn to dedicated chips [16]. Fortunately, Kubernetes provides a device plugin framework to allow specific hardware in the cluster, e.g., Graphic Processing Unit (GPU), NIC, which provides more possibilities for hardware acceleration.

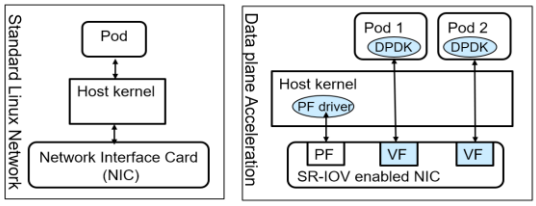

Figure 5. Data Plane Acceleration Bypass Linux Kernel.

D. Hybrid Multi-cloud Networking Orchestration

So far, the networks discussed above are intra-cluster networks, i.e., the communication is within the same cluster. However, as multi-cloud and hybrid cloud become more prevalent in today's business model, the inter-cluster network becomes a critical problem.

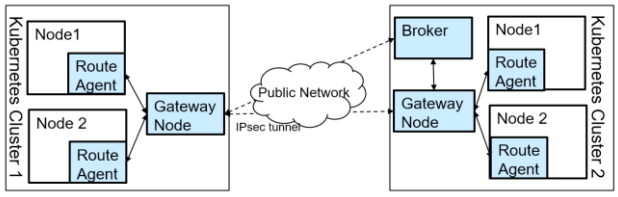

Figure 6. Multi-cluster Networking.

To connect two different clusters, traffic typically goes through the public Internet. An IPsec tunnel is often the choice to ensure secure communication. Figure 6 shows a simplified architecture of multi-cluster networking. A broker is used to exchange the information between clusters, and a gateway node is responsible for establishing IPsec tunnels and updating local cluster information to the central broker. Route agent runs on each node to configure the routes and rules. Project Submariner is a reference solution for this architecture [17]. For more comprehensive networking features, Project Network Service Mesh (NSM) [18] and Tungsten Fabric [19] can be referred.

IV. Design Principles

According to Sections II and III, the cloud-native networks typically consist of agents on each node to forward traffic and implement policies, a centralized control module to communicate with the container runtime and agents, and a data store to keep configurations and states. To design such a system with cloud-native principles, the following guidance is summarized from the best practices.

- *Modularization*

Each network function should be packed in its own container and orchestrated in a dynamic way. Complex network functions can be created by service function chaining. Service dependency can be programmed through a Helm chart in a unified format. Kubernetes style API is recommended to allow unified control.

- *State Separation*

Network functions should be separated to stateless and stateful, in order to scale the stateless functions smoothly. The states of stateful functions can be stored in etcd, a distributed key-value store in Kubernetes.

- *Infrastructure as code*

Network resources should be managed with machine-readable files. All the changes should be documented into files. Therefore, tasks like provision and roll back can be easily automated. Compared to the traditional management with command-line interface, automation removes the risk associated with human error and decreases system downtime.

- *Low-Level Acceleration*

Dedicated chips and hardware components are essential to build future intelligent cloud infrastructure [13]. With Kubernetes's device plugin feature, hardware functions can be exposed to containers for performance improvement. The design of hardware APIs should be consistent and reusable.

- *Built-in Observability and Analytics (AI ready)*

The observability of CNFs should be considered during the design phase, in order to enable continuous monitoring and automated troubleshooting. Output formats should be standardized and compatible with existing monitoring tools like Prometheus [20] and Grafana [21]. Thus, full-stack performance monitoring and analytics, from infrastructure to application, can be supported. In addition, structured data make artificial intelligence easy to apply and pave the way to autonomous network.

- *Platform Agnostic*

The network services should be able to be deployed and orchestrated seamlessly among public cloud, private cloud, and edge cloud. The CNFs should require no changes under different platforms.

V. Testing Methodologies

Software testing today has been modernized by CI/CD and DevOps, two important characteristics of cloud-native patterns. Testing becomes a continuous activity in design, deployment, and operation. In this section, the generic test flow under CI/CD is first introduced, followed by performance testing. Lastly, the observability in CNFs is discussed.

CI is to establish a consistent and automated pipeline to build, package and test applications. With regularly checking new code, testing and integrating it with other parts of the system, organizations can reduce development and testing time from months down to days, even hours. Test suites are often written alongside new features. Unit tests ensure the committed code itself works. Integration tests ensure no breaks are introduced into the main code line. End-to-end tests ensure end user's experience by testing the entire product. Common CNF CI jobs provide the test coverage on command-line interface, authorization,

storage, connectivity, network policy, etc. Security scan and compliance tests are typically included as well.

CD automates the software delivery process. It ensures the verified code changes from development environments can be pushed into production seamlessly. An interesting feature brought by CI/CD is canary testing, which releases the new version of the software only to a small percentage of users, to perform in-production test. New versions can be easily rolled back with Kubernetes orchestration.

Besides functional testing, performance testing is critical for cloud-native networks. The performance requirements of cloud-native networks are twofold. One is the performance of a CNF alone, e.g., how many packets a CNF can process per second. The other is the scalability of handling large amounts of network requests from web scale services, e.g., how fast the CNFs can provision one thousand endpoints or update network policies on thousands of hosts. In order to enable organizations to reliably test and compare performance between VNFs and CNFs, CNCF launched the CNF Testbed project in 2019.

The CNF Testbed project targets to build a repeatable test environment by using immutable hardware, version control on all configurations including underlay networking, and bootstrapping workload repeatably with automation pipeline [22]. The test framework typically includes a Kubernetes cluster with CNFs under test, traffic generator, and underlay networks illustrated in Figure 7. Traffic can be generated either within the cluster or from an external generator. The test steps are listed in TABLE I. The performance metrics evaluated often include CNF deployment time, endpoints provisioning time, network policy update time, idle-time CPU and memory usage, runtime CPU and memory usage, network throughput and latency.

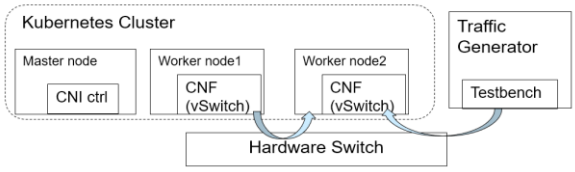

Figure 7. CNF Testbed Framework.

TABLE I. PERFORMANCE TESTING STEPS

1.	Provision hardware and Kubernetes cluster
2.	Deploy CNFs
3.	Deploy traffic generator
4.	Run the traffic benchmarks and tests
5.	Collect performance metrics

According to CNF Testbed's initial results, from VNFs to CNFs, the change will not affect the overall networking performance [22]. In fact, the lightness of container technology allows switching user context more quickly than with VM Hypervisors, and containerized workload could have a more direct interaction with underlying hardware. Communities are looking for more use cases to make more comprehensive comparison.

Since continuous testing and continuous monitoring become the norm today, the observability is critical. Observability includes tracing, metrics, and logs at various levels like cluster level, container level and kernel level. Kernel level tracing is particularly important for the CNFs. Standard Linux tracing tools like perf, ftrace, SysDig can be leveraged. To customize the network tracing, eBPF can be used to translate and load user programs to the kernel. Therefore, kernel networking events can be probed and monitored. Furthermore, the probes can be added into the CNFs program as well. Project IOVisor [23] implemented eBPF based monitoring tools, e.g., trace TCP passive and active connections, trace TCP packet drops with details, trace TCP retransmits. In a customized CNF, eBPF programs can be added to trace the changes of interface counters, interface address, routing tables and network address translation sessions, etc. It is an ongoing project to enrich eBPF-based monitoring tools. With more detailed and critical information extracted, fine-grained testing, fault isolation, and smart analytics are possible [24].

VI. CONCLUSIONS

Cloud-native principles and technologies bring tremendous benefits in terms of business agility, scalability and resiliency. Modern networks adopt this trend to accelerate development speed, improve resiliency with dynamic scaling and safe upgrades, and reduce costs. Kubernetes, a powerful production-grade orchestration platform with high extensibility, accelerates the process of network function containerization.

From PNFs to VNFs and CNFs, the implementation of network functions keeps evolving. There are advantages and issues for each paradigm. Although CNF brings many benefits, not all the workloads could fit perfectly for containers. Considering the performance advantages of network specific hardware, the data plane acceleration with hardware offloading cannot be neglect. This also brings new opportunities for next-generation hardware design. Network equipment and service providers could take a top-down approach, according to the requirements of containerized applications to do the hardware and software co-design, in order to achieve the optimal network solutions and meet the market needs in the cloud era.

REFERENCES

[1] R. Aljamal, A. El-Mousa and F. Jubair, "A User Perspective Overview of The Top Infrastructure as a Service and High Performance Computing Cloud Service Providers," IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology, 2019, pp. 244-249.

[2] M. Villamizar et al., "Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures," 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2016, pp. 179-182.

[3] CNCF, https://www.cncf.io/, [retrieved Oct. 2020].

[4] Kubernetes, https://kubernetes.io/, [retrieved Oct. 2020].

[5] CNI, https://github.com/containernetworking/cni, [retrieved Oct. 2020].

[6] Flannel, https://github.com/coreos/flannel, [retrieved Oct. 2020].

[7] Calico, https://www.projectcalico.org/, [retrieved Oct. 2020].

[8] Multus, https://github.com/intel/multus-cni, [retrieved Oct. 2020].

[9] CNI-Genie, https://github.com/cni-genie/CNI-Genie, [retrieved Oct. 2020].

[10] S. Miano, M. Bertrone, F. Risso, M. Tumolo and M. V. Bernal, "Creating Complex Network Services with eBPF: Experience and Lessons Learned," IEEE 19th International Conference on High Performance Switching and Routing (HPSR), 2018, pp. 1-8.

[11] Cilium, https://cilium.io/, [retrieved Oct. 2020].

[12] Cilium Performace, https://cilium.io/blog/2020/10/09/cilium-in-alibaba-cloud, [retrieved Oct. 2020]

[13] N. Pitaev, M. Falkner, A. Leivadeas, and I. Lambadaris, "Characterizing the performance of concurrent virtualized network functions with OVS-DPDK, FD.IO VPP and SR-IOV", in Proc. Of ACM International Conference on Performance Engineering, 2018, pp 285-292.

[14] L. Linguaglossa, et al., "Survey of Performance Acceleration Techniques for Network Function Virtualization," in Proc. of the IEEE, vol. 107, no. 4, pp. 746-764, 2019.

[15] Antrea, https://antrea.io/, [retrieved Oct. 2020].

[16] D. He, Z. Wang and J. Liu, "A Survey to Predict the Trend of AI-able Server Evolution in the Cloud," in IEEE Access, vol. 6, pp. 10591-10602.

[17] Submarine, https://submariner.io/, [retrieved Oct. 2020].

[18] NSM, https://networkservicemesh.io/, [retrieved Oct. 2020].

[19] Tungsten Fabric, https://tungsten.io/, [retrieved Oct. 2020].

[20] Prometheus, https://prometheus.io/, [retrieved Oct. 2020].

[21] Grafana, https://grafana.com/, [retrieved Oct. 2020].

[22] CNF Testbed, https://github.com/cncf/cnf-testbed, [retrieved Oct. 2020].

[23] IOVisor, https://www.iovisor.org/, [retrieved Oct. 2020].

[24] C. Cassagnes, L. Trestioreanu, C. Joly and R. State, "The rise of eBPF for non-intrusive performance monitoring," IEEE/IFIP Network Operations and Management Symposium, 2020, pp. 1-7.