Cloud Native Network Function Virtualization

True Cloud For NFV

A Technical Paper prepared for SCTE/ISBE by

Alon Bernstein
Distinguished Engineer
cisco systems
alonb@cisco.com

Table of Contents

Introduction

Network Function Virtualization (NFV) is a promising technology that enables the transition of network functions (e.g. routers, CMTS, firewalls etc) from dedicated hardware appliances to general purpose servers. The first wave of NFV was nicknamed "lift-and-shift" because it was about taking a network function and running it as-is in a virtual environment. However, the Cloud-Native architectures that the web giants are using take a different approach to virtualization. They view the cloud as a huge distributed compute platform where "functions" are broken into micro-services and scheduled in a container based environment onto the available compute/storage resources.

The web-native approach view networking applications as just another type of software application and this is indeed the correct view of networking in this framework. But the devil is in the details and out-of-the-box schedulers such as Kubernetes do not address the specific requirements of networking application, specifically when it comes to processing the data plane path.

This paper will highlight the differences between the old and new virtualization approaches, identify the gaps where the cloud native approach needs to be extended for networking and offer solutions. More specifically the presentation will compare and contrast the direction the industry takes with network function virtualization vs. cloud native.

This paper is high-level and outlines the main issues and comments of possible solutions without going into specifics.

What is "cloud native"

The Cloud Native Computing Foundation (https://www.cncf.io) defines cloud native this way:

Cloud native systems will have the following properties:

(a) Container packaged. Running applications and processes in software containers as an isolated unit of application deployment, and as a mechanism to achieve high levels of resource isolation. Improves overall developer experience, fosters code and component reuse and simplify operations for cloud native applications.
(b) Dynamically managed. Actively scheduled and actively managed by a central orchestrating process. Radically improve machine efficiency and resource utilization while reducing the cost associated with maintenance and operations.
(c) Micro-services oriented. Loosely coupled with dependencies explicitly described (e.g. through service endpoints). Significantly increase the overall agility and maintainability of applications. The foundation will shape the evolution of the technology to advance the state of the art for application management, and to make the technology ubiquitous and easily available through reliable interfaces.

It's the opinion of the author that there are some other guidelines that are associated with Cloud Native:

- Allow the software developer to focus as much as possible on the core business logic. Availability/scaling/upgrading are part of the services the infrastructure provides. Open source

software tools provide much of the support infrastructure needed, e.g. databases, software caches, buses etc.
- A certain style of application guidelines that are captured well in the "12 factor app" web site (https://12factor.net) - more details can be found in the "12-factor app" section later in the paper.
- Automation – the scale and complexity of cloud deployment require automation.

There is some confusion in the industry around containers and cloud native. The use of containers does not make a deployment "cloud native" because containers are nothing but a packaging option. Containers work well with a cloud native environment because they are lightweight and it's easy to have many of them, but the use of containers alone does not make a deployment cloud native.

Cloud Native vs. NFV Goals

The goals of cloud native as currently viewed are different then the stated goals of NFV, however, it is the opinion of the author that they will converge, and converge towards the cloud native direction.

- For NFV the goal was to run traditional network functions on general purpose server. The initial focus was the cost savings related to running a common hardware with common life cycle management. There was not much discussion on how to improve the network functions, it was all about placing it, as is, in a virtual machine.
- Cloud native is built to support massively parallel data centers used by the web giants. Its goal is the simplify software application writing by allowing the software engineer to focus on the core "value add" aka "business logic" and using the cloud infrastructure to deal with workload placement, availability, scale and software upgrades.

The key to consolidating these two goals is to understand that networking function is simply software. Because of historical reasons it was implemented on proprietary Operating Systems on embedded systems and with a single minded focus on performance, but functionally it's "just software" and the cloud native principals can be applied to it.

The following sections will discuss the 12-factor app and domain driven design. Note that introductory text on these disciplines tends to use examples that are either from the enterprise world (e.g. credit card processing) or the web world (e.g. connect a web frontend to a database), however even if the examples seem not relevant to networking the underlying principles are.

The 12-factor app

The 12-factor app (https://12factor.net) is a guideline on how to write a modern app that fits well in a cloud environment. Note that the 12-factor app talks about processes – not containers – but it's essentially the same thing. Even from an implementation point of view one can argue that a container is essentially a well-isolated process. The reader is encouraged to read all 12 guidelines; this paper will outline two of them:

- Statelessness (covered under "processes"): This is probably the most critical guideline for writing cloud native applications because it's key to enabling availability/scaling/upgradability. The concept is simple. All the "important" application state is saved in an external database so that if a process crashes it can recover its state from the database. "Unimportant" state is not saved in the database because no harm is done if it is lost. A networking example can be a cable modem registration. There is no need to save all the intermediate states of the registration (TFTP/DHCP/etc) in a database; if a registration processes crashes while a cable modem is trying to register then no significant harm is done (the modem will re-register). However, if a cable modem completed registration and it's passing data then all the state associated with it has to be stored.

- Scale-out (covered under "concurrency"): adding more processing capacity is a simple matter of adding more containers. Because of the stateless nature of the 12-factor apps and because there are no dependencies between the containers it's easy to scale.

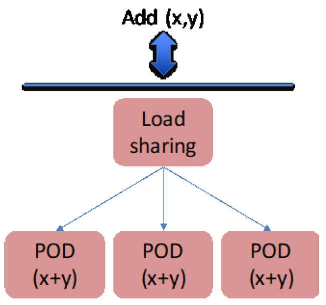
In some cases, it's not easy to apply the 12 factor app to a situation. A case in point is DOCSIS®. The protocol itself is not designed to be stateless. If we designed DOCSIS® today we would probably base it on REST exchange between the CM and CMTS, but the reality is that the current protocol is very stateful and very tangled. The way to address this is to isolate as much as can be isolated to stateless processes. In general, it is not easy to re-factor code to fit the 12 factor app framework and in many cases it requires a re-write.

Domain Driven Design

Breaking a complex function into "micro-services" is key to cloud-native systems. But where should be the demarcation points? When is a micro-service just the right size, not too big and not too small? A good methodology for defining a micro-services architecture is Domain Driven Design (DDD). The core idea in DDD is to focus on the intrinsic complexity and expertise related to a business outcome. This maps very well to the cloud native environment where (a) the developer is encouraged to use open source tools for anything that is not related to the core business outcome, and (b) specific domains of expertise can map directly into micro-services. DDD comes with its own terminology and can be rather heavyweight however the key concepts can be applied to the design without attempting to turn the whole team into DDD experts.

Cloud Native Services

The following example helps illustrate a very simple use case of how an application gets deployed in a cloud native environment. Assume it's a function that adds two numbers x,y and returns a result. This is a trivial example because its inherently stateless – the container that adds to two numbers has no state that has to be saved. Most practical micro-services require some external data store to save state, however this example is sufficient to illustrate the key concepts in cloud native services:



For our example let's assume Kubernetes is the cloud native scheduler. The basic scheduling unit in Kubernetes is a POD and although a POD can contain several containers it's typical for one POD to include only one container. In our example the function x+y maps to a single micro-service which maps to a single container which in turn maps to a single POD. We publish the x+y function as a service to the outside world. It is fronted by a load sharing function, which serves also as a service gateway so that the external users do not have to care how many PODs are used to run the service. And here is where the advantages of cloud native start showing up:

1. Scaling: as the load of requests to add x+y increase the scheduler (Kubernetes in this example) can add more instances of the x+y container and the load sharing module would utilize them, all that while still exposing only a single API externally.
2. Availability: If any of the containers crashes the fault is localized to that container. A new container can be quickly started and the impact of the crash is minimal. But what about the load sharing function itself? Could it be a single point of failure? The way load sharing is protected is typically done by clustering; a set of 3 or 5 load sharing functions that back up each other. The load sharing approach is also the source of the terms "pets vs. cattle" where traditional virtual machines (VMs) fit the pets metaphor – they require a lot of specialized care. Containers in a cloud are counted as cattle, easy to replace if one goes away.
3. Software upgrade: let's assume we have a better version of our x+y micro-service. With cloud native we can do in-production upgrades. The new version can be loaded alongside the existing containers; as more and more new containers get deployed the old ones can be de-commissioned until only the new version is running. Note that with this method there is no service interruption while the software is being upgraded.

As can be seen from the examples above, the front-ending of a service with a load sharing engine that doubles as an API gateway allows for scaling/availability and upgrade models which are very different to those used in traditional networking. For example, a traditional networking way to handle availability is to have a standby-active configuration with state communicated between the active and standby machines. Even in NFV the tendency was to do the same but with virtual machines. In cloud native it's more of an active-active configuration. The state is not communicated between the active and standby machines; instead it's written to a database that a new container can read in order to re-construct its state as it comes online.

Data and Control/Management Planes

When trying to apply cloud native paradigms to NFV, it's obvious that there are two separate domains that need to be addressed:

- Control and Management plane: The control and management planes deal with configuration and session establishments. Even though they are not typical web applications, meaning they are not HTTP/REST transactions, they are essentially workflow and transaction based systems only implemented with somewhat antiquated tools.
- Data plane: The data plane deals with the actual packet forwarding. This is where NFV differs from cloud native because it's not a transaction-based system.

The following sections will outline what it takes to support data and control/management planes in a cloud native framework.

Cloud Native Control/Management Plane

Creating a cloud native control/management plane for networking is fairly straight forward. The cloud native gateway, which is typically HTTP based, has to be replaced with a custom gateway that is based on whichever networking protocol is used (e.g. DOCSIS®). Writing a custom load balancer and integrating it into a cloud native scheduler is fairly straight forward.

The tricky part is writing a control plane application that adheres to the 12-factor app guidelines. This is because old communication protocols tend to be tangled. For example, see the "MAC Domain Descriptor" in DOCSIS®. It's basically a scratchpad for anything from IPv6 to low level PHY settings to controlling alarms. The key is to break out the code as much as possible to stateless and independent micro-services with the understanding that at some point they all get combined. If the micro-service that handles the combination is minimal then the integrity of the system is not harmed.

The other complexity in the control plane is related to maintaining database consistency or eventual consistency; this is also the hardest problem to solve in cloud native in general because it is a large distributed system. The recommendation is to go for eventual consistency as it's easier to code and maintain, but depending on the specific protocol requirements (remember, those might be protocols that were not designed for eventual consistency) it may or may not be possible.

Cloud Native Data Plane

The cloud native data plane is where NFV breaks new ground in cloud native architectures. This is because the transitional service model of cloud native will not work well. For business transactions, or protocol transactions it's ok to place a load balancer in front of the service because of all the benefits we have shown it brings in terms of service scaling etc. However, a load balancer will have a significant impact on performance if used for the data plane. For example, imagine it takes 10 usec to load balance a packet. For control plane applications where the processing can take a millisecond this is negligible, but if we forward 1Gbps of packets the load balancing almost doubles the compute resources needed. Still, the cloud native disciplines can still be used for the data plane as well; the magic is to use controllers and orchestrators to direct the packet streams to the right data plane containers as part of the normal routing/switching that is needed anyhow. In other words, the load balancing gets absorbed by networking and so net-net it does not cost anything while at the same time we can maintain all the benefits of cloud native.

Another issue that needs to be addressed with cloud native NFV is the issue of network resources. Traditionally a scheduler, such as Kubernetes, looks at CPU/memory as resources and makes placement decision based on their level of usage. Network utilization is not counted as a resource. A parallel system may have to support network resource based placement decisions.

Deployment and DevOps

Cloud Native is a natural fit for DevOps because of the ease of promoting new changes into production:

1. Because of the breakup to “micro-services” the impact of a software change is isolated, smaller and less risky, therefor enabling more frequent update then with traditional monolithic software which is contained in a VM.
2. As explained in the “cloud native services” section it’s easy to phase software into in-production systems because it does not cause a service interruption.

It’s clear that with the cloud native approach the service provider can break away from a lengthy qualification cycles and images that stay in the field for years, because operators would rather deal with the issues they know then risk being exposed to new ones in a new software release.

Tools such as Spinnaker (https://www.spinnaker.io) are helpful in managing the pipeline of promoting a code change to production, making it even easier to automate deployments and reap the benefits of frequent updates.

The discussion about DevOps points out to another key aspect of cloud native. It’s more than a technology or a way to write software. To take advantage of cloud native the organization itself has to evolve and adopt to new ways of software deployment, testing and monitoring.

Conclusion

The same principles that made cloud native such a huge success can be summarized at a high level as:

1. The basic ABCs of software engineering: break a complicated system into small and well defined functions. We call it “micro services” today; it might be called some other fancy name tomorrow, but it’s just common sense.
2. The basic ABCs of parallel computing as the cloud can be viewed as a huge parallel computer.
3. The basic ABC of managing a distributed system: that’s the complicated stuff in cloud and cloud native.

“Lift and shift” architectures brought certain benefits of virtualization, mostly related to the fact that the hardware can be shared between different functions. However, they did not solve issues related to scaling, feature velocity and availability because the code was still a monolith captured in a VM. Cloud native (which encompasses the three disciplines mentioned above) is a re-design of the network functions themselves.

Current cloud native systems are focused on managing enterprise and web commerce, not packet forwarding. The purpose of this paper is to show that all the issues related to building a cloud native NFVs are solvable problems. The design principals that made cloud native a success can be applied to network function virtualization as well, driving the same service velocity, availability and scaling to the networking world.

A reader that wishes to learn more about cloud native is encouraged to seek a wealth of information in cncf.io as well as the CNCF channel on Youtube. Of course, the best way to learn these new software paradigms is to code and use them.

Abbreviations

API	Application Programming Interface
CM	Cable Modem
CMTS	Cable Modem Termination System
CNCF	Cloud Native Computing Foundation
CPU	Central Processing Unit
DDD	Domain Driven Design
DevOps	Development and Operations
DHCP	Dynamic Host Configuration Protocol
DOCSIS®	Data Over Cable Service Interface Specification
HTTP	Hypertext Transfer Protocol
NFV	Network Function Virtualization
REST	Representational State Transfer
TFTP	Trivial File Transfer Protocol
VM	Virtual Machine

References

Cloud Native Computing Foundation, https://www.cncf.io/

The Twelve-Factor App, https://12factor.net

Kubernetes (container deployment, scaling, and management system), https://kubernetes.io/

Spinnaker (example of a cloud-centric software update platform), https://www.spinnaker.io/

DOCSIS® 3.1 MAC and Upper Layer Protocols Interface Specification (MULPI), https://apps.cablelabs.com/specification/CM-SP-MULPIv3.1

Randy Bias, "Architectures for open and scalable clouds", (slide 20 is the source of the cattle vs pets metaphor), https://www.slideshare.net/randybias/architectures-for-open-and-scalable-clouds