metaswitch

THE APPLICATION OF CLOUD NATIVE DESIGN PRINCIPLES TO NETWORK FUNCTIONS VIRTUALIZATION

Martin Taylor. CTO, Metaswitch Networks

Network Functions Virtualization is at the heart of the most fundamental transformation ever undertaken by the communications services industry, with profound impacts on technology planning, network engineering, operations and procurement. And despite the enormous upheaval that NFV brings to their business, virtually every network operator in the world today accepts its vital importance to their future, and many have already made great progress in their transformation to a software-centric future.

INTRODUCTION

The approach to NFV taken by the majority of network operators appears to be driven mainly from a bottom-up, technology-oriented perspective. This narrative starts with the observation that most network functions can be provided by software running on commercial off-the-shelf server hardware at considerably lower cost than traditional proprietary purpose-built hardware. The modern way to deploy software is with virtualization, and for virtualization at scale we need a cloud environment. If we are deploying software in a cloud environment, then we can automate operations, and save ourselves a ton of operational expense. And because the service is implemented entirely in software, we should be able to innovate more rapidly than we could before and drive new service revenues.

But there is another perspective on NFV, which is a services-oriented top-down view. This perspective can most clearly be illustrated by the following question: modestly funded start-ups with a few tens of engineers and very limited marketing budgets are building software systems in the cloud that deliver rapidly-evolving and fast-scaling services that are attracting tens or hundreds of millions of users and taking big bites out of traditional network operator revenues (e.g. WhatsApp) – how on earth are they able to do that?

This white paper is about the software techniques that the industry needs to adopt to do NFV right. NFV done right will transform the economics of service delivery, simplify the integration and deployment of new service capabilities, accelerate the creation and progressive enhancement of new services, and enable services to be delivered effectively at any scale. Not done right, NFV could fail spectacularly to deliver any return on a very large investment.

A BRIEF HISTORY OF VIRTUALIZATION AND CLOUD

Virtualization has a long history in the computer industry, and first became a mainstream commercial technology in the mid-1960s on IBM mainframes. The modern era of virtualization was ushered in by the addition of hardware support for virtualization on x86 processors by Intel in 2005-06, which paved the way for the introduction of successful hypervisor products such as VMware. Up to this point, IT shops had installed a separate physical machine for each different server application that they deployed, with the result that most machines were severely under-utilized. With a hypervisor they could safely deploy multiple server applications per host, consolidating their resources and achieving very substantial Capex and Opex savings.

At that time, it was common to see a mix of many different operating systems in use for IT applications: various flavors of Unix, Solaris, Windows and early versions of Linux. Naturally, there was a requirement to be able to mix applications with different operating systems on the same physical host. So hypervisors were designed to expose to applications an emulation of a complete physical x86 server: a virtual machine. The server application, together with the operating system it depends on, runs inside the virtual machine, safely and securely partitioned from other server applications and their supporting guest operating systems deployed on the same host.

The business case for virtualization was a compelling one, and by 2013 more than half of all IT workloads were running virtualized. IT shops began to view their physical servers not so much as a collection of individual machines, but more as a pool of computing resources. When they deploy a server application, they don't much care which particular machine it runs on, so long as it has sufficient resources to perform as required. This is led to the introduction of infrastructure software solutions that treat a collection of x86 machines as an interchangeable pool of resources, and manages the deployment of applications on it: a cloud.

Cloud technology enables compute resources to be treated as a utility, and this opens up the possibility of a market in which compute power can be bought and sold: the public cloud. Economies of scale mean that very large providers of public cloud services can offer compute power at considerably lower cost than can be achieved in small-scale private clouds. As a result, some IT shops now choose to deploy some or all of their applications on public cloud services.

For most of the first decade of cloud technology, the great majority of applications deployed in both public and private clouds were originally written to run on dedicated, bare metal servers. Cloud services offering virtual machines that emulate physical servers, so called Infrastructure as a Service, provide an ideal environment into which such applications can be moved.

THE EMERGENCE OF CLOUD NATIVE

The availability of inexpensive pay-as-you-go compute power in large-scale public clouds opened up a completely new kind of opportunity for entrepreneurs: the ability to create network-based services that could be offered to the public at scale, particularly in the realms of social media, messaging and media distribution. In particular, it massively reduced the amount of capital risk associated with starting up and scaling such services. The new ventures that set out to take advantage of this opportunity were not writing software to run on dedicated servers, and then deploying it on virtual machines in the cloud. Instead, they viewed the cloud as an entirely new kind of distributed computing environment that opened up exciting possibilities for new application architectures.

What these cloud application developers sought, above all, was scalability. They wanted to be able to deploy systems that would scale rapidly through many orders of magnitude with as few limitations as possible, and without the requirement to re-visit fundamental aspects of application architecture along the way. They also wanted resilience and fault tolerance; they recognized that failures can occur at every level of the stack, from individual servers to entire data centers, and from individual virtual machines to entire cloud instances, and they needed to come up with software architectures that would survive multiple such failures and continue to deliver services. But they didn't want to buy fault tolerance in the traditional way by doubling up resource usage. Rather, they expected to absorb the impact of failures through modest amounts of surplus capacity combined with automated self-healing capabilities.

In addition to scalability and fault tolerance, cloud application developers wanted to be able to evolve their software solutions quickly to meet new and emerging service requirements. In practice this meant making it possible for multiple teams to work on the software simultaneously without tripping over each other. These were difficult and challenging problems to solve, but the successful pioneers in cloud-based application development employed some of the best brains in the software industry, and there was a good deal of cooperation and sharing of learnings among them. The design patterns of what came to be known as cloud native software architecture have emerged over the last few years as a consensus within this community.

THE KEY FEATURES OF CLOUD NATIVE ARCHITECTURE

Stateless Processing

The requirement for easy scaling across many orders of magnitude is the driver behind the single most important concept in cloud native architecture: stateless processing.

The concept of stateless processing can be described as follows. A transaction processing system is divided into two tiers. One tier

comprises a variable number of identical transaction processing elements that do not store any long-lasting state. The other tier comprises a scalable storage system based on a variable number of elements that store state information securely and redundantly. The transaction processing elements read relevant state information from the state store as required to process any given transaction, and if any state information is updated in the course of processing that transaction, they write the updated state back to the store.

It's probably not obvious from reading the description above how this approach enables massive scalability. So let's use a practical example to illustrate.

Suppose we are developing an e-commerce application. The application needs to support a number of HTTP transaction types including login to account, add item to shopping basket, review shopping basket, checkout etc. The application code that processes these transactions needs access to certain information (i.e. "state"), for example details of the user's account and the current contents of the shopping basket. In a traditional application architecture, this state would be kept in the application's local storage.

The first problem that we need to solve is how to provide fault tolerance. If a server dies, then any local state that is stored in it is lost. The physical servers that are deployed in cloud environments are not particularly reliable, and failures are fairly frequent. Users get pretty upset if they've spent 30 minutes online grocery shopping, and their shopping basket suddenly disappears. We face a difficult choice here: either we accept the risk that a small proportion of e-commerce sessions will fail due to equipment failure, or we have to deploy a second server to act as a backup, and maintain a shadow copy of all the state on it – which doubles the amount of hardware resources the application is consuming.

Now suppose that we need this application to support millions of concurrent online shopping sessions. A single server (or active-standby pair of servers) is not going to be able to handle the load, so we need to deploy a number of servers. The problem that we now need to solve is that each incoming HTTP request needs to be directed to the correct server, the one that knows about this particular user and session. We therefore need to deploy something like a load-balancer in front of our collection of servers, and the load-balancer needs to be able to identify the user and session from the information in each incoming request, remember which server is handling each user session, and re-direct each request to the correct server. The load-balancer is therefore quite a complex application in its own right. And because it's potentially a single point of failure, it needs to be fault tolerant, which makes it even more complex. But the biggest single issue here is that the performance and capacity of the load-balancer puts an upper limit on the transaction processing load that we can handle. What happens if our e-commerce site is wildly successful and we cannot obtain a load-balancer that is powerful enough to handle all of the demand?

With the stateless processing approach, we implement the elements that process HTTP transactions without any local state storage, and have them read and write state to and from a separate storage system. When an HTTP request arrives at one of these elements, it extracts some information from the request that uniquely identifies the session (for example, from a cookie), and then uses this information to retrieve the current state associated with this session (user account details, contents of shopping basket) from the state store. If the transaction has the effect of changing any of this state, for example because the user added an item to her shopping basket, then the transaction processing element writes the updated state back to the state store.

The difference now is that any incoming HTTP request can be handled by any arbitrary instance of the transaction processing element. We do not have to steer each request to the instance that "knows" about it, because knowledge about each session is available to every processing element instance from the state store. We still need some way to balance the load of incoming requests across the population of transaction processing elements, but we can do this without having to deploy a load-balancer, for example by leveraging DNS to perform dumb round-robin load balancing. By eliminating the load-balancer, we've eliminated the limiting factor on scale. We also don't need to worry about any individual transaction processing element failing. Such failures do not result in the loss of any state, because all the state is stored separately.

The stateless approach is therefore inherently fault tolerant. If any processing element instance dies or becomes unresponsive, then the built-in re-try mechanisms of HTTP will result in subsequent attempts being handled by another instance. So long as we have a modest amount of performance headroom in our population of processing elements, the failure of any one of them has no impact on the service: the load that it would otherwise have handled is simply re-distributed across the remaining instances. We can very easily extend this fault tolerance mechanism across multiple data centers, so that even the loss of an entire data center will not bring down our service.

Individual processing elements can be quite small in scale: we can keep the architecture of these elements simple by not worrying about trying to make them very powerful, for example with support for lots of multi-core parallelism. We handle scaling by deploying as many processing element instances as we need to handle the load, an approach which is known as "scale out" (in contrast to "scale up"). We can also change the number of processing elements on the fly (scaling both out and in) in response to changing load – enabling us to make the most efficient use of compute resources at all times.

All of this depends, of course, on our ability to build and deploy a highly scalable and very fault-tolerant storage system in which to keep all of our application state. Because this is an absolutely fundamental requirement of the stateless processing design pattern, there has been a lot of investment in this area, particularly by the main Web-scale players. Many of the solutions

that they have built to address this need are available as open source. For example, one of the leading distributed state stores, Apache Cassandra, was originally developed at Facebook, and is now used by Netflix, Twitter, Instagram and Webex among many others. Test results published by Netflix show Cassandra performance scaling linearly with number of nodes up to 300, and handling over a million writes per second with 3-way redundancy – more than enough to handle the needs of most telco-style services even with many hundreds of millions of subscribers. Cassandra includes support for efficient state replication between geographically separate locations, and therefore provides an excellent basis for extremely resilient geo-redundant services.

It's perhaps worth pointing out that stateless processing is by no means the only design pattern seen in cloud native applications, although it's definitely the most prominent. Other design patterns worth mentioning include stream processing (based on frameworks such as Heron or Storm) and serverless processing, best exemplified by Amazon Lambda. These have only emerged relatively recently, and won't be discussed further in this document – but they definitely have potential to advance the state of the art in Network Functions Virtualization.

Microservices

After stateless processing, the second most frequently cited aspect of the cloud native approach to software design is microservices, defined as follows:

> *Microservices is a software architecture style in which complex applications are composed of small, independent processes communicating with each other using language-agnostic APIs. These services are highly decoupled and focus on doing a single small task well, facilitating a modular approach to system-building.*

Microservices is a big topic: entire books have been written about it, and we only have room for a brief summary here. The main benefits of a microservices approach are as follows:

Composability and reusability. Microservices encourages the development of modular software components each of which performs a very specific task that is exposed via a well-documented API. Components built this way lend themselves to easy re-use in a variety of different circumstances, enabling applications to be "composed" by combining a suitable set of microservices glued together by a lightweight front end.

Technology heterogeneity. Microservices enables development teams to pick the best software technology and language for the implementation of any given application component, without worrying about the rest of the system. Components are loosely-coupled, typically via Web services APIs, and this hides their implementation details.

Efficient scaling. Each microservice can be designed to scale out independently of other microservices associated with a given application, which typically means we get more efficient use of resources than with monolithic applications where all functions have to scale in lockstep.

Ease of development and deployment. It's possible to make incremental enhancements to microservices and deploy these to production independently of other microservices. If any problems arise from the new version of a given microservices component, the change can quickly be rolled back. This allows for a DevOps approach to the progressive enhancement of an overall application, enabling innovations to be introduced much more rapidly than with monolithic applications which inevitably accumulate many changes between releases, requiring far more comprehensive testing.

Those with a long history in software may be tempted to dismiss microservices as just a new label for Service-Oriented Architecture (SOA), which has been around for many years. There are, of course, many similarities and some practitioners talk about microservices as "fine-grained SOA". The main difference from SOA is the size and scope of the service components: making them fine-grained improves composability, reusability and ease of deployment. Making them too fine-grained may introduce unacceptable inefficiency in the application, so getting the balance right is important.

The microservices approach is not a panacea. Highly distributed loosely-coupled systems bring their own complications, and the complexity of a large application does not disappear just because it is reduced to a set of relatively simple components. Nevertheless, most of the Web-scale players are strongly bought into microservices, none more so than Netflix. Following a disastrous outage in 2008, Netflix started transitioning away from a single monolithic Web application, and has now deployed in excess of 500 microservices to support their Web presence and business operations. Netflix has blogged extensively about its microservices journey, and this material is essential reading for anyone wanting to get under the skin of this approach to system design.

Open Source Software

Making use of open source software components is not in any sense a fundamental requirement of a cloud native architecture, but it's an observable fact that most developers of cloud native applications leverage open source software very effectively.

We've already mentioned one open source project that is widely used in cloud native applications – the scalable state storage database Apache Cassandra. There are many other open source solutions for state storage, including MongoDB, Couchbase and Memcached. And there are open source solutions for many other kinds of generic functions that are needed for cloud native applications and their management: Web servers (Apache HTTP Server, NGINX), protocol stacks (libcurl, snmpd), client-side scripting (jQuery), secure communications (OpenSSL), DNS (dnsmasq), monitoring (monit), log collection / storage / visualization (Elasticsearch, Fluentd, Kibana) and so on.

Many of these open source projects have substantial communities supporting them and many years of broad exposure in the field, so they can be incorporated into cloud native applications with confidence. Occasionally, using these kinds of open source software in some new application exposes bugs or deficiencies which the community behind the project may not view as high priorities to fix. In that case, the developers of the application usually address the problem themselves, and upstream the fixes. Making necessary improvements in open source components of a cloud native application is one of the overheads that should be taken into account in the development planning process.

Making wise use of open source software can completely transform the economics of developing and supporting complex Web-scale applications by dramatically reducing the amount of new code that needs to be written, and by leveraging the community to provide support and bug fixes. Relatively small teams of developers can complete substantial projects much faster and with fewer bugs than if they had to write all of the application code, and can focus on the most important aspect of new applications: innovation.

Containers

In the discussion above on the history of virtualization, we described the hypervisor and its support for the deployment of application software in virtual machines. But there is an alternative approach to virtualization that happens to be particularly well-suited to cloud native applications: Linux containers.

Containers leverage a long-standing method for partitioning in Linux known as "namespaces", which provides separation of different processes, filesystems and network stacks. A container is a secure partition based on namespaces in which one or more Linux processes run, supported by the Linux kernel installed on the host system.

The main difference between a container and a virtual machine is that a virtual machine needs a complete operating system installed in it to support the application, whereas a container only needs to package up the application software, with the optional addition of any application-specific OS dependencies, and leverages the operating system kernel running on the host.

Containers offer a number of advantages over virtual machines, including the following:

Lower overhead. Because they do not (in most cases) contain complete operating system images, containers have a far smaller memory footprint than virtual machines, and therefore consume considerably less hardware resources. Their small footprint may make it feasible to deploy instances of software to serve single tenants for some kinds of services, and this could simplify the design of the software very considerably.

Startup speed. Virtual machine images are large because they include a complete guest operating system, and the time taken to start a new VM is largely dictated by the time taken to copy its image to the host on which it is to run, which may take many seconds. By contrast, container images tend to be very small, and they can often start up in less than 50 ms. This enables cloud native applications to scale and heal extremely quickly, and also allows for new approaches to system design in which containers are spawned to process individual transactions, and are disposed of as soon as the transaction is complete.

Reduced maintenance. Virtual machines contain guest operating systems, and these must be maintained, for example to apply security patches to protect against recently discovered vulnerabilities. Containers require no equivalent maintenance.

Ease of deployment. Containers provide a high degree of

portability across operating environments, making it easy to move a containerized application from development through testing into production without having to make changes along the way. Furthermore, containers allow workloads to be moved easily between private and public cloud environments. Being much more straightforward to deploy in the cloud than virtual machines, they are also much easier to orchestrate.

If you need to deploy an application that was originally designed to run on a dedicated server into a cloud environment, chances are you will need to deploy it in a virtual machine because of operating system or hardware dependencies. But if you are writing new software to run in a cloud environment (in other words, cloud native software), then it's very easy to do so in a container-friendly way. Most cloud native software in the Web-scale world today is deployed in containers because of the compelling benefits they offer.

Design for Orchestration

Cloud native applications tend to comprise a substantial number of different software components, partly because they usually implement stateless processing (and therefore have separate components for transaction processing and state storage), and partly because they are usually decomposed into a number of microservices. Furthermore, each microservice is designed to scale out, and so multiple instances of each microservice component need to be deployed to handle the load on the application. For these reasons, deploying a cloud native application at scale may require the instantiation of many tens or hundreds of virtual machines or containers.

It is totally infeasible to carry out the deployment of such an application manually, so cloud native applications are invariably orchestrated in some way so as to automate the deployment process. Likewise, orchestration is needed to automate operations such as scaling of the different microservices and healing failed instances because these would be too complex and onerous to perform manually.

With this in mind, the cloud native application designer pays close attention to the requirements of orchestration and operations automation right from the outset. The main focus is on achieving the simplest possible process for bringing up the components of the application, mainly by minimizing the amount of configuration that needs to be injected into each component. The following practices are commonly employed in cloud native applications to keep things simple from an orchestration standpoint.

Automated IP address assignment. Cloud native application components invariably use DHCP to obtain IP addresses, so the orchestrator does not need to be involved in IP address management.

Shared configuration stores. Cloud native application components very often participate in a shared distributed key-value store from which they can obtain most or all of the configuration they need without the orchestrator having to take responsibility for this.

Automated discovery. Cloud native application components typically discover the peers with which they need to communicate either via a shared configuration store or via DNS.

Elimination of hard dependencies. Many inter-component dependencies typically exist within a given cloud native application, but the components are designed to be brought up in any order. If one component depends on a microservice exposed by another component, and that microservice is not yet available, then the component will keep trying to connect to it until it becomes available.

APPLYING CLOUD NATIVE PRINCIPLES TO VIRTUALIZED NETWORK FUNCTIONS

We've discussed cloud native software architecture in the context of Web-scale applications such as messaging, social media and e-commerce, all of which are essentially transactional in nature. At this point, it is reasonable to ask the question: can these techniques really be applied to the implementation of virtualized network functions, given that these may be somewhat different in nature from Web-scale applications?

In considering how cloud native principles may be applied to the development of VNFs, we need to make a clear distinction between control plane functions and data plane (or user plane) functions.

Control Plane Functions

Control plane functions involve the exchange and processing of messages. For example, routers exchange Border Gateway Protocol messages to learn about the reachability of IP address blocks, and subscribers exchange Session Initiation Protocol messages with an IP Multimedia Subsystem in order to negotiate the establishment of a voice or video session. These functions are transactional in exactly the same sense as the Web-scale applications that we've used as examples of cloud native architecture in action, and all of the cloud native principles can be fully applied to their implementation. Metaswitch's cloud native IMS core solution, Clearwater, is a good example of this.

Data Plane Functions

Data plane functions involve processing packets or packet flows at various levels of the protocol stack. For example, routers forward packets at the IP layer, and may also manipulate packets by terminating tunnels, inserting VLAN tags and so on, while session border controllers forward media packets at the application layer, and may perform various media processing functions. It could possibly be argued that a data plane function is transactional in the sense that each incoming packet represents a "transaction". However, the work done on each packet in a data plane function is typically many orders of magnitude less than the work done in processing a control plane transaction, and simple economics requires us to process many orders of magnitude more packets

with a given amount of compute resource compared with control plane transactions. It is impractical to implement a stateless processing model for data plane functions because there is far too much overhead involved in fetching the required state to process each packet from a separate store. The state that we require to process each packet must be locally resident in the network function, in memory or (preferably) in the processor's cache, in order for us to process packets economically.

In the section above on stateless processing, we explained that the stateless approach enables us easily to scale out an application, and implement fault tolerance with an active-active N+k redundancy model. So if we can't apply stateless processing to data plane functions, does that mean that we can't build a scale-out, active-active N+k data plane function? The answer to this is an emphatic no. By applying appropriate ingenuity in the way we manage and store state, and how we steer packet flows, we absolutely can build data plane functions that scale out with active-active N+k redundancy. For example, we can divide the state information in any one data plane instance into logical blocks or "shards", and re-distribute these shards across the remaining population of data plane instances when one fails, while modifying the steering of flows to match.

The next topic we need to consider is whether it makes sense to decompose data plane functions into microservices. We can certainly imagine defining any given data plane function as a sequence of basic actions to be applied to each packet (a packet processing graph), but does it make sense to implement the function with a separate software component for each basic action? The answer to this question depends on exactly how these components are combined together to deliver the complete function. Implementing each basic action as a separately deployable software element in a virtual machine or container, and stringing them together by means of Service Function Chaining or some similar technique, may provide a great deal of flexibility and composability, but it does so at the expense of enormous inefficiency. This is because the work done in the underlying fabric to encapsulate and forward packets between each node of the packet processing graph is likely to be considerably greater than the work done by the packet processing functions themselves. On the other hand, if the software elements that implement each of the basic actions can be composed into a packet processing graph in the context of a single engine, in which packets are passed between components in memory, then we have a "microservices" data plane solution that combines composability nicely with efficiency.

This concept of a composable packet processing engine in which multiple software components that perform basic actions on packets are combined into a single deployable element is gaining currency in the industry. Two open source projects that implement this concept were launched in 2016: FD.io and BESS (Berkeley Extensible Software Switch), both of which appear to offer great promise for the rapid development of data plane VNFs. We believe that this approach is the right way to think about the application of microservices in the data plane domain.

The remaining aspects of the cloud native approach – leveraging of open source software, containerization and design for orchestration – are all fully applicable to data plane functions. We have already mentioned two open source projects that address the data plane – FD.io and BESS. Linux containers are fully capable of supporting the packaging of data plane functions, although it should be noted that, as of January 2017, none of the container orchestration solutions (including Kubernetes) is currently able to handle the detailed configuration of network connectivity to meet the complete needs of data plane functions. We expect this to change in the near future. And finally, data plane functions are just as amenable to design for easy orchestration as control plane functions.

NETWORK FUNCTION SOFTWARE--TRADITIONAL ARCHITECTURE

Having described the main features of the cloud native approach to software design, we should now characterize traditional software architectures – like those that are found in physical network appliances – in order to highlight just how fundamentally different the cloud native approach is.

Stateful processing. All state required by a processing element to enable it to do its work is stored locally. This has two main negative impacts by comparison with cloud native. Scaling requires a stateful load balancer which puts an upper limit on achievable scale. And fault tolerance is usually implemented with a 1+1 active / standby approach, which doubles the hardware resources needed to support the function.

Monolithic design. The application is one big lump of code. If there is some decomposition, it usually reflects the physical architecture of the hardware appliance for which the software was designed, with a software module for each blade, but these modules are generally tightly-coupled and have complex interdependencies. The package typically includes large amounts of functionality that is irrelevant to most individual use cases, but the entire package needs thorough testing of all its functionality when any change is made to any part of it. This testing overhead severely limits the frequency of new releases, meaning long cycles for the introduction of any innovation. There is no possibility of separating small, distinct elements of software functionality from the main body of code, and making use of them elsewhere.

Preponderance of proprietary software. The great majority, if not all, of the software is written specifically for this network function, and very little use of open source software is made. This dramatically increases the time and cost of developing the software. The high costs of development have to be recovered from customers in the form of high prices, and long development times slow down innovation.

Operating system dependencies. The software may make heavy use of specialized operating system or middleware functions, for example to perform state replication to a standby system. This

requires the application to be packaged up with its operating system and middleware and deployed in a virtual machine. Because it is not possible to deploy the application in containers, it requires more hardware resources than cloud native applications, it is considerably harder to deploy and orchestrate, and it needs a good deal more maintenance.

Hardware dependencies. The software in many networking appliances is written to take advantage of specific hardware capabilities provided by a purpose-built, proprietary platform. As it stands, this software cannot even run on a bare metal off-the-shelf server, let alone in a cloud environment. Substantial elements of the software typically need to be re-written to run on standard hardware. In some cases, a software emulation of the native proprietary hardware environment is created to enable existing binaries to be deployed on standard hardware without having to modify them. It hardly needs to be said that this is not a recipe for promoting rapid innovation.

Complex configuration and bring-up procedure. Physical appliances are typically deployed only once in their lives, so little attention is paid to simplifying their bring-up procedure. Complex sequences of commands need to be entered by hand, IP addresses need to be manually assigned and configured, and this initial configuration must be applied consistently across the different modules of the system. Such systems are extremely difficult to orchestrate successfully.

VIRTUALIZATION AND THE VNF ARCHITECTURE DILEMMA

Many network operators have a reasonable understanding of cloud native and its advantages, and if they had a choice, would strongly prefer to deploy Virtualized Network Functions built on cloud native principles rather than those that started out their lives as appliances and follow a traditional design pattern – what we call “ported appliances”.

But there’s a real problem here. Most of the network functions that network operators wish to virtualize have evolved over many years and have acquired a great deal of complexity as they have adapted to meet each new generation of requirements and industry standards. The software that powers the physical versions of these functions may comprise millions of lines of code, and the original codebase may have begun its life fifteen or twenty years ago. As we have seen, the cloud native approach to software architecture has some really fundamental differences from the traditional approach, and these differences run so deep that it is usually not remotely feasible to consider re-factoring the existing software to fully embody cloud native principles.

It is entirely possible to build complex telco-grade and standards-compliant network functions using a cloud native approach. Metaswitch is one of the first in the industry to have demonstrated this, with its Project Clearwater, a cloud native implementation of

the IMS Call Session Control Functions that was first released in May 2013 and is now in production in a number of networks. But it's really only feasible to build cloud native VNFs by starting with a clean sheet of paper. This necessarily means that early releases of such VNFs have had no field exposure in large-scale production deployments and no reference customers. As a result, telcos are extremely reluctant to take a risk on them, preferring to stick with what they perceive to be tried and trusted solutions: mature software that started life as part of physical appliances, and that has been ported to run in a virtualized environment.

The problem with an NFV strategy based on deploying ported appliance software is that it will fail to deliver the majority of benefits that network operators are looking for from NFV.

Opex reductions. Ported appliances are generally extremely costly and complex to on-board to management and orchestration systems, and the on-boarding adaptations will need constant attention throughout the life of the product since new releases of VNF software are very likely to break them. Ported appliances generally don't scale out, so additional complexity in the form of load-balancers will need to be deployed and managed. From an operations management point of view, a ported appliance looks much the same as physical appliance, and will require a similar amount of effort to manage. And finally, ported appliances will invariably need to be deployed in virtual machines, which come with a considerably greater operational overhead than containers.

Service innovation velocity. Ported appliances will suffer from the same long release cycles as the software on the physical devices from which they originated, so there will be little or no opportunity to introduce service innovations any more rapidly than before.

Capex reduction. Ported appliances that meet all of the functional and operational requirements demanded by network operators to replace physical equivalents will be available only from those same vendors that traditionally supplied those physical devices. Without the disruptive impact of new vendors in the market, there will be no incentive for the incumbent vendors to reduce prices. Furthermore, ported appliances invariably make inefficient use of hardware resources, partly because they tend to implement 1+1 fault tolerance, and partly because their hardware-centric developers are not skilled in the art of delivering good performance on standard hardware.

PRACTICAL GUIDANCE

The VNF architecture dilemma presents a really challenging problem for network operators. There's a difficult risk/reward trade-off to be made: cloud native VNFs are clearly capable of delivering far more of the full potential benefits of NFV, but they may appear to be a considerably riskier option than ported appliances.

We would make two recommendations to network operators who are facing this dilemma:

Make the investment in evaluating cloud native VNFs. If there is a credible cloud native solution on the market for some particular network function that you are planning to virtualize, make the investment to properly evaluate it. You may be very pleasantly surprised.

Insist that your VNF vendors do their best to embrace cloud native practices. It is probably not reasonable to expect vendors to re-factor complex existing network function software to implement stateless processing or microservices. But it is reasonable to expect them to radically simplify their bring-up and configuration processes so as enable straightforward on-boarding to orchestrators.

CONCLUSION

The extraordinarily successful growth of over-the-top social media, messaging and real-time communications applications in recent years has demonstrated very clearly the enormous power of the cloud native approach to software design. These kinds of Web-scale applications have proven themselves to be massively scalable, highly fault tolerant, extremely cost-effective, and capable of evolving very rapidly to better meet the needs of their users.

The Network Functions Virtualization movement was born of the recognition by leading telcos that they could learn something from the success of the Web-scale world and apply those learnings to their businesses. In this white paper, we have attempted to distil out the key software techniques that are prevalent in the Web-scale world, and show how they may be applied in a telco environment. We have focused on the cloud native approach to software design, and shown how it can be applied to the building of Virtualized Network Functions to greatly improve scalability, fault tolerance, efficiency, orchestratability and service innovation velocity.

No-one involved in NFV is under any illusions about the enormity of the challenge that the industry is facing, or the extent of the upheaval that network operators will have to go through in order successfully to virtualize their networks. That's why it's so important that we get NFV right. And cloud native is NFV done right.

ABOUT THE AUTHOR

Martin Taylor is chief technical officer of Metaswitch Networks. He joined the company in 2004, and headed up product management prior to becoming CTO. Previous roles have included founding CTO at CopperCom, a pioneer in Voice over DSL; VP of Network Architecture at Madge Networks; and business general manager at GEC-Marconi. In January 2014, Martin was recognized by Light Reading as one of the top five industry "movers and shakers" in Network Functions Virtualization.