NFVRG
Internet Draft
Category: Informational

S. Natarajan
Google
R. Krishnan
A. Ghanwani
Dell
D. Krishnaswamy
IBM Research
P. Willis
BT
A. Chaudhary
Verizon
F. Huici
NEC

Expires: January 2017 July 8, 2016

An Analysis of Lightweight Virtualization Technologies for NFV

draft-natarajan-nfvrg-containers-for-nfv-03

Abstract

Traditionally, NFV platforms were limited to using standard virtualization technologies (e.g., Xen, KVM, VMWare, Hyper-V, etc.) running guests based on general-purpose operating systems such as Windows, Linux or FreeBSD. More recently, a number of light-weight virtualization technologies including containers, unikernels (specialized VMs) and minimalistic distributions of general-purpose OSes have widened the spectrum of possibilities when constructing an NFV platform. This draft describes the challenges in building such a platform and discusses to what extent these technologies, as well as traditional VMs, are able to address them.

at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire in January 2017.

Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Table of Contents

1. Introduction

This draft describes the challenges when building an NFV platform by describing to what extent different types of lightweight virtualization technologies, such as VMs based on minimalistic distributions, unikernels and containers, are able to address them.

2. Lightweight Virtualization Background

2.1. Containers

Containers are a form of operating-system virtualization. To provide isolation, containers such as Docker rely on features of the Linux kernel such as cgroups, namespaces and a union-capable file system such as aufs and others [AUFS]. Because they run within a single OS instance, they avoid the overheads typically associated with hypervisors and virtual machines.

2.2. OS Tinyfication

OS tinyfication consists of creating a minimalistic distribution of a general-purpose operating system such as Linux or FreeBSD. This involves two parts: (1) configuring the kernel so that only needed features and modules are enabled/included (e.g., removing extraneous drivers); and (2) including only the required user-level libraries and applications needed for the task at hand, and running only the minimum amount of required processes. The most notable example of a tinyfied OS is the work on Linux tinyfication [LINUX-TINY].

2.3. Unikernels

Unikernels are essentially single-application virtual machines based on minimalistic OSes. Such minimalistic OSes have minimum overhead and are typically single-address space (so no user/kernel space divide and no expensive system calls) and have a co-operative scheduler (so reducing context switch costs). Examples of such minimalistic OSes are MiniOS [MINIOS] which runs on Xen and OSv [OSV] which runs on KVM, Xen and VMWare.

3. Challenges in Building NFV Platforms

In this section, we outline the set of main challenges for an NFV platform in the context of lightweight virtualization technologies as well as traditional VMs.

3.1. Performance (SLA)

Performance requirements vary with each VNF type and configuration. The platform should support the specification, realization and runtime adaptation of different performance metrics. Achievable performance can vary depending on several factors such as the workload type, the size of the workload, the set of virtual machines sharing the underlying infrastructure, etc. Here we highlight some of the challenges based on potential deployment considerations.

3.1.1. Challenges

. VNF provisioning time (including up/down/update) constitutes the time it takes to spin-up the VNF process, its application-specific dependencies, and additional system dependencies. The resource choices such as the hypervisor type, the guest and host OS flavor and the need for hardware and software accelerators, etc., constitute a significant portion of this processing time (instantiation or down time) when compared to just bringing up the actual VNF process.

. The runtime performance (achievable throughput, line rate speed, maximum concurrent sessions that can be maintained, number of new sessions that can be added per second) for each VNF is directly dependent on the amount of resources (e.g., virtual CPUs, RAM) allocated to individual VMs. Choosing the right resource setting is a tricky task. If VM resources are over-provisioned, we end up under-utilizing the physical resources. On the contrary if we under-provision the VM resources, then upgrading the resource to an advanced system setting might require scaling out or scaling up of the resources and re-directing traffic to the new VM; scaling up/down operations consume time and add to the latency. This

overhead stems from the need to account resources of components other than the actual VNF process (e.g., guest OS requirements).

. If each network function is hosted in individual VMs/containers, then an efficient inter-VM networking solution is required for performance.

3.2. Continuity, Elasticity and Portability

VNF service continuity can be interrupted due to several factors: undesired state of the VNF (e.g., VNF upgrade progress), underlying hardware failure, unavailability of virtualized resources, VNF SW failure, etc. Some of the requirements that need consideration are:

3.2.1. Challenges:

- VNF's are not completely decoupled from the underlying infrastructure. As discussed in the previous section, most VNFs have a dependency on the guest OS, hypervisor type, accelerator used, and the host OS (this last one applies to containers too). Therefore porting VNFs to a new platform might require identifying equivalent resources (e.g., hypervisor support, new hardware model, understanding resource capabilities) and repeating the provisioning steps to bring back the VNF to a working state.

- Service continuity requirements can be classified as follows: seamless (with zero impact) or non-seamless continuity (accepts measurable impacts to offered services). To achieve this, the virtualization technology needs to provide an efficient high availability solution or a quick restoration mechanism that can bring back the VNF to an operational state. For example, an anomaly caused by a hardware failure can impact all VNFs hosted on that infrastructure resource. To restore the VNF to a working state, the user should first provision the VM/container, spin-up and configure the VNF process inside the VM, setup the interconnects to forward network traffic, manage the VNF-related state, and update any dependent runtime agents.

- Addressing the service elasticity challenges require a holistic view of the underlying resources. The challenges for presenting a holistic view include the following

    - Performing Scalable Monitoring: Scalable continuous monitoring of the individual resource's current state is needed to spin-up additional resources (auto-scale or auto-

heal) when the system encounters performance degradation or spin-down idle resources to optimize resource usage.

- Handling CPU-intensive vs I/O-intensive VNFs: For CPU-intensive VNFs the degradation can primarily depend on the VNF processing functionality. On the other hand, for I/O intense workloads, the overhead is significantly impacted by to the hypervisor/host features, its type, the number of VMs/contaiers it manages, the modules loaded in the guest OS, etc.

3.3. Security

Broadly speaking, security can be classified into:

- Security features provided by the VNFs to manage the state, and

- Security of the VNFs and its resources.

Some considerations on the security of the VNF infrastructure are listed here.

3.3.1. Challenges

- The adoption of virtualization techniques (e.g., para-virtualization, OS-level) for hosting network functions and the deployment need to support multi-tenancy requires secure slicing of the infrastructure resources. In this regard, it is critical to provide a solution that can ensure the following:

  - Provision the network functions by guaranteeing complete isolation across resource entities (hardware units, hypervisor, virtual networks, etc.). This includes secure access between VM/container and host interface, VM-VM or container-to-container communication, etc. For maximizing overall resource utilization and improving service agility/elasticity, sharing of resources across network functions must be possible.

  - When a resource component is compromised, quarantine the compromised entity but ensure service continuity for other resources.

  - Securely recover from runtime vulnerabilities or attacks and restore the network functions to an operational state. Achieving this with minimal or no downtime is important.

Realizing the above requirements is a complex task in any type of virtualization option (virtual machines, containers, etc.)

- Resource starvation / Availability: Applications hosted in VMs/containers can starve the underlying physical resources such that co-hosted entities become unavailable. Ideally, countermeasures are required to monitor the usage patterns of individual VMs/containers and ensure fair use of individual resources.

3.4. Management

The management and operational aspects are primarily focused on the VNF lifecycle management and its related functionalities. In addition, the solution is required to handle the management of failures, resource usage, state processing, smooth rollouts, and security as discussed in the previous sections. Some features of management solutions include:

- oCentralized control and visibility: Support for web client, multi-hypervisor management, single sign-on, inventory search, alerts & notifications.
- oProactive Management: Creating host profiles, resource management of VMs/containers, dynamic resource allocation, auto-restart in HA model, audit trails, patch management.
- oExtensible platform: Define roles, permissions and licenses across resources and use of APIs to integrate with other solutions.

Thus, the key requirements for a management solution

- Simple to operate and deploy VNFs.
- Uses well-defined standard interfaces to integrate seamlessly with different vendor implementations.
- Creates functional automation to handle VNF lifecycle requirements.
- Provide APIs that abstracts the complex low-level information from external components.
- Is secure.

3.4.1. Challenges

The key challenge is addressing the aforementioned requirements for a management solution while dealing with the multi-dimensional complexity introduced by the hypervisor, guest OS, VNF functionality, and the state of network.

4. Benchmarking Experiments

Having considered the basic requirements and challenges of building an NFV platform, we now provide a benchmark of a number of lightweight virtualization technologies to quantify to what extent they can be used to build such a platform.

4.1. Experimental Setup

In terms of hardware, all tests are run on an x86-64 server with an Intel Xeon E5-1630 v3 3.7GHz CPU (4 cores) and 32GB RAM.

For the hypervisors we use KVM running on Linux 4.4.1 and Xen version 4.6.0. The virtual machines running on KVM and Xen are of three types:

(1)Unikernels, on top of the minimalistic operating systems OSv and MiniOS for KVM and Xen, respectively. The only application built into them is iperf. To denote them we use the shorthand unikernel.osv.kvm or unikernels.minios.xen.

(2)Tinyfied Linux (a.k.a. Tinyx), consisting of a Linux kernel version 4.4.1 with only a reduced set of drivers (ext4, and netfront/blkfront for Xen), and a distribution containing only busybox, an ssh server for convenience, and iperf. We use the shorthand tinyx.kvm and tinyx.xen.

(3)Standard VM, consisting of a Debian distribution including iperf and Linux version 4.4.1. We use the shorthand standardvm.kvm and standardvm.xen for it.

For containers, we use Docker version 1.11 running on Linux 4.4.1.

It is worth noting that the numbers reported here for virtual machines (whether standard, Tinyx or unikernels) include the following optimizations to the underlying virtualization technologies. For Xen, we use the optimized Xenstore, toolstack and hotplug scripts reported in [SUPERFLUIDITY] as well as the accelerated packet I/O derived from persistent grants (for Tx)

[PGRANTS]. For KVM, we remove the creation of a tap device from the VM's boot process and use a pre-created tap device instead.

4.2. Instantiation Times

We begin by measuring how long it takes to create and boot a container or VM. The beginning time is when we issue the create operation. To measure the end time, we carry out a SYN flood from an external server and measure the time it takes for the container/VM to respond with a RST packet. The reason for a SYN flood is that it guarantees the shortest reply time after the unikernels/container is booted. It is just to measure boot time, nothing to do with real-world deployments and DoS attacks.

Technology Type	Time (msecs)
standardvm.xen	6500
standardvm.kvm	2988
Container	1711
tinyx.kvm	1081
tinyx.xen	431
unikernel.osv.kvm	330
unikernels.minios.xen	31

The table above shows the results. Unsurprisingly, standard VMs with a regular distribution (in this case Debian) fare the worst, with times in the seconds: 6.5s on Xen and almost 3s on KVM. The Docker container with iperf comes next, clocking in at 1.7s. The next best times are from Tinyx: 1s approximately on KVM and 431ms on Xen. Finally, the best numbers come from unikernels, with 330ms for OSv on KVM and 31ms for MiniOS on Xen. These results show that at least when compared to unoptimized containers, minimalistic VMs or unikernels can have instantiation times comparable to or better than containers.

4.3. Throughput

To measure throughput we use the iperf application that is built in to the unikernels, included as an application in Tinyx and the Debian-based VMs, and containerized for Docker. The experiments in this section are for TCP traffic between the guest and the host

where the guest resides: there are no NICs involved so that rates are not bound by physical medium limitations.

Technology Type	Throughput (Gb/s) Tx	Throughput (Gb/s) Rx
standardvm.xen	23.1	24.5
standardvm.kvm	20.1	38.9
Container	45.1	43.8
tinyx.kvm	21.5	37.9
tinyx.xen	28.6	24.9
unikernel.osv.kvm	47.9	47.7
unikernels.minios.xen	49.5	32.6

The table above shows the results for Tx and Rx. The first thing to note is that throughput is not only dependent on the guest's efficiency, but also on the host's packet I/O framework (e.g., see [CLICKOS] for an example of how optimizing Xen's packet I/O subsystem can lead to large performance gains). This is evident from the Xen numbers, where Tx has been optimized and Rx not. Having said that, the guest also matters, which is why, for example, Tinyx scores somewhat higher throughput than standard VMs. Containers and unikernels (at least for Tx and for Tx/Rx for KVM) are fairly equally matched and perform best, with unikernels having a slight edge.

4.4. RTT

To measure round-trip time (RTT) from an external server to the VM/container we carry out a ping flood and report the average RTT.

Technology Type	Time (msecs)
standardvm.xen	34
standardvm.kvm	18
Container	4
tinyx.kvm	19
tinyx.xen	15
unikernel.osv.kvm	9
unikernels.minios.xen	5

As shown in the table above, the Docker container comes out on top with 4ms, but unikernels achieve for all practical intents and purposes the same RTT (5ms on MiniOS/Xen and 9ms on OSv/KVM). Tinyx fares slightly better than the standard VMs.

4.5. Image Size

We measure image size using the standard "ls" tool.

Technology Type	Size (MBs)
standardvm.xen	913
standardvm.kvm	913
Container	61
tinyx.kvm	3.5
tinyx.xen	3.7
unikernel.osv.kvm	12
unikernels.minios.xen	2

The table shows the standard VMs to be unsurprisingly the largest and, followed by the Docker/iperf container. OSv-based unikernels are next with about 12MB, followed by Tinyx (3.5MB or 3.7MB on KVM and Xen respectively). The smallest image is the one based on MiniOS/Xen with 2MB.

4.6. Memory Usage

For the final experiment we measure memory usage for the various VMs/container. To do so we use standard tools such as "top" and "xl" (Xen's management tool).

Technology Type	Usage (MBs)
standardvm.xen	112
standardvm.kvm	82
Container	3.8
tinyx.kvm	30
tinyx.xen	31
unikernel.osv.kvm	52
unikernels.minios.xen	8

The largest memory consumption, as shown in the table above, comes from the standard VMs. The OSv-based unikernels comes next due to the fact that OSv pre-allocates memory for buffers, among other things. Tinyx is next with about 30MB. From there there's a big jump to the MiniOS-based unikernels with 8MB. The best result comes from the Docker container, which is expected given that it relies on the host and its memory allocations to function.

5. Discussion

In this section we provide a discussion comparing and contrasting the various lightweight virtualization technologies in view of the reported benchmarks. There are a number of issues at stake:

- Service agility/elasticity: this is largely dependent on the ability to quickly spin up/down VMs/containers and migrate them. Clearly the best numbers in this category come from unikernels and containers.

- Memory consumption: containers use and share resources from the common host they use and so each container instance uses up less memory than VMs, as shown in the previous section (although unikernels are not far behind). Note: VMs also have a common host (or dom0 in the case of Xen) but they incur the overhead of each having its own guest OS.

- Security/Isolation: an NFV platform needs to provide good isolation for its tenants. Generally speaking, VM-based technologies have been around for longer and so have had time to iron out most of the security issues they had. Type-1 hypervisors (e.g., Xen), in addition, provide a smaller attack surface than Type-2 ones (e.g., KVM) so should in principle be more robust. Containers are relatively newcomers and as such still have a number of open issues [CONTAINER-SECURITY]. Use of kernel security modules like SELinux [SELINUX], AppArmor [APPARMOR] along with containers can provide at least some of the required features for a secure VNF deployment. Use of resource quota techniques such as those in Kubernetes [KUBERNETES-RESOURCE-QUOTA] can provide at least some of the resource guarantees for a VNF deployment.

- Management frameworks: both virtual machines and containers have fully-featured management frameworks with large open source communities continually improving them. Unikernels might need a bit of "glue" to adapt them to an existing framework (e.g., OpenStack).

- Compatibility with applications. Both containers and standard VMs can run any application that is able to run on the general-purpose OS those VMs/containers are based on (typically Linux). Unikernels, on the hand, use minimalistic OSes, which might present a problem. OSv, for example, is able to build a unikernels as long as the application can be recompiled as a shared library. MiniOS requires that the application be directly compiled with it (c/c++ is the default, but MiniOS unikernels based on OCaml, Haskell and other languages exist).

Overall, the choice between standard virtual machines, tinyfied ones, unikernels or containers is often not a black and white one. Rather, these technologies present points in a spectrum where criteria such as security/isolation, performance, and compatibility with existing applications and frameworks may point NFV operators, and their clients, towards a particular solution. For instance, an operator for whom excellent isolation and multi-tenancy is a must might lean towards hypervisor-based solutions. If that operator values ease of application deployment he will further choose guests based on a general-purpose OS (whether tinfyied or not). Another operator might put a prime on performance and so might prefer unikernels. Yet another one might not have a need for multi-tenancy (e.g., Google, Edge use cases such as CPE) and so would lean towards enjoying the benefits of containers. Hybrid solutions, where containers are run within VMs, are also possible. In short, choosing a virtualization technology for an NFV platform is (no longer) as simple as choosing VMs or containers.

6. Conclusion

In this draft we presented the challenges when building an NFV platform. We further introduced a set of benchmark results to quantify to what extent a number of virtualization technologies (standard VMs, tinfyied VMs, unikernels and containers) can meet those challenges. We conclude that choosing a solution is nuanced, and depends on how much value different NFV operators place on criteria such as strong isolation, performance and compatibility with applications and management frameworks.

7. Future Work

Opportunistic areas for future work include but not limited to developing solutions to address the VNF challenges described in Section 3, distributed micro-service network functions, etc.

8. IANA Considerations

This draft does not have any IANA considerations.

9. Security Considerations

VM-based VNFs can offer a greater degree of isolation and security due to technology maturity as well as hardware support. Light-weight virtualization technologies such as unikernels (specialized VMs) and tinyfied VMs which were discussed enjoyed the security benefits of a standard VM. Since container-based VNFs provide abstraction at the OS level, it can introduce potential vulnerabilities in the system when deployed without proper OS-level security features. This is one of the key implementation/deployment challenges that needs to be further investigated.

In addition, as containerization technologies evolve to leverage the virtualization capabilities provided by hardware, they can provide isolation and security assurances similar to VMs.

10. Contributors

11. Acknowledgements

The authors would like to thank Vineed Konkoth for the Virtual Customer CPE Container Performance white paper. The authors would like to acknowledge Louise Krug (BT) for their valuable comments.

12. References

12.1. Normative References

12.2. Informative References

[AUFS] "Advanced Multi-layered Unification Filesystem," https://en.wikipedia.org/wiki/Aufs

[CONTAINER-SECURITY] "Container Security article," http://www.itworld.com/article/2920349/security/for-containers-security-is-problem-1.html

[ETSI-NFV-WHITE] "ETSI NFV White Paper," http://portal.etsi.org/NFV/NFV_White_Paper.pdf

[ETSI-NFV-USE-CASES] "ETSI NFV Use Cases," http://www.etsi.org/deliver/etsi_gs/NFV/001_099/001/01.01.01_60/gs_NFV001v010101p.pdf

[ETSI-NFV-REQ] "ETSI NFV Virtualization Requirements," http://www.etsi.org/deliver/etsi_gs/NFV/001_099/004/01.01.01_60/gs_NFV004v010101p.pdf

[ETSI-NFV-ARCH] "ETSI NFV Architectural Framework," http://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.01.01_60/gs_NFV002v010101p.pdf

[ETSI-NFV-TERM] "Terminology for Main Concepts in NFV," http://www.etsi.org/deliver/etsi_gs/NFV/001_099/003/01.01.01_60/gs_nfv003v010101p.pdf

[KUBERNETES-RESOURCE-QUOTA] "Kubernetes Resource Quota," http://kubernetes.io/v1.0/docs/admin/resource-quota.html

[KUBERNETES-SELF-HEALING] "Kubernetes Design Overview," http://kubernetes.io/v1.0/docs/design/README.html

[LINUX-TINY] "Linux Kernel Tinification," https://tiny.wiki.kernel.org/

[MINIOS] "Mini-OS - Xen," http://wiki.xenproject.org/wiki/Mini-OS

[OSV] "OSv - The Operating System Designed for the Cloud," http://osv.io/

[PGRANTS] http://lists.xenproject.org/archives/ html/xen-devel/2015- 05/msg01498.html

[SELINUX] "Security Enhanced Linux (SELinux) project," http://selinuxproject.org/

[SUPERFLUIDITY] "The Case for the Suplerfluid Cloud," F. Manco, J. Martins, K. Yasukata, J. Mendes, S. Kuenzer, and F. Huici. USENIX HotCloud 2015

[APPARMOR] "Mandatory Access Control Framework," https://wiki.debian.org/AppArmor

[VCPE-CONTAINER-PERF] "Virtual Customer CPE Container Performance White Paper," http://info.ixiacom.com/rs/098-FRB-840/images/Calsoft-Labs-CaseStudy2015.pdf

Authors' Addresses

Sriram Natarajan
Google
natarajan.sriram@gmail.com

Ram (Ramki) Krishnan
Dell
ramki_krishnan@dell.com

Anoop Ghanwani
Dell
anoop@alumni.duke.edu

Dilip Krishnaswamy
IBM Research
dilikris@in.ibm.com

Peter Willis
BT
peter.j.willis@bt.com

Ashay Chaudhary
Verizon
the.ashay@gmail.com

Felipe Huici
NEC
felipe.huici@neclab.eu