CONTAINERS / DEVOPS / KUBERNETES / MICROSERVICES

How Hypernetes Brings Multi-tenancy to Microservice Architectures

23 Nov 2015 10:12am, by Thibault Bronchain

Editor's Note: The following is a contributed post from Thibault Bronchain, the developers ambassador for Hypernetes, which explains how this open source software for running secure multi-tenant Docker containers, based on the Hyper hypervisor-based Docker engine.

Microservices is an architecture where each functions of an application are separated from each other and communicate through APIs and protocols, often through HTTP REST APIs, or WebSockets. It differs from monolithic applications, where all the functions of an application are located in the same code.

Debates exist on whether using network sockets and HTTP protocols to communicate are the best options, as they could easily become a bottleneck with some architecture. However, when well designed, microservices bring some non-negligible advantages:

- Ease of development: once designed, each feature can be distinctly separated to each other, helping groups of developers to focus on their work, and not being influenced by other parts
- Flexible scaling: when it comes to scaling, microservices express their real strength. In monolithic infrastructures, whenever better performances are needed, a new "instance" of the full app is started, doubling the power, in theory. In practice, it is like buying a second identical house when you need an additional room. With microservices, it is possible to scale only the part that require to be scaled, optimizing the cost and general efficiency of the infrastructure.

Containers Got It Right

A first step when designing a microservices architecture is to separate each component, or role, of the whole architecture into small, independent pieces.

Containers, such as Docker, can help, in a number of ways:

- Philosophically, by separating each component of an application in a distinct container.

- Practically, by packaging these components and their necessary libraries into ready-to-launch and easy to distribute packages: the container images.

Despite the strengths of Docker containers, their runtime execution architecture has a security weakness. Docker containers execution relies on runc (formerly *libcontainer*, and originally LXC), which is a set of binding and optimizations around Linux kernel isolation technologies (cgroup, namespaces...). They provide a great isolation for many applications, however, they are a risky choice in the case of multi-tenants environments. All containers on the same host share the same Linux kernel with that host, which first enlarges the possible attack surface, but also limits the running applications to one common unique kernel.

In other words, Docker containers as we know them are great microservices packaging solutions, however, they lack critical runtime features for multi-tenants environments.

Multi-Tenancy

If we strictly focus on the isolation part, two approaches are feasible to ensure a better security of a multi-tenants container-based infrastructure.

- By providing rock-tested containers ourselves; creating a PaaS where users can deploy their application by providing their source code. This is the kind of approach Heroku is using. In that case, because users don't have the access of the runtime, the risk of security holes and issues is somewhat reduced. However, the privilege escalating risks are still present, if the runtime environment gets compromised. Also, limiting the choice of runtime available implies restrictions and eventually constraints from a user's perspective.
- To let any users deploy their own containers, without any control, it is necessary to provide a truly underlying isolated environment. A common approach, that we can see in Google Container Engine, or AWS ECS, is to let each user build its own cluster of virtual machine, on which containers can be scheduled with the help of a scheduler (Kubernetes, Mesos, Swarm).

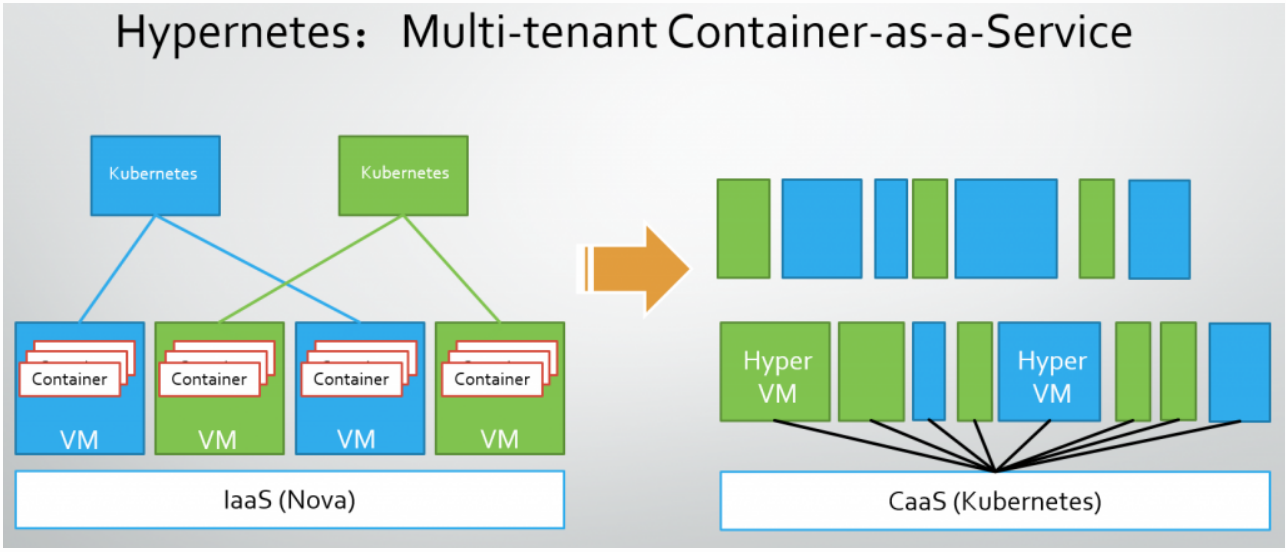
The problem here — and especially with the second option — is when it comes to scaling. Today's clusters are composed of identical VMs scaled up and down on demand. However, this approach may lead to the loss of one of microservices' strengths: the flexible scaling. In the current clustering approach, each machine has to meet the requirements of the most "expensive" containers (storage, memory, CPU). Doing this,

providing a new machine for a container requiring more RAM will lead to a waste of storage, a machine for a database container to a waste of computing resources, and so on. Otherwise, you may have to create VMs according to the container size (DB instance for DB containers, cache instance for cache container, etc.), and make sure your scheduler place the proper containers on the proper VM. Then you face the question: "Why do I need this scheduler?"

Furthermore, if we look at the customization bit, then we realize there is no truly flexible solution to define or tweak a specific kernel version for your containers. This may be a real drawback in some applications (like Network Functions Virtualization), where new and mature technologies constantly mix with each other.

Hypernetes: The Multi-Tenant Kubernetes Distro

To address this dilemma, we have built Hypernetes as a distribution of Kubernetes, the containers' scheduler of Google Container Service. Kubernetes is an open source orchestration system for Docker containers. It handles scheduling onto nodes in a compute cluster and actively manages workloads to ensure that their state matches the user's declared intentions. Using the concepts of "labels" and "pods", it groups the containers which make up an application into logical units for easy management and discovery.



We believe Kubernetes as being to future of containers scheduling and orchestration, by its ease of use, reliability, power and contributors community.

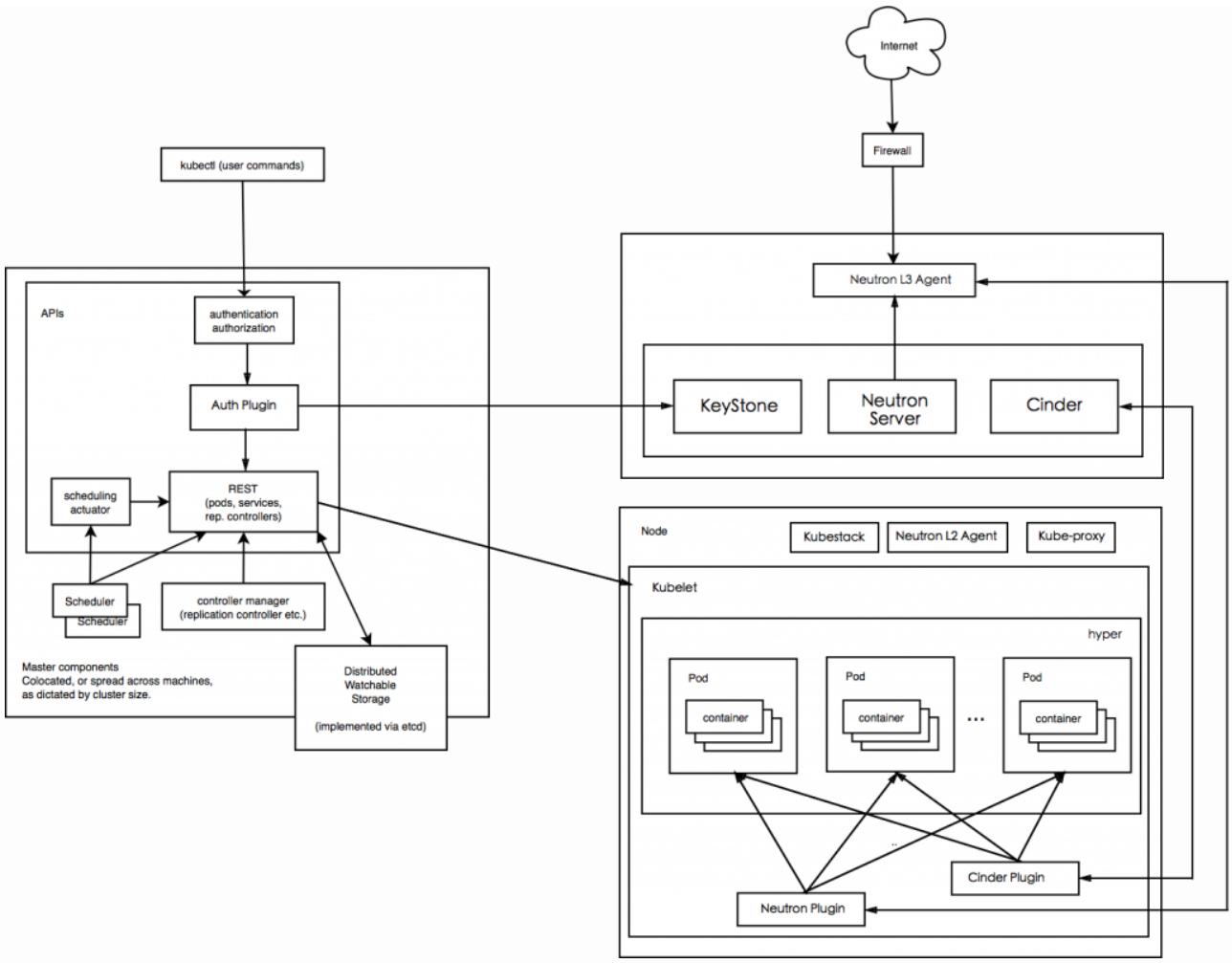
Hypernetes is based on Hyper, the hypervisor-based Docker engine. Hyper allows running Docker images with any hypervisor (KVM, Xen, Vbox, ESX). Hyper is different from the minimalist Linux distributions like CoreOS by the fact that Hyper runs on the physical box and loads the Docker images from the metal into VM instances created on-

demand, in which no guest OS is present. Instead of virtualizing a complete operating system, Hyper boots a minimalist kernel in the VM to host the Docker images. In short, your Pod is a virtual machine.

With this approach, Hyper is able to bring some encouraging merits:

- 300ms to boot a new HyperVM instance with a Pod of Docker images.
- 20MB for minimal memory footprint of a HyperVM instance.
- Immutable HyperVM, only kernel + images, serving as atomic unit (Pod) for scheduling.
- Immune from the shared kernel problem in LXC – i.e. isolated by VM.
- Work seamlessly with OpenStack components, e.g. Neutron, Cinder, due to the nature of a hypervisor.
- BYOK, bring-your-own-kernel is somewhat mandatory for a public cloud platform.

Hyper (Hyperd) runs directly on all your bare-metal machines to provision HyperVM (with Docker images) in a millisecond. Within the VM, the "Hyperstart" init process is launched on top of the Hyper Kernel (or any compatible Linux kernel) to run Docker images as Pod. The Kubelet agent runs on each bare-metal host and manages HyperVM with Hyperd's APIs. The formed cluster of "Kubelets" is managed with the help of a "Kubernetes Master" server.



The Hypernetes Architecture.

We just see that Hypernetes "Pods" are virtual machines (VMs). VMs are not constructed the same was as Linux containers, and management of external peripherals (networking, storage, and so on) cannot be done in the same way it is done with containers.

To ensure a reliable and powerful networking and storage management, Hypernetes relies on two major OpenStack components: Cinder coupled with Ceph for storage and Neutron for networking. In addition, Hypernetes uses KeyStone for identities management, enabling multi-tenants capabilities.

The big strength of relying on virtualization and OpenStack components is their modularity, reliability, and interoperability.

Virtualization technologies have been used for years to build reliable systems, and success in large-scale deployments shows us its magnitude. In addition, by speaking the OpenStack language, Hypernetes enable the combination of already existing and deployed OpenStack additions (i.e. networking plugins) with containers. In other words: don't throw away the great parts of your current architecture!