Tuesday, May 24, 2016

Hypernetes: Bringing Security and Multi-tenancy to Kubernetes (https://kubernetes.io/blog/2016/05/hypernetes-security-and-multi-tenancy-in-kubernetes/)

Today's guest post is written by Harry Zhang and Pengfei Ni, engineers at HyperHQ, describing a new hypervisor based container called HyperContainer

While many developers and security professionals are comfortable with Linux containers as an effective boundary, many users need a stronger degree of isolation, particularly for those running in a multi-tenant environment. Sadly, today, those users are forced to run their containers inside virtual machines, even one VM per container.

Unfortunately, this results in the loss of many of the benefits of a cloud-native deployment: slow startup time of VMs; a memory tax for every container; low utilization resulting in wasting resources.

In this post, we will introduce HyperContainer, a hypervisor based container and see how it naturally fits into the Kubernetes design, and enables users to serve their customers directly with virtualized containers, instead of wrapping them inside of full blown VMs.

HyperContainer

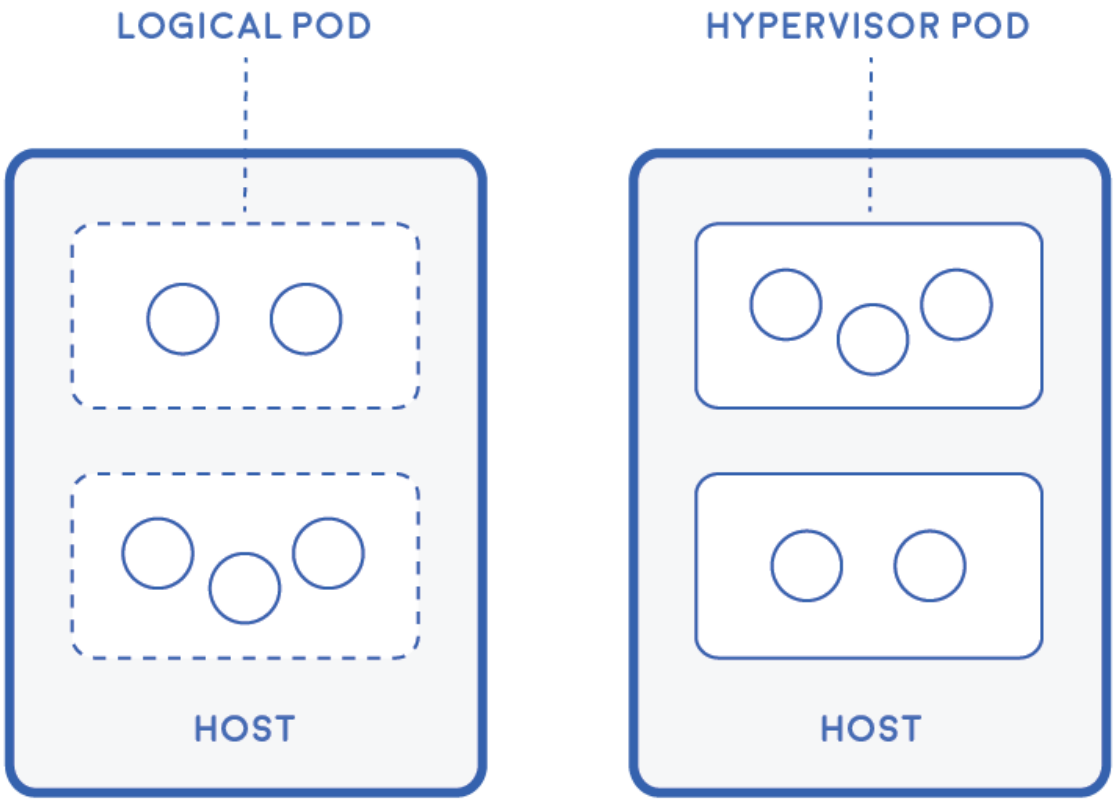
HyperContainer (http://hypercontainer.io/) is a hypervisor-based container, which allows you to launch Docker images with standard hypervisors (KVM, Xen, etc.). As an open-source project, HyperContainer consists of an OCI (https://github.com/opencontainers/runtime-spec) compatible runtime implementation, named runV (https://github.com/hyperhq/runv/), and a management daemon named hyperd (https://github.com/hyperhq/hyperd). The idea behind HyperContainer is quite straightforward: to combine the best of both virtualization and container.

We can consider containers as two parts (as Kubernetes does). The first part is the container runtime, where HyperContainer uses virtualization to achieve execution isolation and resource limitation instead of namespaces and cgroups. The second part is the application data, where HyperContainer leverages Docker images. So in HyperContainer, virtualization technology makes it possible to build a fully isolated sandbox with an independent guest kernel (so things like `top` and /proc all work), but from developer's view, it's portable and behaves like a standard container.

HyperContainer as Pod

The interesting part of HyperContainer is not only that it is secure enough for multi-tenant environments (such as a public cloud), but also how well it fits into the Kubernetes philosophy.

One of the most important concepts in Kubernetes is Pods. The design of Pods is a lesson learned (Borg paper section 8.1 (http://static.googleusercontent.com/media/research.google.com/zh-CN//pubs/archive/43438.pdf)) from real world workloads, where in many cases people want an atomic scheduling unit composed of multiple containers (please check this example (https://github.com/kubernetes/examples/tree/master/staging/javaweb-tomcat-sidecar) for further information). In the context of Linux containers, a Pod wraps and encapsulates several containers into a logical group. But in HyperContainer, the hypervisor serves as a natural boundary, and Pods are introduced as first-class objects:



HyperContainer wraps a Pod of light-weight application containers and exposes the container interface at Pod level. Inside the Pod, a minimalist Linux kernel called HyperKernel is booted. This HyperKernel is built with a tiny Init service called HyperStart. It will act as the PID 1 process and creates the Pod, setup Mount namespace, and launch apps from the loaded images.

This model works nicely with Kubernetes. The integration of HyperContainer with Kubernetes, as we indicated in the title, is what makes up the Hypernetes (https://github.com/hyperhq/hypernetes) project.

Hypernetes

One of the best parts of Kubernetes is that it is designed to support multiple container runtimes, meaning users are not locked-in to a single vendor. We are very pleased to announce that we have already begun working with the Kubernetes team to integrate HyperContainer into Kubernetes upstream. This integration involves:

- container runtime optimizing and refactoring
- new client-server mode runtime interface
- containerd integration to support runV

The OCI standard and kubelet's multiple runtime architecture make this integration much easier even though HyperContainer is not based on Linux container technology stack.

On the other hand, in order to run HyperContainers in multi-tenant environment, we also created a new network plugin and modified an existing volume plugin. Since Hypernetes runs Pod as their own VMs, it can make use of your existing IaaS layer technologies for multi-tenant network and persistent volumes. The current Hypernetes implementation uses standard Openstack components.

Below we go into further details about how all those above are implemented.

Identity and Authentication

In Hypernetes we chose Keystone (http://docs.openstack.org/developer/keystone/) to manage different tenants and perform identification and authentication for tenants during any administrative operation. Since Keystone comes from the OpenStack ecosystem, it works seamlessly with the network and storage plugins we used in Hypernetes.

Multi-tenant Network Model

For a multi-tenant container cluster, each tenant needs to have strong network isolation from each other tenant. In Hypernetes, each tenant has its own Network. Instead of configuring a new network using OpenStack, which is complex, with Hypernetes, you just create a Network object like below.

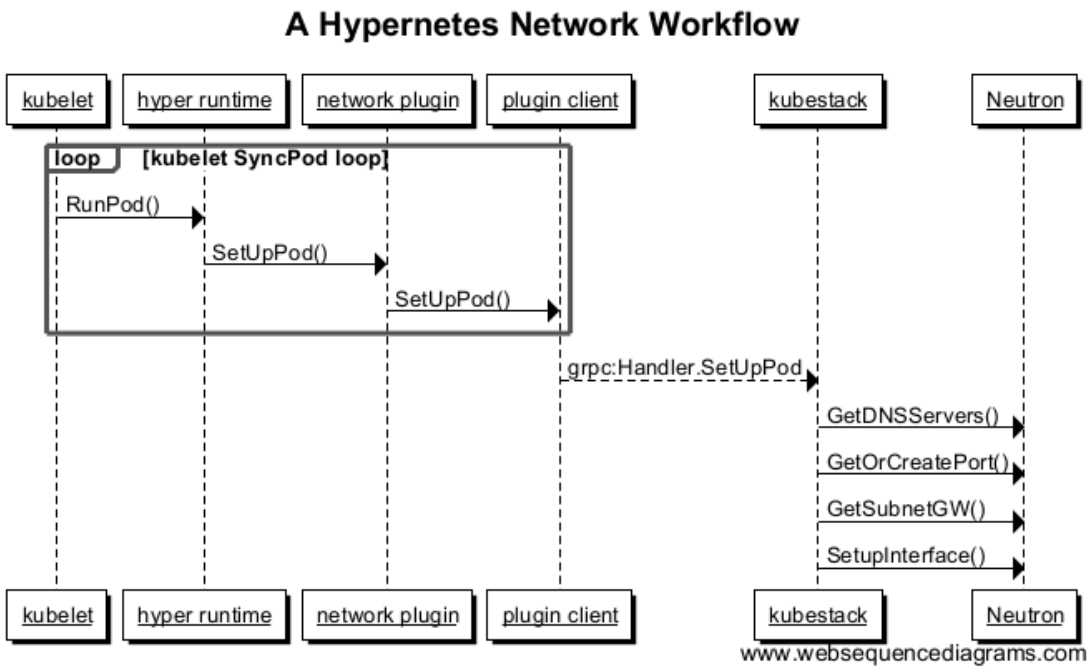
```
apiVersion: v1
kind: Network
metadata:
  name: net1
spec:
  tenantID: 065f210a2ca9442aad898ab129426350
  subnets:
    subnet1:
      cidr: 192.168.0.0/24
      gateway: 192.168.0.1
```

Note that the tenantID is supplied by Keystone. This yaml will automatically create a new Neutron network with a default router and a subnet 192.168.0.0/24.

A Network controller will be responsible for the life-cycle management of any Network instance created by the user. This Network can be assigned to one or more Namespaces, and any Pods belonging to the same Network can reach each other directly through IP address.

```
apiVersion: v1
kind: Namespace
metadata:
  name: ns1
spec:
  network: net1
```

If a Namespace does not have a Network spec, it will use the default Kubernetes network model instead, including the default kube-proxy. So if a user creates a Pod in a Namespace with an associated Network, Hypernetes will follow the Kubernetes Network Plugin Model (http://kubernetes.io/docs/admin/network-plugins/) to set up a Neutron network for this Pod. Here is a high level example:



{: HyperContainer wraps a Pod of li.big-img}

Hypernetes uses a standalone gRPC handler named kubestack to translate the Kubernetes Pod request into the Neutron network API. Moreover, kubestack is also responsible for handling another important networking feature: a multi-tenant Service proxy.

In a multi-tenant environment, the default iptables-based kube-proxy can not reach the individual Pods, because they are isolated into different networks. Instead, Hypernetes uses a built-in HAproxy in every HyperContainer (https://github.com/hyperhq/hyperd/blob/2072dd8e28a02a25ae6a819f81029b47a579e683/servicediscovery/servicediscovery.go) as the portal. This HAproxy will proxy all the Service instances in the namespace of that Pod. Kube-proxy will be responsible for

updating these backend servers by following the standard OnServiceUpdate and OnEndpointsUpdate processes, so that users will not notice any difference. A downside of this method is that HAproxy has to listen to some specific ports which may conflicts with user's containers.That's why we are planning to use LVS to replace this proxy in the next release.

With the help of the Neutron based network plugin, the Hypernetes Service is able to provide an OpenStack load balancer, just like how the "external" load balancer does on GCE. When user creates a Service with external IPs, an OpenStack load balancer will be created and endpoints will be automatically updated through the kubestack workflow above.

Persistent Storage

When considering storage, we are actually building a tenant-aware persistent volume in Kubernetes. The reason we decided not to use existing Cinder volume plugin of Kubernetes is that its model does not work in the virtualization case. Specifically:

The Cinder volume plugin requires OpenStack as the Kubernetes provider.

The OpenStack provider will find on which VM the target Pod is running on

Cinder volume plugin will mount a Cinder volume to a path inside the host VM of Kubernetes.

The kubelet will bind mount this path as a volume into containers of target Pod.

But in Hypernetes, things become much simpler. Thanks to the physical boundary of Pods, HyperContainer can mount Cinder volumes directly as block devices into Pods, just like a normal VM. This mechanism eliminates extra time to query Nova to find out the VM of target Pod in the existing Cinder volume workflow listed above.

The current implementation of the Cinder plugin in Hypernetes is based on Ceph RBD backend, and it works the same as all other Kubernetes volume plugins, one just needs to remember to create the Cinder volume (referenced by volumeID below) beforehand.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - name: nginx-persistent-storage
      mountPath: /var/lib/nginx
  volumes:
  - name: nginx-persistent-storage
    cinder:
      volumeID: 651b2a7b-683e-47e1-bdd6-e3c62e8f91c0
      fsType: ext4
```

So when the user provides a Pod yaml with a Cinder volume, Hypernetes will check if kubelet is using the Hyper container runtime. If so, the Cinder volume can be mounted directly to the Pod without any extra path mapping. Then the volume metadata will be passed to the Kubelet RunPod process as part of HyperContainer spec. Done!

Thanks to the plugin model of Kubernetes network and volume, we can easily build our own solutions above for HyperContainer though it is essentially different from the traditional Linux container. We also plan to propose these solutions to Kubernetes upstream by following the CNI model and volume plugin standard after the runtime integration is completed.

We believe all of these open source projects (https://github.com/hyperhq/) are important components of the container ecosystem, and their growth depends greatly on the open source spirit and technical vision of the Kubernetes team.

Conclusion

This post introduces some of the technical details about HyperContainer and the Hypernetes project. We hope that people will be interested in this new category of secure container and its integration with Kubernetes. If you are looking to try out Hypernetes and HyperContainer, we have just announced the public beta of our new secure container cloud service (Hyper_ (https://hyper.sh/)), which is built on these technologies. But even if you are running on-premise, we believe that Hypernetes and HyperContainer will let you run Kubernetes in a more secure way.

~Harry Zhang and Pengfei Ni, engineers at HyperHQ