More Than Secure: Containerd + KataContainers as Kubernetes Runtime

Lei Zhang, Alibaba Group & Fupan Li, HyperHQ
Kubernetes

Control Panel

api-server

Etcnd

pod, node list

bind

Scheduling

Workloads Orchestration

kubelet

Node

kubelet

Node

kubelet

Node
Kubernetes + containerd

kubelet

containerd

runC

clone(), setns(), pivot_root()

Linux Kernel

C C C C C C

Node
Linux Container

• **Container Runtime**
  - The dynamic view and boundary of your running process
  - Namespace + Cgroups

• **Container Image**
  - The static view of your program, data, dependencies, files and directories
  - rootfs

FROM busybox
ADD temp.txt /
VOLUME /data
CMD [“echo hello”]
KataContainers

- **Container Runtime**
  - Each Pod is hypervisor isolated
    - Independent guest kernel
  - Secure as VM
  - Fast as container
- **Container Image**
  - Same as Linux container
Container Security

- Linux container
  - Dropping Linux capabilities
  - Read-only mount points
  - Mandatory access controls (MAC)
    - SELinux & AppArmor
  - Dropping syscalls
    - SECCOMP
  - In 99.99% cases
    - wrap containers in VMs

- KataContainers
  - Hardware virtualization
  - Independent Linux instance per Pod
    - e.g. run Linux 3.16 container on a Linux 4.0 host
Kubernetes + KataContainers

kubelet

???

KataContainers

virtualization

Linux Kernel

Node

VM VM VM VM VM
Container Runtime Interface (CRI)

- Describe what kubelet expects from container runtimes
- Imperative container-centric interface
  - why not pod-centric?
    - Every container runtime implementation needs to understand the concept of pod.
    - Interface has to be changed whenever new pod-level feature is proposed.
CRI Spec

• Sandbox

  • How to isolate Pod environment?
    • Linux container: infra container + pod level cgroups
    • Kata: light-weighted VM

• Container

  • Linux container: namespace + cgroups
  • Kata: namespace containers controlled by hyperstart
How CRI Works

kubelet

SyncLoop

kubelet

SyncPod

GenericRuntime

CRI Spec

CRI grpc

Sandbox
Create
Delete
List
Container
Start
Exec
Image
Pull
List

api-server

Etcd

Management

pod, node list

bind

Workloads
Orchestration

Scheduling

pod

CRI shim

docker

client api

dockershim

remote
(no-op)

Container Runtime
How CRI shim works

1. RunPodSandbox(foo)
2. CreatContainer(A)
3. StartContainer(A)
4. CreatContainer(B)
5. StartContainer(B)

$ kubectl run foo ...

Node

Pod foo

container A

container B

foo (vm)

A

B

docker runtime

vm runtime

CNI add()
Wrap Up

Node

Linux Kernel

CRI shim

KataContainers

syscall

kublet

CRI

Do your work here!

KataContainers
Use containerd/cri as CRI shim

Kata-runtime cli

If vsock is available, no proxy here.

Pod sandbox

container

container

container

container
But …

• Too many containerd-shim, large resource footprint

• CRI is a well-defined interface for Kubernetes to consume, not for runtimes
  • gVisor/KataContainers/VM
    • Un-match to existing CRI shims

• Maintenance nightmare
  • e.g. cri-o VS cri-containerd + gVisor/KataContainers/VM-based runtimes, oh my …
Containerd ShimV2

- A “standard interface” between CRI shim and container runtime!
  - CRI -> containerd -> OCI runtime
  - CRI -> containerd -> shimV2 -> OCI runtime
What’s the difference?

• Previous:
  • Call `containerd-shim`
    • This will start a shim process per container

• Now:
  • Call `containerd-shim start`
    • Implement “start” operation as you wish:
      • Start containerd-shim when creating sandbox
      • Reuse existing containerd-shim when creating container
Containerd + ShimV2 + KataContainers
Containerd Shim V2

Kubelet

Cri-containerd

Kata-Containerd-shimv2

KataContainers

Virtualization

Linux Kernel

Node

Do your work here!
// Create a new sandbox or container with the underlying OCI runtime
func (s *service) Create(ctx context.Context, r *taskAPI.CreateTaskRequest) (_ *taskAPI.CreateTaskResponse, err error) {
    return nil, erdefs.ErrNotImplemented
}

// Start a process
func (s *service) Start(ctx context.Context, r *taskAPI.StartRequest) (*taskAPI.StartResponse, error) {
    return nil, erdefs.ErrNotImplemented
}

// Delete the initial process and container
func (s *service) Delete(ctx context.Context, r *taskAPI.DeleteRequest) (*taskAPI.DeleteResponse, error) {
    return nil, erdefs.ErrNotImplemented
}

// Exec an additional process inside the container
func (s *service) Exec(ctx context.Context, r *taskAPI.ExecProcessRequest) (*ptypes.Empty, error) {
    return nil, erdefs.ErrNotImplemented
}

// ResizePty of a process
func (s *service) ResizePty(ctx context.Context, r *taskAPI.ResizePtyRequest) (*ptypes.Empty, error) {
    return nil, erdefs.ErrNotImplemented
}

// State returns runtime state information for a process
func (s *service) State(ctx context.Context, r *taskAPI.StateRequest) (*taskAPI.StateResponse, error) {
    return nil, erdefs.ErrNotImplemented
}

// Pause the container
func (s *service) Pause(ctx context.Context, r *taskAPI.PauseRequest) (*ptypes.Empty, error) {
    return nil, erdefs.ErrNotImplemented
}

// Resume the container
func (s *service) Resume(ctx context.Context, r *taskAPI.ResumeRequest) (*ptypes.Empty, error) {
    return nil, erdefs.ErrNotImplemented
}

// Kill a process with the provided signal
func (s *service) Kill(ctx context.Context, r *taskAPI.KillRequest) (*ptypes.Empty, error) {
    return nil, erdefs.ErrNotImplemented
}

// Pids returns all pids inside the container
func (s *service) Pids(ctx context.Context, r *taskAPI.PidsRequest) (*taskAPI.PidsResponse, error) {
    return nil, erdefs.ErrNotImplemented
}
Live Demo

• Kubernetes + containerd + shimV2 + KataContainers
  1. kubeadm installed, 3 nodes cluster on GCE, nested virtualization
  2. Pod lifecycle
  3. Independent kernel
     1. No kernel sharing with host
  4. Strong isolation
     1. e.g. forkbomb
  5. High density, small footprint
     1. 100 KataContainers in one GCE Node in 2mins
Real Case

- 1.5 Engineers + 1 GSoC student
- Pull Request
  - https://github.com/kata-containers/runtime/pull/572
  - Expected to be merged in next 2 weeks
Read Our Story

GSoC 18: Kata Containers support for containerd

My story started in January 2018 when I noticed that the CNCF community had some container technology-related ideas for GSoC 2018. After browsing the topics I was very interested in the topic of "KataContainers support for containerd". I already had some knowledge about Kubernetes, so I spent quite some time to deeply study documentation and code for "containerd" and "KataContainers". This helped me to gain a much better understanding of these two projects. I drafted a design proposal and was finally selected as a GSoC candidate. I believe the study of these open-source projects helped a lot in my design proposal.

In the Kata/CR native manner, the theory was that we could avoid using too many independent shim and proxy processes. Some investigations showed a shim consuming too much memory, causing huge overhead costs in high-density cases. So our project was aimed at removing independent shim and proxy processes to save memory, and start the container more quickly.

In the beginning of the project, we were going to develop a kata-runtime plugin for containerd. When we implemented part of the interfaces that containerd runtime plugin mocked, the kata container operations worked successfully. That moment was engraved on my mind. It meant that I went from a container user to a container developer. So excited!
Thank You!
Lei Zhang, Alibaba Group & Fupan Li, HyperHQ