How Container Runtimes matter in Kubernetes?
About me

- Works @ NTT Open Source Software Center
- Contributes to containerd and other related projects.
- Docker community leader, Tokyo

@kunalkushwaha
Agenda

- Kubernetes Architecture.
- What is CRI (Container Runtime Interface)
- What is OCI (Open Container Initiative)
- CRI & OCI Implementations
- Why runtimes affect Kubernetes.
- Runtime Benchmarking results
- Analyzing for various workloads
- Summary
Kubernetes Architecture

A typical Kubernetes cluster
- `kubectl` is tool for user to interact with k8s cluster.
- Master node interpret the command and if required interact with worker nodes.
Master Node Overview

Important components of Kubernetes Master Node
Master Node Control Flow

- API Server plays a central part for cluster communication
- etcd store all definition of kubernetes resources
- Scheduler and Control Manager push commands for workers via API Server
Kubernetes Architecture

User

kubectl

Kubernetes Master

Kubernetes Worker #1

Kubernetes Worker #n
Important components of Kubernetes Worker Node
Kubernetes Worker Control Flow

- Kubelet is the primary Node agent. API Server talks to Kubelet.
- Service Proxy enables user to access applications running on node.
- Docker running on node is used for creating Pods.
Kubernetes Worker Control Flow

- Kubelet is the primary Node agent. API Server talks to Kubelet.
- Service Proxy enables user to access applications running on node.
- Docker running on node is used for creating Pods.
With alternative container runtimes, Kubelet code gets bloated to support each.
Container Runtime Interface

- Introduced in Kubernetes 1.5 *(2016)*
- Interfaces for gRPC service for Runtime & Image Management
- Container centric interfaces
- Pod containers as Sandbox containers
- Current status: v1alpha2

*https://github.com/kubernetes/kubernetes/blob/release-1.5/docs/proposals/container-runtime-interface-v1.md*
Kubelet with CRI

CRI solves supporting various runtime alternatives with no change in Kubelet
Container Runtime

Kubernetes Master

Kubelet

CRI

Container Runtime

Kubernetes Worker

Shim
What is Container Runtime

- Provides core primitives to manage containers on host
- Container execution & supervision
- Network Interfaces and management
- Image management
- Manage local storage
- e.g. LXC, Docker, rkt
Open Container Initiative

- Container runtime & Image specification
- Runtime specs define input to create a container
- Multiple platform supported (Linux, Windows, Solaris & VM)
- `runc` is default implementation of OCI Runtime Specs
- Current Runtime Specs status: v1.0.1
Gap between Kubelet & OCI runtime

Kubelet Requirements for Runtime

- Manage images (pull / push / rm ..)
- Talks CRI / gRPC
- Prepare environment to successfully instantiate container.
- Prepare network for pod

OCI Runtime

- Do not understand concept of image
- Input is OCI specs (json and rootfs)
- Consume the rootfs and container config file (json)
- Attach network as pre-start hook.
Runtime in Kubernetes

Apart from OCI, another runtime component is required
Runtime in Kubernetes

- High level runtime implement CRI gRPC services
- Take care of all prerequisite to successfully operate OCI runtimes
Runtime in Kubernetes

- OCI runtime works as low-level runtime
- High-level runtime provides inputs to OCI runtime as per OCI Specs
CRI Implementations

- Dockershim
- CRI-O
- Containerd
- Frakti
- rktlet
Dockershim

- Embedded into Kubelet.
- Dockershim talks to docker, which manage pods.
- Default CRI implementation & enjoy majority in current kubernetes deployments
**CRI-O**

- CRI-O reduces the one extra hop from docker.
- CRI-O uses CNI for providing networking to pods.
- Monolithic design (understands CRI and outputs OCI compatible)
- Works with all OCI runtimes.
containerD

- containerD, with revised scope eliminates the extra hop required by docker.
- Redesigned storage drivers for simplicity and better performance.
- Extensible design, CRI service runs as plugin.
- Uses CNI for networking
- Works with all OCI runtimes.
- Frakti runtime was designed to support VM based runtime to kubernetes.
- It supports mixed runtimes
  - Linux containers for privilege containers and runV containers for rest
  - Though uses dockershim to use linux containers, result into extra hops
- Also supports Unikernels
Frakti v2- Coming soon

- Frakti v2 will be implemented as runtime plugin for containerD.
- Reduce extra hops and implementation effort too.
OCI Runtimes

- runC
  - Default OCI specs implementation
  - Isolation based on Namespace, cgroups, secomp & MAC (AppArmor, SELinux)

- runV

- Clear Containers

- kata-runtime

- gVisor
## OCI Runtimes

<table>
<thead>
<tr>
<th>Runtimes</th>
<th>Details</th>
</tr>
</thead>
</table>
| **runC** | - Default OCI specs implementation  
- Isolation based on Namespace, cgroups, secomp & MAC (AppArmor, SELinux) |
| **runV** | - OCI compliant VM based runtime  
- Uses optimized qemu & KVM.  
- A light weight guest kernel is used. |
| **Clear Containers** |  |
| **kata-runtime** |  |
| **gVisor** |  |
## OCI Runtimes

<table>
<thead>
<tr>
<th>Runtime</th>
<th>Description</th>
</tr>
</thead>
</table>
| runC             | - Default OCI specs implementation  
|                  | - Isolation based on Namespace, cgroups, secomp & MAC (AppArmor, SELinux)                        |
| runV             | - OCI compliant VM based runtime  
|                  | - Uses qemu & KVM.  
|                  | - A light weight guest kernel is used.                                                            |
| Clear Containers | - Hardware-virtualized containers using Intel's VT-x  
|                  | - Utilize DAX “direct access” feature of 4.0 kernel                                               |
| kata-runtime     |                                                                                                  |
| gVisor           |                                                                                                  |
OCI Runtimes

- **runC**
  - Default OCI specs implementation
  - Isolation based on Namespace, cgroups, secomp & MAC (AppArmor, SELinux)

- **runV**
  - OCI compliant VM based runtime
  - Uses qemu & KVM.
  - A light weight guest kernel is used.

- **Clear Containers**
  - Hardware-virtualized containers using Intel's VT-x
  - Utilize DAX “direct access” feature of 4.0 kernel

- **kata-runtime**
  - Best of runV & cc-containers
  - 1.0 Release (22nd May, 2018)
  - Under active development

- **gVisor**
## OCI Runtimes

<table>
<thead>
<tr>
<th>RunTime</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>runC</td>
<td>Default OCI specs implementation, isolation based on Namespace, cgroups, secomp &amp; MAC (AppArmor, SELinux)</td>
</tr>
<tr>
<td>runV</td>
<td>OCI compliant VM based runtime, uses qemu &amp; KVM, light weight guest kernel is used.</td>
</tr>
<tr>
<td>Clear Containers</td>
<td>Hardware virtualized containers using Intel's VT-x, utilize DAX “direct access” feature of 4.0 kernel</td>
</tr>
<tr>
<td>kata-runtime</td>
<td>Best of runV &amp; cc-containers, 1.0 Release (22&lt;sup&gt;nd&lt;/sup&gt; May, 2018), under active development</td>
</tr>
<tr>
<td>gVisor</td>
<td>Sandbox based containers, it intercepts application system call, acts like kernel, similar approach as User Mode Linux (UML), under active development</td>
</tr>
</tbody>
</table>
Final candidates for Evaluation

High-level Runtime

Dockershim
CRI-O
containerD

Low-level Runtime

runC
runV
Kata containers
clear containers
Why runtimes affect kubernetes
- Kubernetes offers variety of choices to tune the system
Kubernetes offers variety of choices to tune the system
- Once rest of components finalized
  - for deployment and management runtime is only variable factor.
  - For application performance only low level runtime matters.
Performance benchmarking

Application deployment performance

- Container operations (Create, start, stop, remove)

Application Performance

- Containerization / Virtualization overhead.
Performance benchmarking process

- Prerequisite:
  - Pull Sandbox Image
  - Pull Container Image (ubuntu:latest)

Benchmark Environment

- Architecture: x86_64
- CPU(s): 8
- Core(s) per socket: 4
- Model name: i7-3630QM CPU @ 2.40GHz
- Virtualization: VT-x
- Kernel: linux 4.15
- OS: Ubuntu

Create
- Create & Run PodSandbox
  - Create Application Container

Start
- Start Application Container

Stop
- Stop Application Container

Delete
- Delete Application Container
  - Stop PodSandbox
  - Delete PodSandbox

4 Threads x 50

- Rootfs prepared from Image
- Writable area for container
- CNI plugin invocation for Network
# runC Performance

## Software versions

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Containerd</td>
<td>v1.1.0</td>
</tr>
<tr>
<td>cri-o</td>
<td>v1.10.1</td>
</tr>
<tr>
<td>Docker</td>
<td>18.05.0.ce</td>
</tr>
<tr>
<td>Runc</td>
<td>v1.0</td>
</tr>
<tr>
<td></td>
<td>git #69663f0bd4b</td>
</tr>
</tbody>
</table>

## Performance

The chart shows a comparison of the times taken for various operations among Containerd, cri-o, and Docker. Here are some key observations:

- **Create**: The times range from 0.73 seconds for Docker to 0.26 seconds for Containerd.
- **Start**: The times range from 0.75 seconds for Docker to 0.03 seconds for cri-o.
- **Stop**: The times range from 0.58 seconds for Containerd to 0.24 seconds for cri-o.
- **Delete**: The times range from 0.53 seconds for Docker to 0.27 seconds for Containerd.

### Performance differences

- **Performance difference due to high level runtime**
- **Low-level runtime (runC) is constant in all**
- **cri-o and docker share same graph driver design, could be reason for high create time.**
- **containerD perform better in almost all case.**
## Latency with runC

<table>
<thead>
<tr>
<th></th>
<th>containerd</th>
<th>cri-o</th>
<th>dockershim</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time to start</strong></td>
<td>0.46</td>
<td>0.76</td>
<td>1.05</td>
</tr>
<tr>
<td><strong>Time to stop</strong></td>
<td>0.46</td>
<td>0.77</td>
<td>0.77</td>
</tr>
</tbody>
</table>

**Time before application start running in runC container**

**Time before resources are released after application stops**

Less is better

- **cri-o & containerD both perform better than docker**
- **In performance, containerD performs better than cri-o**
Kata-runtime Performance

Software versions

- Containerd : v1.1.0
- cri-o : v1.10.1
- Docker : 18.05.0.ce
- kata-runtime: v1.0

Difference is mainly due to high level runtime performance.

- Bug in Stop logic, while invoked through CRI
  - Takes < 2 seconds, if done directly through docker or containerD
Latency with Kata

Latency with kata-container is comparable with all high-level runtimes.

High-level Runtime don’t make much difference if low-level runtime consume most

Less is better
kata vs runV vs clear-containers

Software versions

- **Containerd**: v1.1.0
- **Docker**: 18.0.0.0
- **Frakti**: v1.10.0
- **runV**: v1.0.0

Stop function of cc-containers & runV looks normal. Hence fix required for kata containers.

Kata containers performance is in-between runV and cc-runtime.
## Latency with VM based runtimes

<table>
<thead>
<tr>
<th>Runtime</th>
<th>Time to Start</th>
<th>Time to Stop</th>
</tr>
</thead>
<tbody>
<tr>
<td>kata + containerd</td>
<td>2.27</td>
<td>1.29</td>
</tr>
<tr>
<td>cc-container + containerd</td>
<td>3.02</td>
<td>2.3</td>
</tr>
<tr>
<td>runV + frakti</td>
<td>10.65</td>
<td>0.86</td>
</tr>
</tbody>
</table>

- **Time to start**: kata + containerd takes 2.27 seconds, cc-container + containerd takes 3.02 seconds, runV + frakti takes 10.65 seconds.
- **Time to stop**: kata + containerd takes 1.29 seconds, cc-container + containerd takes 2.3 seconds, runV + frakti takes 0.86 seconds.

Less is better. Kata is still in active development.

runV performs for container operations is best in VM containers.
Performance Overhead – Low-level runtimes

I/O Throughput

More is better

<table>
<thead>
<tr>
<th>Mb/s</th>
<th>runC</th>
<th>kata-containers</th>
<th>runV</th>
<th>clear containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>211</td>
<td>101</td>
<td>65.6</td>
<td>102</td>
<td></td>
</tr>
</tbody>
</table>

Average System Load

Less is better

<table>
<thead>
<tr>
<th>CPU Load</th>
<th>runC</th>
<th>kata-containers</th>
<th>runV</th>
<th>clear containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.62</td>
<td>3.17</td>
<td>3.17</td>
<td>3.91</td>
<td></td>
</tr>
</tbody>
</table>

Runtime performance overhead affect application running inside container.

runC perform best in both IO throughput and average CPU load.

kata-containers perform best among VM containers.
Workloads
Serverless

• Host functions instead of applications?
  • Functions as service
  • e.g. AWS Lambda
• Ideal Platform
  • Low latency
  • High parallelism i.e. high density.
  • Low on resources (CPU, Memory)
## Serverless platform

<table>
<thead>
<tr>
<th></th>
<th>containerd + runC</th>
<th>cri-o + runC</th>
<th>Frakti + runV</th>
<th>Any + kata-containers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Latency</strong></td>
<td>Best</td>
<td>Better</td>
<td>Good</td>
<td>Average</td>
</tr>
<tr>
<td>Cold start</td>
<td>Best</td>
<td>Better</td>
<td>Better</td>
<td>Average</td>
</tr>
<tr>
<td>Warm start</td>
<td>Better</td>
<td>Best</td>
<td>Average</td>
<td>Good</td>
</tr>
<tr>
<td><strong>Density</strong></td>
<td>Best</td>
<td>Good</td>
<td>Average</td>
<td>Average</td>
</tr>
<tr>
<td><strong>Security</strong></td>
<td>Good (namespace + seccomp + SELinux)</td>
<td>Good (namespace + seccomp + SELinux)</td>
<td>Best (VM based)</td>
<td>Best (VM Based)</td>
</tr>
<tr>
<td><strong>Stability</strong></td>
<td>Stable (defined support cycle for each release)</td>
<td>Stable/Best with Openshift (Not defined)</td>
<td>Stable (managed by hyper.sh) (not defined)</td>
<td>Under Active development</td>
</tr>
<tr>
<td><strong>Support Cycle</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Peak hour demand / Micro Services

- Mostly applications are of type Micro services.
- Ideally immutable
- Quick scale up and scale down.
- Ideal Platform
  - Low latency for start application and free resources.
  - Better utilize the host system.
Mean Time To Recover (MTTR) - DevOps

• Short Lived containers
• Frequent updates
• Fast recovery is important.
• Low on resources
## Micro-services / MTTR

<table>
<thead>
<tr>
<th></th>
<th>containerd + runC</th>
<th>cri-o + runC</th>
<th>Frakti + runV</th>
<th>Any + kata-containers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Latency</strong></td>
<td>Best</td>
<td>Better</td>
<td>Good</td>
<td>Average</td>
</tr>
<tr>
<td><strong>Density</strong></td>
<td>Best</td>
<td>Better</td>
<td>Average</td>
<td>Good</td>
</tr>
<tr>
<td><strong>Security</strong></td>
<td>Good (namespace + seccomp + SELinux)</td>
<td>Good (namespace + seccomp + SELinux)</td>
<td>Best (VM based)</td>
<td>Best (VM Based)</td>
</tr>
<tr>
<td><strong>Stability</strong></td>
<td>Stable (defined support cycle for each release)</td>
<td>Stable/Best with Openshift (Not defined)</td>
<td>Stable (managed by hyper.sh) (not defined)</td>
<td>Under Active development</td>
</tr>
<tr>
<td><strong>Support Cycle</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Long running containers

- Migrated application.
- Stateful containers.
- Hard to scale containers.

Requirements
- Stability
- Security
- Performance
- Migration
# Long running containers

<table>
<thead>
<tr>
<th></th>
<th>containerd + runC</th>
<th>cri-o + runC</th>
<th>Frakti + runV</th>
<th>Any + kata-containers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stability</strong></td>
<td>Best</td>
<td>Stable/Best with Openshift</td>
<td>Good</td>
<td>Under Active development</td>
</tr>
<tr>
<td><strong>Support Cycle</strong></td>
<td>(defined support cycle for each release)</td>
<td>(Not defined)</td>
<td>(managed by hyper.sh)</td>
<td>(not defined)</td>
</tr>
<tr>
<td><strong>Security</strong></td>
<td>Good (namespace + seccomp + SELinux)</td>
<td>Good (namespace + seccomp + SELinux)</td>
<td>Best (VM based)</td>
<td>Best (VM Based)</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>Best</td>
<td>Best</td>
<td>Average</td>
<td>Better</td>
</tr>
<tr>
<td><strong>Overhead</strong></td>
<td>Best</td>
<td>Best</td>
<td>Required</td>
<td>Required</td>
</tr>
<tr>
<td><strong>Migration</strong></td>
<td>Required</td>
<td>Required</td>
<td>Required</td>
<td>Required</td>
</tr>
<tr>
<td><strong>Governance</strong></td>
<td>CNCF + OCI</td>
<td>Kubernetes Incubator + OCI</td>
<td>Kubernetes + hypersh</td>
<td>OpenStack Foundation</td>
</tr>
</tbody>
</table>

- Best
- Good
- Under Active development
- Average
- Better
- Required
Summary

• CRI and OCI enable more choices for container runtimes.
• For Cloud Native workloads, Linux containers based runtimes suite better.
• High level runtime performance do not matter much for long running containers, So low level runtime performance & capabilities become focus.
• VM based runtimes are promising, but still need some time to reach flexibility and usability as Linux containers runtime.
• Migration of monolithic applications / high security applications to modern platform like kubernetes will get boost with VM based runtimes.
Few more OCI runtimes

• Runtime getting ready for OCI complaint
  • rkt - container runtime from CoreOS
    • https://github.com/rkt/rkt
    • https://github.com/rkt/rkt/issues/3368
  • gVisor - Sandbox based containerization
    • https://github.com/google/gvisor
  • railcar – linux containers in implementation in rust
    • https://github.com/oracle/railcar
    • slow development
  • crun – linux containers in implementation in C
    • https://github.com/giuseppe/crun
    • Fully featured but lack clarity on maintenance and support.
Thank You