MENU

Docker components explained

27 APRIL 2018 on Docker, Kubernetes, Containerd

It's all started with a pressure of splitting the monolithic implementation of Docker and Moby Project as result. Now Docker consist of several components on particular machine and confusion can happen when people are talking about components of Docker. So let't improve the situation...

Docker CLI (docker)

`/usr/bin/docker`

Docker is used as a reference to the whole set of docker tools and at the beginning it was a monolith. But now docker-cli is only responsible for user friendly communication with docker.

So the command's like `docker build ...` `docker run ...` are handled by Docker CLI and result in the invocation of **dockerd** API.

Dockerd

`/usr/bin/dockerd`

The Docker daemon - **dockerd** listens for Docker API requests and manages host's Container life-cycles by utilizing **containerd**

dockerd can listen for Docker Engine API requests via three different types of Socket: unix, tcp, and fd. By default, a unix domain socket is created at `/var/run/docker.sock`, requiring either root permission, or docker group membership. On Systemd based systems, you can communicate with the daemon via Systemd socket activation, use `dockerd -H fd://`.

There are many configuration options for the daemon, which are worth to check if you work with docker (dockerd).

My impression is that dockerd is here to serve all the features of Docker (or Docker EE) platform, while actual container life-cycle management is "outsourced" to **containerd**.

Containerd

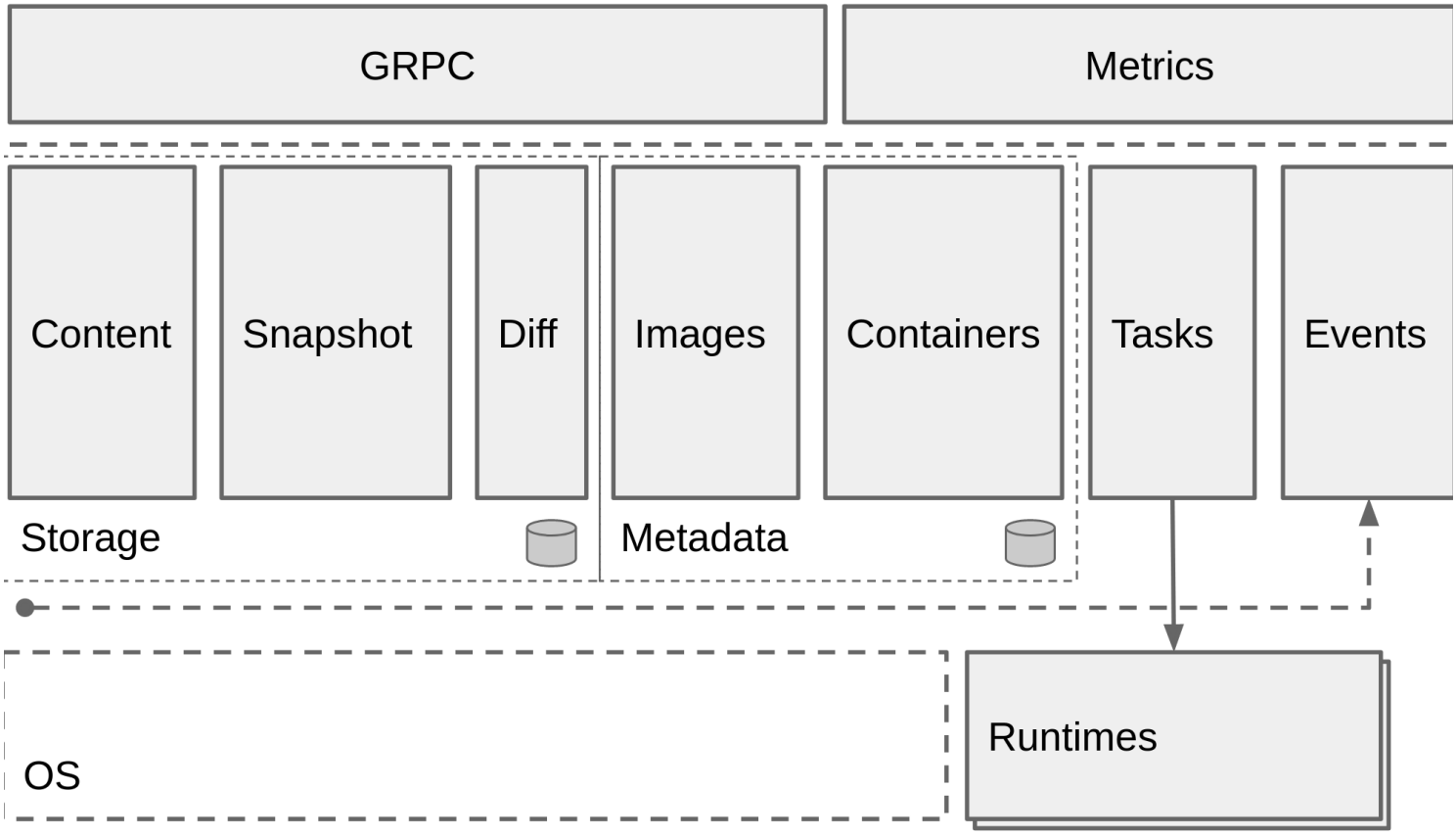
`/usr/bin/docker-containerd`

containerd was introduced in Docker 1.11 and since then took main responsibilty of managing containers life-cycle. containerd is the *executor for containers*, but has a wider scope than *just executing* containers. So it also take care of:

- Image push and pull
- Managing of storage
- Of course executing of Containers by calling **runc** with the right parameters to run containers...

- Managing of network primitives for interfaces
- Management of network namespaces containers to join existing namespaces

containerd fully leverages the *OCI runtime specification*[1], image format specifications and OCI reference implementation (runc). Because of its massive adoption, containerd is the industry standard for implementing OCI. It is currently available for Linux and Windows.



As shown on the picture above, containerd includes a daemon exposing gRPC API over a local UNIX socket. The API is a low-level one designed for higher layers to wrap and extend. Containerd uses **RunC** to run containers according to the OCI specification.

containerd is based on the Docker Engine's core container runtime to benefit from its maturity and existing contributors, however containerd is designed to be embedded into a larger system, rather than being used directly by developers or end-users.

Well now other vendors can use contanerd without have to deal with docker related parts.

let's go through some subsystems of containerd...

RunC

`/usr/bin/docker-runc` runc (OCI runtime) canbe seen as component of containerd.

runc is a command line client for running applications packaged according to
the OCI format and is a compliant implementation of the OCI spec.

Containers are configured using bundles. A bundle for a container is a directory
that includes a specification file named "config.json" and a root filesystem.
The root filesystem contains the contents of the container.

Assuming you have an OCI bundle you can execute the container

```
# run as root
cd /mycontainer
runc run mycontainerid
```

containerd-ctr

`/usr/bin/docker-containerd-ctr` (docker-)containerd-ctr - it's barebone CLI (ctr) designed specifically for development and debugging purpose for direct communication with containerd. It's included in the releases of containerd. By that less interesting for docker users.

containerd-shim

`/usr/bin/docker-containerd-shim`

The shim allows for daemonless containers. According to Michael Crosby it's basically sits as the parent of the container's process to facilitate a few things.

- First it allows the runtimes, i.e. runc,to exit after it starts the container. This way we don't have to have the long running runtime processes for containers.
- Second it keeps the STDIO and other fds open for the container in case containerd and/or docker both die. If the shim was not running then the parent side of the pipes or the TTY master would be closed and the container would exit.
- Finally it allows the container's exit status to be reported back to a higher level tool like docker without having the be the actual parent of the container's process and do a wait4.

How it all works together

We can do an experiment. First we check what *Docker processes* are running right after Docker installation.

```
ps fxa | grep docker -A 3
# prints:
2239 ?        Ssl    0:27 /usr/bin/dockerd -H fd://
 2397 ?        Ssl    0:18  \_ docker-containerd -l unix:///var
...
```

well at this point we see that dockerd is started and containerd is running as a child process too. Like described, dockerd needs containerd ;)
Now let's run one simple container, that executes for a minute and then exits.

```
docker run -d alpine sleep 60
```

Now we should see it in the process list in the next 60 seconds. Let's check again:

```
ps fxa | grep dockerd -A 3
#prints
 2239 ?        Ssl    0:28 /usr/bin/dockerd -H fd://
 2397 ?        Ssl    0:19  \_ docker-containerd -l unix:///var
15476 ?        Sl     0:00      \_ docker-containerd-shim 3da7
15494 ?        Ss     0:00          \_ sleep 60
```

Now we see the whole process chain:

dockerd --> **containerd** --> **containerd-shim** --> "sleep 60" (desired process in the container).

We do not see runc in the chain, we know containerd-shim takes over after runc has started the container. Also we know that

theoretically containerd-shim can survive crash of containerd. But in current docker version it's not activated by default.

However it's pretty long chain with possible disadvantages that such chains might have.

How it all works in Kubernetes

You might imagine that Kubernetes do not need Docker-specific parts. As of now, it's exactly the case...

Kubernetes "speaks" with contanerd directly as depicted in the picture. If interested, check how it was in between.

I hope this might help all Docker users. Give me a hint if something is not precise.

1. *The OCI Runtime Specification outlines how to run a containers "filesystem bundle" that is unpacked on disk. At a high-level an OCI implementation would download an OCI Image (OCI Image Specification) then unpack that image into an OCI Runtime filesystem bundle. At this point the OCI Runtime Bundle would be run by an OCI Runtime.* ↩

Alexander Holbreich **Share this post**

Read more posts by this author.

http://alexander.holbreich.org/alexander/

Gowtham GK • 4 months ago

Lovely Article... Cleared me a lot.

1 • Reply • Share ›

luke • a year ago

Nice article!

1 • Reply • Share ›

Matthew Ceroni luke • 9 months ago

Agree with this statement. I had read other similar articles/blogs but got confused with all the terms.

This was simple and to the point, and the example at the end really drove it home.

1 • Reply • Share ›

AlexH Mod Matthew Ceroni • 9 months ago

thank you

• Reply • Share ›

AlexH Mod luke • a year ago

thank you

• Reply • Share ›

AlexH — After tries with queries and aggregations, i come to the conclusion that it's not a good idea, and result and

7 comments • 5 years ago

grok — I like the standardization which systemd brings to the major distros, especially Debian. I don't mind that

READ THIS NEXT

Services in systemd

Previosly (meanwhile years ago) i've wrote an intro to systemd. Recently i had close contact with it, i want...

YOU MIGHT ENJOY

Elasticsearch: Working with Indices

Continuing article series on Elasticsearch this article explains things around indices. Creating an index Grab you'r favorite REST tool...