Where are containerd's graph drivers?

Michael Crosby Follow
Nov 15, 2017 · 5 min read

The short answer is that containerd does not have graph drivers, but it does have snapshotters. We didn't make this change because we wanted to invent something new or had a severe case of NiH (not invented here). We set out to fix a few long standing issues that are caused by the design of graph drivers. To better understand some of these issues, we first have to look at the history of graph drivers and how they were originally developed.

Types of container filesystems

In the container world we use two types of filesystems: overlays and snapshotting filesystems. AUFS and OverlayFS are overlay filesystems which have multiple directories with file diffs for each "layer" in an image. Snapshotting filesystems include devicemapper, btrfs, and ZFS which handle file diffs at the block level. Overlays usually work on common filesystem types such as EXT4 and XFS whereas snapshotting filesystems only run on volumes formatted for them.

The history of graph drivers

So what are graph drivers? How were they born? Why are they called graph drivers? All very good questions.

To answer these questions we have to go back 3 or some years ago to the Docker 0.8 timeframe. At the time, Docker only supported Ubuntu, as it was the only distro that shipped with AUFS, an overlay filesystem that Docker used to layer images and container's read-write layers. During the 0.8 timeframe, we were working on getting Docker to run on old kernels. In order to do this, we needed Docker to support more filesystems than just AUFS. As such, Device mapper(LVM thinpool) support was contributed as an alternative to AUFS so that Docker could run on older kernels.

At first, device mapper seemed like it would solve all our problems. The goal to have Docker running on all kernels and distros was solved by device mapper indeed. In addition, we were hoping to remove AUFS support and only use device mapper for container filesystems. However, after testing the initial implementation, it was not the solution that we hoped for. There were stability issues, slow builds, and udev issues between distros. We still needed to give users a way to run Docker on various distros but still give existing users on Ubuntu the type of performance they were used to.

In order to provide Docker to a broader user base on a variety of distros, we decided that filesystem support in Docker needs to be pluggable. So, late one night, Solomon and I sat down and started to design a new driver API to support multiple filesystems in Docker. Solomon mocked out and designed the API, while I ported over AUFS to validate that the interface would work.

We called this new API "graph drivers" because Docker modeled the images and the relationship of images to various layers in a graph and the filesystems mostly stored images. Naming is hard…

Now, "back in my day", the initial graph driver interface was very simple and it worked well. However, as years passed, more and more requirements were added. The graph driver API began to grow to support things like:

- Build Optimizations such as caching and layer sharing for faster build times
- Content Addressability to ensure the filesystems were securely identified
- Changing runtime requirements from LXC to runc

The result of these changing requirements was the graph driver API and its underlying implementations started to be intertwined with the high level features that they were supporting. This resulted in a few issues:

- The graph driver API became unnecessarily complex

- Drivers had built optimizations coded into every driver
- Drivers were tightly coupled to container lifecycle
- Maintenance was hard with the scope of graph drivers

I don't think we could have foreseen these design issues in the beginning, since we didn't have the requirements as we do today. Live and learn I guess. ¯_(ツ)_/¯

Snapshotters to the rescue

Fast forward 3 years in the future and it was time to rethink how graph drivers worked with containers. As part of the containerd work, we had the time to hit the breaks and fix some of these long standing issues.

API complexity

To solve the issue of the complex API that the graph drivers have grown into, we needed to look at what was required for both overlay and snapshotting filesystems to function. We knew that snapshotters were less flexible than overlay filesystems because of the strict parent-child relationship that snapshots have. Because they work at the block level, you HAVE to have a parent snapshot before creating a child snapshot.

As a general rule when writing system software, we try to find the implementations that are the least flexible in terms of I/O to create an interface around. This is the reason why containerd calls the filesystem component "snapshotters" as we modeled the interface after them. After we developed the initial API for snapshotters, it was almost effortless to make overlay filesystems function like one.

You can see the full interface along with relevant information on each method here https://godoc.org/github.com/containerd/containerd/snapshot#Snapshotter

Container lifecycle changes for mounts

With graph drivers, it was always the driver's responsibility to mount and unmount a root filesystem for a container. This came from when we were still using LXC and had to execute a container within a fully mounted rootfs. As we transitioned over to runc, this was no longer a requirement.

I wanted all mounts to happen inside a mount namespace and not on the host. I also didn't want the snapshotters to mount anything, and if they don't mount, they don't unmount as well.

There are a few benefits to this. One is that the caller, being a builder or execution component, can decide when it needs to mount the rootfs and when execution has ended, so that, it can unmount. In addition to mounting everything in a separate mount namespace for a container, when the container dies, the kernel will unmount everything in that namespace. This eliminates the issues with stale file handles that plague some graph drivers.

We achieved this by designing the snapshot API to return a serialized mount call that can be mounted and unmounted by the caller to the location of their choosing. This is used by the execution component in containerd to mount a container's root filesystem in the *containerd-shim* and unmounted at the end of the task execution.

Maintenance

Lastly, we wanted to make sure snapshotters were something that we can support in the long run. This was achieved by having a simpler interface that allowed us to remove most of the caller specific requirements, so that snapshotters could focus on one thing, being a snapshotter. You can see this by looking at the LOC of some of the snapshotter. Now, I know, LOC is not a good judge of complexity for software but when I'm working on a bug, it's easier to find a bug in a package with 100 lines of code vs 1000.

At the time of writing this post the overlay snapshotter is currently 403 lines of code including comments and btrfs is 372 lines of code with comments.

In the end, snapshotters are an evolution of graph drivers. We set out to fix the long standing issues with graph drivers that users were facing and fix them in a way that we can support for years to come.

Learn more about containerd:

- Get started with containerd
- Get started with CRI-containerd
- Check out the containerd repo
- Join the #containerd channel on slack