Storage Options for Software Containers

MARK LAMOURINE

Mark Lamourine is a senior software developer at Red Hat. He's worked for the last few years on the OpenShift project. He's a coder by training, a sysadmin and toolsmith by trade, and an advocate for the use of Raspberry Pi style computers to teach computing and system administration in schools. Mark has been a frequent contributor to the *;login:* book reviews. markllama@gmail.com

Software containers are likely to become a very important tool over the next few years. While there is room to argue whether or not they are a new tool, it is clear that they have certain elements that are clearly immature. Storage is one of those elements.

The problem isn't that we need new storage services. The new factors are due to the characteristics of containers themselves and how they differ from traditional bare-metal hosts and virtual machines (VMs). Also, storage isn't an issue on single hosts where it can be mounted manually for each individual container. Large container farms present problems that are related to those for VM-based IaaS services but that are complicated by VMs' lack of clean boundaries.

There are two common container mechanisms in use today: LXC and Docker. LXC is the older mechanism and requires careful crafting of the container environment, although it also provides more control to the user. Creating LXC containers requires a significant level of expertise. LXC also does not provide a simple means to copy and re-instantiate existing containers.

Docker is the more recent container environment for Linux. It makes a set of simplifying assumptions and provides tools and techniques that make creating, publishing, and consuming containers much easier than it has ever been. This has made container technology much more appealing than it was before, but current container systems only manage individual containers on single hosts. As people begin trying to put containers and containerized applications to use at larger scales, the remaining problems, such as how to manage storage for containers, are exposed.

In this article I'm going to use Docker as the model container system, but all of the observations apply as well to LXC and to container systems in general.

A Container Primer

The first thing to understand is that containers don't contain [1]. A container is really a special view of the host operating system that is imposed on one or more processes. The "container" is really the definition of the view that the processes will see. In some senses they are similar to *chroot* environments or *BSD jails* but the resemblance is superficial and the mechanism is entirely different.

The enabling mechanism for Linux containers is *kernel namespaces* [2, 3]. Kernel namespaces allow the kernel to offer each process a different view of the host operating system. For example, if a contained process asks for `stat(3)` for the root (`/`), the namespace will map that to a different path (when seen by an uncontained process): for example, `/var/lib/docker/devicemapper/mnt/<ID>/rootfs/`. Since the file system is hierarchical, requests for information about files beneath the root node will return answers from inside the mapped file tree.

In all *NIX operating systems, PID 1 is special. It's the init process that is the parent of all other processes on a host. In a Docker container, there is a master process, but it is generally not the system process. Rather, it may be a shell or a Web server. But from inside the container, the master process will appear to have PID 1.

There are six namespaces that allow the mapping of different process resources [6]:

- mount—file systems
- UTS—nodename and domain name
- IPC—inter-process communication
- PID—process identification
- network—network isolation
- user—UID mapping

A process running "in a container" is, in fact, running directly on the container host. All of the files it sees are actually files inside the host file system. The "containment" of the process is an illusion, but a useful one. This lack of the traditional boundaries is what makes container storage management something new.

Software Container Farms and Orchestration

If all you want to do is run a single container on a single host with some kind of storage imported, there's no real problem. You manually create or mount the storage you want, then import the storage when you create the container. Both LXC and Docker have means of indicating that some external file-system root should be re-mapped to a new mount point inside the container. When you want to create a container farm, where placement of individual containers is up to the orchestration system, then storage location becomes interesting. In a container farm, the person who requests a new container only gets to specify the characteristics, not the deployment location.

There are a number of container orchestration systems currently under development. CoreOS is using a system called Fleet [3]. Google and Red Hat are working on Kubernetes [4]. Both have slightly different focus and features but in the end they will both have to create the environment necessary to run containers on hosts that are members of a cluster. I think it's too early to tell what will happen in the area of container orchestration system development even over the short term.

I'm not going to talk about how the orchestration systems will do their work, I'm only going to talk about the flavors of storage they will be called on to manage and the characteristics and implications of each. But first, let's look at how Docker handles storage without an orchestration system.

Docker Ephemeral Storage

When you ask Docker to create a container, it unpacks a collection of tar archives that together are known as the *image*. If no external volumes are specified, then the only file tree mounted is the unpacked image. Each of the running containers is unpacked into `/var/lib/docker/devicemapper/mnt/<ID>/rootfs` where `<ID>` is the container ID returned when the container is created using the devicemapper driver. Different storage drivers will have slightly different paths.

This is *ephemeral storage* in the sense that when the container is deleted, the storage is reclaimed and the contents deleted (or unmounted). This file tree is not shared with other containers. Changes made by processes in the container are not visible to other containers.

Docker Volumes—Shared and Persistent Storage

The Dockerfile VOLUME directive is used to define a mount point in a container. It basically declares, "I may want to mount something here or share something from here." Docker can mount different things at that point in response to arguments when a container is created.

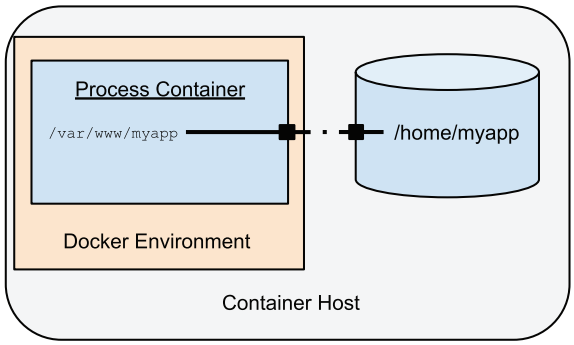

Figure 1: The Docker VOLUME directive creates a mount point in the container. The file tree below the mount point in the image is placed in a separate space when the container is created. It can be exported to other containers or imported either from a container or from external storage.

When you create a new Docker container from an image that declares a volume, but you don't provide an external mount point, then the content below the volume mount point is placed in its own directory within the Docker workspace (`/var/lib/docker/vfs/dir/`).

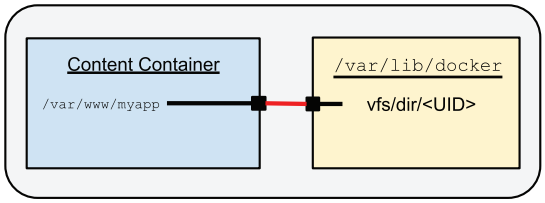

Figure 2: A container connected to an "internal" volume. This is created by Docker as a default if no external volume is offered.

You can share this storage between containers on the same host with the docker `--volumes-from` command line option. This causes the declared volumes from an existing container to be mounted on any matching volumes in the new container.

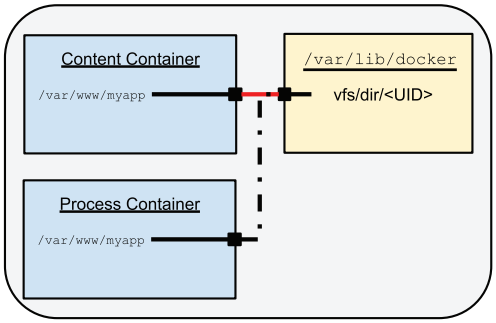

Figure 3: Two containers sharing a volume. The volume is created by Docker when the first container is created. The second container mounts from the first using the `--volumes-from` option.

Shared storage using what's known as a *data container* can be treated as persistent across restarts of an application. The application container can be stopped and removed and then recreated or even upgraded. The data container will be available to remount when the application is restarted.

This volume sharing will also work with *host storage*.

External Host Storage

In this case "external" means "from outside the container." When you start a Docker container, you can indicate that you want to mount a file or directory from the host to a point inside the container.

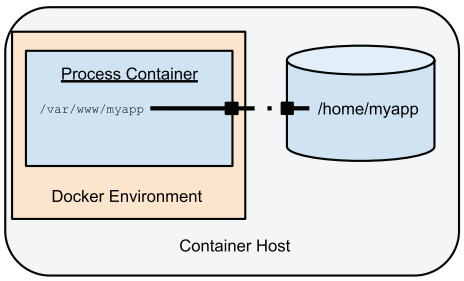

Figure 4: A container with host storage. The host storage is bind mounted onto the container volume mount point.

Host storage is useful when you are creating individual containers on a single host. You can create and manage the space on the host that you're going to mount inside before you start the container. This gives you more control over the initialization of the storage, and you can view and modify it easily from the host while the application is running.

This becomes more difficult, though, when you start working with a large container farm. The whole idea of container farms is that all of the hosts are identical and that the containers can be placed and moved to maintain the balance of the load within the cluster. The only way to do that practically is to move the storage off the container host entirely.

Containers, the `mount(8)` Command and Container Orchestration

I've mentioned the lack of the host boundary when managing containers. The `mount(8)` command is where this appears for storage. You can't run `mount(8)` from inside a container without special privileges. Since a container is just a special view of the host, any file system mounted into a container must be mounted onto the host before the container is started. In general, processes inside containers are not given the privileges to affect anything on the host outside the container. (If they can, they're not very well contained, are they?)

For similar reasons, Docker cannot cause the host to mount new file systems, whether local or remote. Docker restricts itself to controlling how the container sees the resources that the host already has available. Docker manages individual containers on single hosts. For large numbers of containers spread across multiple hosts, the orchestration system will have to provide a way to connect storage to containers within the cluster. In the rest of this article, I'll talk about where the storage will come from.

Docker and the Host Boundary

At this point you've seen everything that Docker can do with storage. Docker limits itself to using the resources available on a host. Its job is to provide those resources to the interior of containers while maintaining the limited view the containers have of the host outside. This means that Docker itself is unaware of any containers on other hosts or of any other resource that has not been attached to the host when the container is created. Docker can't make changes to the host environment on behalf of a container.

This is where orchestration systems come in. A good orchestration system will have a way to describe a set of containers and their interactions to form a complete application. It will have the knowledge and means to modify the host environment for new containers as they are created.

Network Storage

Most machines have block storage mounted directly on the host. Network storage extends the reach of individual hosts to a larger space than is possible with direct storage, and it allows for the possibility of sharing storage between multiple hosts.

I'm going to group all of the traditional off-host storage services under the umbrella of "network storage." These include NAS, SAN, and more modern network storage services. There are a few differences.

NAS services like NFS, CIFS, and AFS don't appear on the host as devices. They operate using IP protocols on the same networks that carry the other host traffic. Their unit of storage is a file (or directory). They generally don't tolerate latency very well. In their original form, NAS services don't support distributed files or replication. In most cases they don't require a service running on the client host to manage the creation of connections or the file traffic. NAS services can be provided by specialized appliances or by ordinary hosts running the service software.

There is a new class of distributed network services that provide replication and higher performance than single-point NAS does. Gluster and Ceph are two leading distributed NAS services. Clients run special daemons that distribute and replicate copies of the files across the entire cluster. The files can either be accessed on the client hosts or be served out over NFS to other clients.

SAN systems include Fibre Channel, InfiniBand, and iSCSI. Fibre Channel and InfiniBand require special hardware networks and connections to the host. iSCSI can run over IP networks and so does not require special hardware and networks, although for the best performance, users often need to create distinct IP networks to avoid conflicts with other data traffic. SAN services usually require some additional software to map the service end points to *NIX device files, which can be partitioned, formatted, and mounted like ordinary attached storage. SAN services provide very low latency and high throughput, to the point where they can substitute for attached storage.

For container systems these all pose essentially the same problem. The orchestration system must locate and mount the file system on the host. Then it must be able to import the storage into the container when it is created.

File Ownership and UID Mapping

One major unsolved problem for Docker (at the time of this writing) is control of the ownership and access permissions on files.

*NIX file ownership is defined by UID and GID numbers. For a process to access a file, the UID of a process must match the file owner UID, and the user must have the correct permissions or get access via group membership and permissions. With the exception of the root user (UID 0, GID 0), the assignment of UID/GID is arbitrary and by convention.

The UID assignment inside a container is defined by the `/etc/passwd` file built into the container image. There's no relation to the UID assignment on the container host or on any network storage.

When a process inside a container creates a file, it will be owned by the UID of the process in the container even when seen from the host. When using host, network, or cloud block storage, any process on the host with the same UID will have the same access as the processes inside the container.

Access in the other direction is also a problem. If files on shared storage are created with a UID that does not match the process UID inside the container, then the container process will fail when accessing the storage.

This will also benefit developers trying to create generic container images that are able to share storage between containers. Currently, any two containers that mean to share storage must have identical user maps.

The Linux kernel namespace system includes the ability to map users from inside a container to a different one on the host. The Docker developers are working on including user namespaces, but they present a number of security issues that have to be resolved in the process.

Container Hosts and Storage Drivers

Even before the introduction of Docker there was a movement to redefine the way in which software is packaged and delivered. CoreOS [5] and, more recently, Project Atomic [6] are projects which aim to create a stripped down host image that contains just the components needed to run container applications. Since they just run containers, much of the general purpose software normally installed on a host or VM isn't needed. These lean images do not need patching. Rather, the host image is replaced and rebooted as a unit (hence, "Atomic").

Although this simplifies the maintenance of both the host and the containers, using "forklift updates," the rigid image formats make adding drivers or other customizations difficult. There is a very strong pressure to keep the host images small and to include only critical software. Everything that can be put into a container is, even tools used to monitor and manage the container host itself.

These purpose-made container hosts will need to provide a full range of network storage drivers built into the image, or they will have to be able to accept drivers running in containers if they want to compete with general purpose hosts configured for containers. It's not clear yet which drivers will be available for these systems, but they are being actively developed.

Cloud Storage

Cloud services take a step further back. They disassociate the different components of a computer system and make them self-service. They can be managed through a user interface or through an API.

Cloud storage for applications usually takes one of two forms: *object storage* and *block storage*. (The third form of cloud storage, *image storage,* is used to provide bootable devices for VMs.)

Object Storage

All of the cloud services, public and private, offer some form of object storage, called *Swift* in OpenStack. The AWS object store is *S3,* and Google offers *Google Cloud Storage* (not to be confused with Google Cloud Engine Storage; see "Block Storage," below).

Object stores are different from the other storage systems. Rather than mounting a file system, the data are retrieved through a REST API directly by processes inside the container. Each file is retrieved as a unit and is placed by the calling application into an accessible file space. This means that object storage doesn't need any special treatment by either the container system or the orchestration system.

Object stores do require some form of identification and authentication to set and retrieve data objects. Managing sensitive information in container systems is another area of current work.

Container images that want to use object stores must include any required access software. This may be an API library for a scripting language or, possibly, direct coding of the HTTP calls.

The push-pull nature of object stores makes them unsuitable for uses that require fast read/write access to small fragments of a file. Access can have very high latency, but the objects are accessed as a unit, so they are unlikely to be corrupted during the read/write operations. The most common uses are for configuration files and for situations where data inconsistency from access collisions can be accepted in the short term.

Block Storage

Cloud block storage appears on a (virtual) host as if it were direct attached storage. Each cloud storage system has a different name for its own form of block storage. OpenStack uses the *Cinder* service. On Amazon Web Services it's known as *EBS*. Google calls it *GCE Storage* (not to be confused with Google Cloud Storage).

Cloud block storage systems are controlled through a published API. When a process requests access to cloud storage, a new device file is created. Then the host can mount the device into the file system. From there Docker can import the storage into containers.

The challenge for an orchestration system is to mount each block device onto a container host on-demand and make it available to the container system. Since each cloud environment has a different API, either they all must be hard-coded into the orchestration system or the orchestration system must provide a plugin mechanism.

So far the only combination I've seen work is Kubernetes in Google Cloud Engine, but developers on Kubernetes and others all recognize the need for this feature and are actively developing.

Summary

Container systems in general and Docker in particular are limited in scope to the host on which they run. They create containers by altering the view of the host that contained processes can see. They can only manage the resources that already exist on the host.

Orchestration systems manage multiple containers across a cluster of container hosts. They allow users to define complex applications composed of multiple containers. They must create or configure the resources that the containers need, and then trigger the container system, like Docker, to create the containers and bind them to the resources.

Currently, only Kubernetes can mount GCE Storage when running in the GCE environment.

For container systems to scale, the orchestration systems will need to be extended to be able to communicate and manage the various network and cloud storage systems. Docker and the orchestration systems will need to be able to manage user mapping as well as file access controls.

In both Fleet and Kubernetes, the development teams are actively working to address all of these issues, and I expect that there will be ways to manage storage in large container farms very soon. Once there are, containers will begin to fulfill their promise.

For a more detailed treatment of containers, see the article by Bottomley and Emelyanov [7].

References

[1] Daniel Walsh, "Are Docker Containers Really Secure?": http://opensource.com/business/14/7/docker-security-selinux.

[2] Linux kernel namespaces: http://lwn.net/Articles/531114/.

[3] CoreOS Fleet: https://github.com/coreos/fleet.

[4] Google Kubernetes: https://github.com/GoogleCloudPlatform/kubernetes.

[5] CoreOS container hosts: https://coreos.com/.

[6] Project Atomic: http://www.projectatomic.io/.

[7] James Bottomley and Pavel Emelyanov, "Containers," *;login:*, vol. 39, no. 5, Oct. 2014: https://www.usenix.org/publications/login/october-2014-vol-39-no-5/containers.