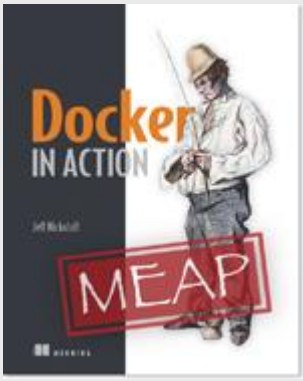

Docker in Action: Shared Memory

By Jeff Nickoloff

In this article, excerpted from the book Docker in Action, I will show you how to open access to shared memory between containers.

Linux provides a few tools for sharing memory between processes running on the same computer. This form of inter-process communication (IPC) performs at memory speeds. It is often used when the latency associated with network or pipe based IPC drags software performance below requirements. The best examples of shared memory based IPC usage is in scientific computing and some popular database technologies like PostgreSQL.

Docker creates a unique IPC namespace for each container by default. The Linux IPC namespace partitions shared memory primitives like named shared memory blocks and semaphores, as well as message queues. It is okay if you are not sure what these are. Just know that they are tools used by Linux programs to coordinate processing. The IPC namespace prevents processes in one container from accessing the memory on the host or in other containers.

Sharing IPC Primitives Between Containers

I've created an image named `allingeek/ch6_ipc` that contains both a producer and consumer. They communicate using shared memory. Listing 1 will help you understand the problem with running these in separate containers.

Listing 1: Launch a Communicating Pair of Programs

```
# start a producer
docker -d -u nobody --name ch6_ipc_producer \
allingeek/ch6_ipc -producer


# start the consumer
docker -d -u nobody --name ch6_ipc_consumer \
allingeek/ch6_ipc -consumer
```

Listing 1 starts two containers. The first creates a message queue and starts broadcasting messages on it. The second should pull from the message queue and write the messages to the logs. You can see what each is doing by using the following commands to inspect the logs of each:

```
docker logs ch6_ipc_producer
 docker logs
ch6_ipc_consumer
```

If you executed the commands in Listing 1 something should be wrong. The consumer never sees any messages on the queue. Each process used the same key to identify the shared memory resource but they referred to different memory. The reason is that each container has its own shared memory namespace.

If you need to run programs that communicate with shared memory in different containers, then you will need to join their IPC namespaces with the `--ipc` flag. The `--ipc` flag has a container mode that will create a new container in the same IPC namespace as another target container.

Listing 2: Joining Shared Memory Namespaces

```
# remove the original consumer docker
rm -v ch6_ipc_consumer

# start a new consumer with a joined IPC namespace
docker -d --name ch6_ipc_consumer \
        --ipc container:ch6_ipc_producer \
allingeek/ch6_ipc -consumer
```

Listing 2 rebuilds the consumer container and reuses the IPC namespace of the `ch6_ipc_producer` container. This time the consumer should be able to access the same memory location where the server is writing. You can see this working by using the following commands to inspect the logs of each:

```
docker logs ch6_ipc_producer
 docker logs
ch6_ipc_consumer
```

Remember to cleanup your running containers before moving on:

```
 # remember: the v option will clean up volumes,
 #           the f option will kill the container if it is running,
 #           and the rm command takes a list of containers
 docker rm -vf ch6_ipc_producer ch6_ipc_consumer
```

There are obvious security implications to reusing the shared memory namespaces of containers. But this option is available if you need it. Sharing memory between containers is safer alternative to sharing memory with the host.

For source code, sample chapters, the Online Author Forum, and other resources, go to
http://www.manning.com/nickoloff/