Improvements to the Docker Container Engine

B.Tech. Project Stage I

Pratik Fegade

advised by

Prof. Umesh Bellur and
Prof. Purushottam Kulkarni

Abstract
Operating system virtualization or containers as they are called, is increasingly becoming wide spread these days in situations where strict isolation is not a requirement. In a setting where a provider provides customers services on the cloud, it becomes necessary for the provider to be able to differentiate between the users. The provider often does this by controlling the amount of various resources allocated to the various customers. Containers, which are based on kernel features like namespaces and cgroups are very well posed to provide a rich API for the service provider to be able to control resource allocations to individual service instances. Docker, a popular container engine does not export this API in its entirety. In this work, we aim to expose to Docker users this API.

1 Introduction

Over the past couple of decades, virtualization has revolutionized the data centres for various service providers, by decoupling the services with the hardware thus enabling better server consolidation, resource usage and control and better QoS due to dependent techniques like live migration and reliability[6]. Traditional hypervisor based virtualization provisions in general heterogeneous OS instances on the same physical machine. Being independent OS instances there is a significant amount of performance and data isolation between two VMs. However, often one might not be in need of such high degree of isolation. In that case one can do with significantly lesser overheads by using containers. Apart from lower resource overheads, containers also provide benefits like low start up and tear down times and leaner image files.

Docker[4] is a popular container engine based initially on LXC[7] and now on libcontainer. Docker has many related tools like Docker swarm, Docker compose, etc. that make the provisioning of an application on containers across a network easy and seamless. Also, the Docker Registry provides an open source Docker image storage and distribution service.

Both LXC and libcontainer use Linux kernel features to create and manage containers. The kernel feature that provides resource isolation and control for processes and thus for containers is cgroups[3]. Docker however does expose to the user the entire of the rich control cgroups provide over resources like CPU, memory, block IO and network IO. Also, there are certain very desirable resource control knobs one would want Docker to provide, which cgroups themselves do not provide. These are the basic problems we try to address in the project. It is predicted how Resource as a Service will be a common paradigm to provide on the cloud in the future[10]. A fine grained control over the resources allocated to containers would be indispensable then.

2 Docker: An Introduction

Docker is basically a wrapper over the libcontainer (or LXC) methods to provide facilities like starting, restarting, pausing and unpausing containers. It is essential to make use of a number of kernel features to create containers. Some of these include

1. cgroups
2. namespaces
3. SELinux or AppArmor
4. chroot

Docker provides a convenient way to exploit these features to create and manage containers and run applitcations in them. Below, we describe Docker in some detail.

2.1 Docker: Usage

Docker provides a CLI interface to create and manage containers and images. A general Docker command thus is of the form

```
docker <subcommand> <options> <arguments>
```

We explain the relevant details in the further sections[5].

2.2 Containers and Images

Docker containers are the analog of running guest virtual machines. An image is required to start a container. An image is thus the analog of a virtual machine snapshot. Both containers and images however are significantly lightweight as compared to their virtual machine counterparts. One can thus spin up a container named `ubuntu_container` from an image called `ubuntu:14.04` using Docker as follows.

```
docker run --name='ubuntu_container' ubuntu:14.04
```

Docker images can be built in two of the following ways

1. **Commiting containers**
   One may start a container from an image, install applications in it and (or) make necessary changes and commit these changes to an image. These changes are saved and will be reflected in a container started from the newly made image.

2. **Dockerfiles**
   This is an automated way to build an image from another. A small Dockerfile to set up apache in Ubuntu 12.04 is shown in Fig. 1[1]. When the `build` subcommand is run with this Dockerfile as the argument, Docker will start from the `ubuntu:12.04` image, start a container from it, run the stated commands in the container and commit the container to form the required image. Specifically, the apache2 package will installed, some environment variables set, port 80 exposed and the start up command of the container when it is started is set.

```
# A basic apache server. To use either add or bind mount content under /var/www
FROM ubuntu:12.04

MAINTAINER John Doe version: 0.1

# Install the necessary packages
RUN apt-get update && apt-get install -y apache2 && apt-get clean && \
    rm -rf /var/lib/apt/lists/*

# Set some environment variables
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2

# Expose the HTTP port
EXPOSE 80

# Set command to run on start of container
CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]
```

Figure 1: A sample Dockerfile

2.3 Volumes

Docker containers are supposed to be as stateless as possible. The container is then just an instance of an application without any associated which makes the management easy and efficient. Practically, applications however need some state that persists across instances. Docker volumes are the entities that allow a container to store this state. Volumes exist inside containers and allow the container to store data outside the container in the filesystem of the host. Volumes can also be used to share data across containers. One can add a volume to a container by the `-v` option of the `run` subcommand. There are others ways to create volumes too, which can be found in the documentation for Docker volumes.

2.4 Docker: Internals

The basic control flow in Docker is described below.

1. Users enters a command to say, start a container
2. The Docker client communicates with the server daemon using a REST API over HTTP
3. The daemon uses libcontainer or LXC API methods to set up a container and the required process(es) in it.
4. LXC or libcontainer use kernel functionalities like cgroups and namespaces to create, manage and stop the containers.

This is illustrated in Fig. 2

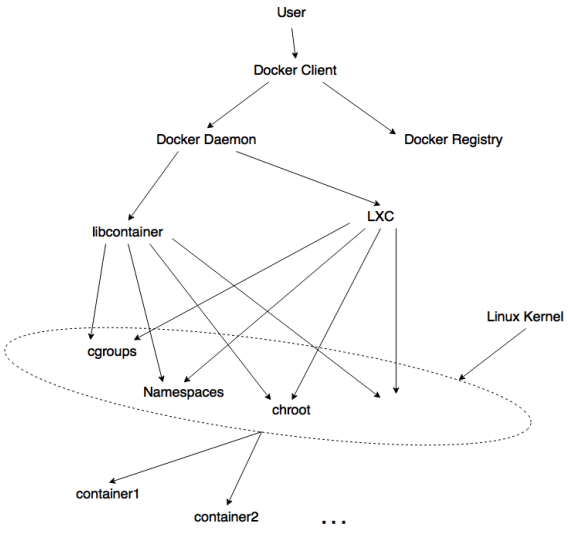

Figure 2: Docker control flow

The source code of Docker is written in Golang. A moderately detailed description of the source code can be found in the appendix of the report.

3 Motivation

As mentioned above, amongst other things, Docker exports the services provided by the Linux kernel for a convenient management of the containers. Cgroups provide a rich set of resource allocation parameters to control the resources allocated to processes. Docker does not export all of these parameters. A description of some these and why they might be useful to Docker users is given below.

One can always manually set the cgroup parameters bypassing Docker. Also, if one is using LXC as Docker's execution driver, we can achieve the same using a command similar to (The `net_cls` subsystem is used here)

```
docker run --lxc-conf="lxc.cgroup.net_cls.classid = 0x00100001" <image> <command>
```

However, this is tedious and requires the user to have low level knowledge about the container when throttling block IO of a container using volumes

1. Block IO throttling[2]
   Cgroups provide facilities to throttle block IO in various ways (A list of the different that can be set can found in the documentation[2]). These might be useful when one wants to restrict the IO of a container that does secondary activities like logging or similar, or even to guarantee some bandwidth to a container.

2. Prioritizing container network usage[9]
   Cgroups provide the `net_prio` subsystem to assign priorities to the network usage of different processes. This would be useful in the case of container in order to discriminate amongst the services provided to different customers based on their subscription type.

3. Classifying container network usage[8]
   Cgroups provide the `net_cls` subsystem to tag network packets from different processes. These tags can then be used for further processing with tools like tc, say for throttling the network or for logging purposes.

Cgroups do not however provide a way to throttle the network usage of processes to a given rate (say in the units of bytes or packets per second). We think this will also be a useful feature for Docker users.

We thus aim to add the above options to Docker in the current work.

4 Work Done

Docker 1.9.0 has been used as a base for the following work. The code for the work can be found here. All the work has been done on a system running Ubuntu 14.04 with Linux kernel 3.13. LXC does not yet support the work done below. We plan to add support in Docker for these options with LXC in the future. Also, there exists some code in Docker apparently for Windows compatibility which we have not yet touched.

4.1 Exporting more control knobs

We have exported knobs to throttle block IO of containers. This has been done by adding options similar to the cpu-shares option above to the run subcommand. Thus in order to start a container with a throttled block IO read rate, the following command could be used

```
docker run --blkio-reed-limit=400 db_server
```

4.2 Dynamic Resource limits

Cgroups allow changing the resource allocations to processes when the processes are active and running. Docker, however does not have options to do the same for containers. We have not been able to understand why this might be so. We have added a new subcommand to Docker called `modresources` that a user can use to change the allocations using the same options as he would use when starting a container with the `run` subcommand. Thus if `db\_server\_instance` was an active container and one wanted to change is cpu shares when it was running, one would the following command.

```
docker modresources -container=db_server_instance -cpu-shares=50
```

4.3 Testing and Evaluation

Currently, we have exported cgroup functionalities to the Docker end user. In order to test this implementation, we start a container with the appropriate resource limits and check if the limits get reflected in the cgroupfs (found at `/sys/fs/cgroupfs/`) on the system. When the `modresources` subcommand is used to change resource allocations, we again check if these changes have been correctly reflected in the cgroupfs.

4.4 Problems faced

We list down the problems faced during the course of the work below.

1. DNS servers are incorrect in containers
   Docker was run on an Ubuntu 14.04 VM using Virtualbox. The default DNS in the build container was being set to Google's DNS servers (one of them being 8.8.8.8) and not to the ones of the network we were working on. This is an issue as Docker uses containers to build itself and it needs to install dependences from its repositories.
   Solution: Manually specify DNS servers to be used by Docker by editing the `/etc/default/docker` file. One needs to add a setting for Docker as follows

   ```
   DOCKER_OPTS="--dns <local nameserver>"
   ```

2. Server client version race conditions
   Docker is run on OSX using boot2docker. Updating the source code/merging it upstream with the Docker repository will upgrade it. The server however still is a lower incompatible version. Upgrading the boot2docker binary should help, but did not. Thus, developing Docker on OSX or windows which use boot2docker is inconvenient.

3. Cgroup options not set when running Docker daemon in the debug mode
   The Docker daemon uses the service infrastructure in Linux. The daemon logs then go in the folder `/var/log/`. We could get Docker to debug log in this way. Also, it was inconvenient to access the logs for debugging purposes this way. we Thus started the daemon using the command

   ```
   docker daemon -D
   ```

   This prints the logs on the console. However it was observed that the resource limits passed on to Docker through its CLI interface were not getting reflected in the cgroupfs when the daemon was started in this manner.

4. Block IO throttling across partitions.
   Cgroups allow throttling of block IO for a process for a particular logical device. the volumes attached to a container may lie on different such devices and the end user is not expected to know this. As a result, he would just specify a limit on the block IO of the container. As applying a single collectively on a set of devices is not possible using cgroups, currently, we apply the limit separately on all the devices from which the container has attached volumes. This is justifiable because we assume that a container would generally run a single or few processes, and often a process would access and process data on the same logical device on the host.
   Note The attachment of a new volume residing on a new device would not currently apply the limit on this new device. The user would have to re run the container or use the modresources subcommand described below.

5 Future Work

Following is a list of further directions for the work.

1. Exporting cgroups options to control network IO of containers as described above.
2. Adding a feature to throttle network IO of containers to a specific hard limit.
3. Cgroups provide a large number of knobs to control resource allocations to processes. If these need to be exported to Docker, we plan to add a facility for the user to be able to specify these in a convenient config file rather on the command line which is tedious and inconvenient.

References

[1] `https://github.com/kstaken/dockerfile-examples/blob/master/apache/Dockerfile`.

[2] Block io controller. `https://www.kernel.org/doc/Documentation/cgroups/blkio-controller.txt`.

[3] Cgroups. `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`.

[4] Docker. `https://www.docker.com`.

[5] Docker-beginner's tutorial. https://blog.talpor.com/2015/01/docker-beginners-tutorial/.

[6] Linux* containers streamline virtualization and complement hypervisor-based virtual machines.

[7] Lxc. https://linuxcontainers.org.

[8] Network classifier cgroup. https://www.kernel.org/doc/Documentation/cgroups/net_cls.txt.

[9] Network priority cgroup. https://www.kernel.org/doc/Documentation/cgroups/net_prio.txt.

[10] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. The resource-as-a-service (raas) cloud. In *Presented as part of the*, Berkeley, CA. USENIX.

Appendix

Below is a brief description of each of the source code folders of Docker explored during the work.

```
./
+-- api/
| +-- client/
| |     Code to files process the subcommands and communicate with the server
| +-- server/
|       Code to communicate with the client and pass on the requests
         to appropriate places (daemon, builder...)
+-- builder/
|    Code for parsing Dockerfiles and building images from them
+-- bundles/
|    This folder, if present contains the binaries
+-- daemon/
| |  Most of the Docker functionalities requiring use of the execution drivers are
      coded up here. Some methods here are called from code in ./api/server/
| +-- execdriver/
| | +-- execdrivers/
| | |      Interface code for the execution drivers (LXC or libcontainer)
| | +-- lxc/
| | |      Communicate with the LXC API here
| | +-- native/
| | |      Communicate with the libcontainer API
| | +-- windows/
| +-- graphdriver/
| |     Code for storage backend of Docker
+-- docker/
|    Docker main files. The server and the client start their execution here
+-- runconfig/
|    Some important structs like hostconfig and config are defined here
+-- vendor/
  +-- src/
      +-- github.com/
        +-- Sirupsen/
        | +-- logrus/
        |         The logger
        +-- opencontainers/
          +-- runc/
              +-- libcontainer/
                      libcontainer
```