docker

Continuous Integration and Testing with Docker

Daniele Andreotti - 20/02/2015

Overview

A good approximation: **Docker is a chroot environment on steroids**.

Docker runs Unix processes, but it gives a better isolation and control over resources utilization.

Docker is based on a client-server architecture

Both the Docker client and the daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon.

Expose a well defined RESTful API.

Images and Containers

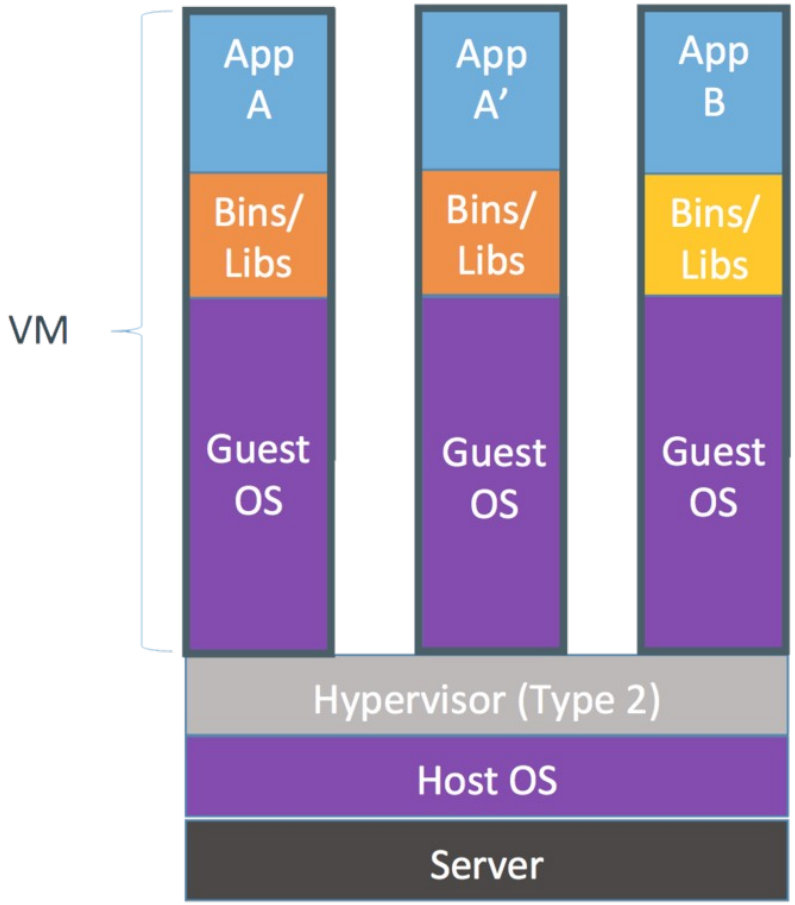
A Docker image is a read-only template.

For example, an image could contain an Ubuntu operating system with Apache and a web application installed.

A container is a running instance of an image.

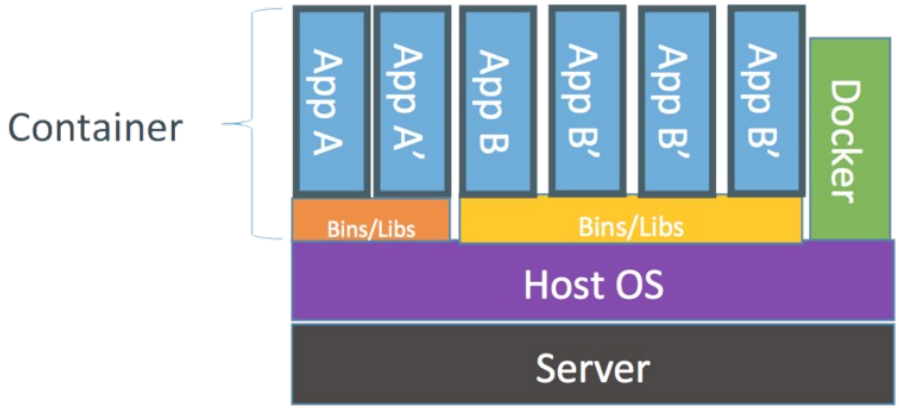
Each container is an isolated application platform.

Containers vs. VMs



Containers are isolated, but share OS and, where appropriate, bins/libraries

...result is significantly faster deployment, much less overhead, easier migration, faster restart



Containers

Kernel shared among the Docker container (the guest) and the “hypervisor” OS

Docker can run on other platforms through a further virtualization layer (e.g. Boot2Docker for OSX or Windows) which:

- creates a thin Linux-VM where running the docker daemon
- installs the docker client on the host
- talks via HTTP/REST to the daemon

Under the hood

Namespaces

Like 'chroot', namespaces allow processes to see some aspects of the operating system independently.

This provides a layer of isolation: each container receives its own network stack and process space, as well as its instance of a file system.

Control groups

Is a Linux kernel feature to limit the resource usage of certain processes.

It's a more flexible 'nice'.

For example, limiting the memory available to a specific container.

Union file systems

The filesystem of an image consists of a series of layers. Docker makes use of union file systems to combine these layers into a single image.

The **boot file system** contains the bootloader and the kernel.

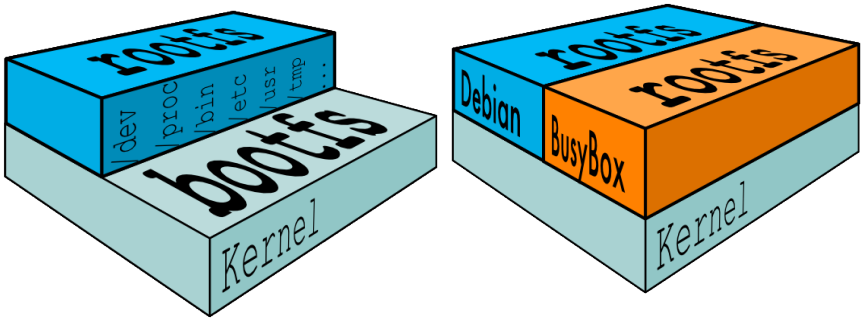
The **root file system** includes the typical directory structure of a Unix-like operating system.

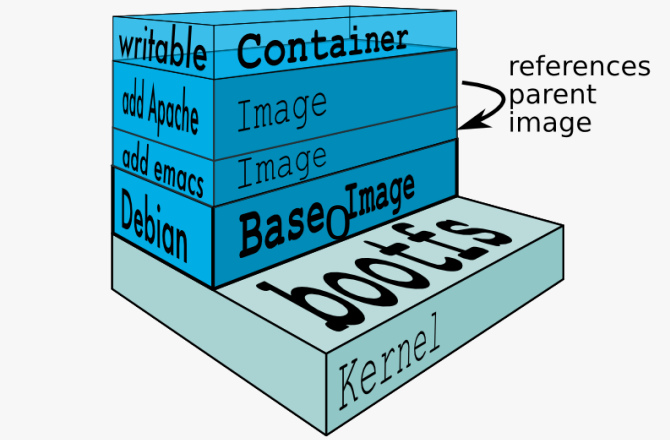
In general, the contents and organization of the root file system are usually what make software packages dependent on one distribution versus another. Docker can help solve this problem by running multiple distributions at the same time.

Docker images are built from these base images using a simple, descriptive set of steps. Each instruction creates a new layer. These instructions are stored in a file called a **Dockerfile**.

There is no need to distribute a whole new image, just the update, making distributing Docker images faster and simpler.

Docker can make use of several union file system variants including: AUFS, btrfs, DeviceMapper, ...





http://developerblog.redhat.com/2014/09/30/overview-storage-scalability-docker/

Example 1

Run a simple container on a given image

List available images:

```
$ docker images
REPOSITORY      TAG         IMAGE ID            CREATED         VIRTUAL SIZE

centos          centos6     25c5298b1a36        10 weeks ago        215.8 MB
```

Run the container on the image providing a basic command

```
$ docker run centos:centos6 echo "Hello world"
Hello world
```

Example 2

Starting from a Dockerfile, build a simple image and run a container on it *interactively*
The base image is downloaded from the official registry (Docker Hub)

```
$ more /etc/issue
Ubuntu 12.04.5 LTS \n \l
```

HOST

```
$ cat Dockerfile
FROM centos:centos6

$ docker build --tag test/centos6:1.0 .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM centos:centos6
centos:centos6: The image you are pulling has been verified
511136ea3c5a: Pull complete
5b12ef8fd570: Pull complete
a30bc9f3097e: Pull complete
Status: Downloaded newer image for centos:centos6
 ---> a30bc9f3097e
Successfully built a30bc9f3097e

$ docker images
REPOSITORY        TAG      IMAGE ID          CREATED           VIRTUAL SIZE
test/centos6      1.0      25c5298b1a36      10 weeks ago      215.8 MB

$ docker run --interactive --tty test/centos6:1.0 /bin/bash
[root@984f50e8f28c /]# more /etc/issue
CentOS release 6.6 (Final)
```

GUEST

Example 3

Run a container mounting a volume in read/write mode from the host

```
$ mkdir /tmp/test
$ touch /tmp/test/data.txt

$ docker run -it -v /tmp/test:/opt/:rw centos:centos6 /bin/bash

[root@5b1e81c4d70e /]# ls -l /opt/data.txt
-rw-rw-r-- 1 1000 1000 0 Feb 16 21:34 /opt/data.txt
[root@5b1e81c4d70e /]# echo "Hello" > /opt/data.txt
[root@5b1e81c4d70e /]# exit
exit

$ cat /tmp/test/data.txt
Hello
```

Docker networking

A virtual bridge interface (**docker0**) is created at the host level when the service is started

For each container two “peer” interfaces are created:

- one at the host level (e.g. **veth5fb53c8**)
- another which becomes the **eth0** for the container

Communication between containers happens through the bridge interface that automatically forwards packets between any other network interfaces that are attached to it.
By binding every veth* interface to the docker0 bridge, Docker creates a virtual subnet shared between the host machine and every Docker container.

What is docker-registry?

Docker registry is basically a repository for images.

You can manage images:

- by using the public repository, DockerHub
- or by using by using a private repository

Every single command in a Dockerfile yields a new Docker image with an individual id.

This commit can be tagged for easy reference with a Docker tag.

- Tags are the means to share images on public and private repositories.

Securing docker registry

Private Docker registry comes without support for authentication:

- Anyone who knows registry URL can push their own Docker images. So we need some authentication.

We setup nginx in front of Docker registry for basic authentication and SSL encryption.

Tag

- When you build a Docker image using a Dockerfile, you can set a tag for the final image by passing the parameter **-t <tag>**.
- The syntax for a tag is **repository:[tag]**.
- You can share a **demo** image to Docker Hub by first building, and then pushing it:
  > docker build -t <username>/demo .
  > docker push <username>/demo

  where <username> is the name registred in DockerHub by the user.
- In case you want to use a private registry, you need to set the URL to your private registry as the username. First tag it accordingly and then push it, assuming your registry is listing on localhost port 5000:
  > docker tag <username>/demo localhost:5000/demo
  > docker push localhost:5000/demo

Push/Pull images example


```
$ docker images
REPOSITORY            TAG          IMAGE ID         CREATED          VIRTUAL SIZE
italiangrid/base      latest       c0493b41d761     9 days ago       778.1 MB

$ docker tag -f italiangrid/base cloudvm128.cloud.cnaf.infn.it/italiangrid/base

$ docker push cloud-vm128.cloud.cnaf.infn.it/italiangrid/base
The push refers to a repository [cloud-vm128.cloud.cnaf.infn.it/italiangrid/base] (len: 1)
Sending image list
Pushing repository cloud-vm128.cloud.cnaf.infn.it/italiangrid/base (1 tags)
Image 511136ea3c5a already pushed, skipping
Image 5b12ef8fd570 already pushed, skipping
Image 510cf09a7986 already pushed, skipping
Pushing tag for rev [e8b9b5527607] on {https://cloud-
vm128.cloud.cnaf.infn.it/v1/repositories/italiangrid/base/tags/latest}

$ docker pull cloud-vm128.cloud.cnaf.infn.it/italiangrid/base
Pulling repository cloud-vm128.cloud.cnaf.infn.it/italiangrid/base
e8b9b5527607: Download complete
511136ea3c5a: Download complete
5b12ef8fd570: Download complete
510cf09a7986: Download complete
Status: Downloaded newer image for cloud-vm128.cloud.cnaf.infn.it/italiangrid/base:latest
```

Our CI and CD infrastructure

Several components are involved in the setup of our Continuous Integration (CI) and Continuous Deployment (CD) workflow

Jenkins — Continuous Intergation service

Revision control system

Cloud provisioning stack

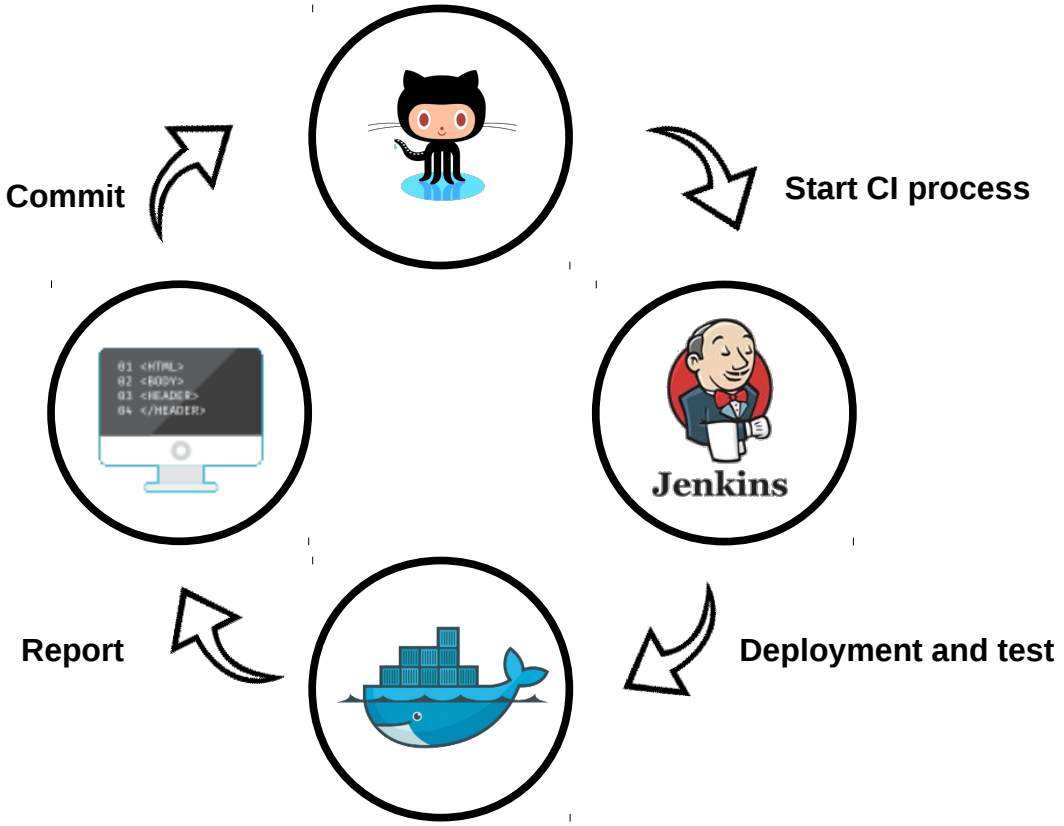
OS platform running Docker

Jenkins and Github

Jenkins provides continuous integration services for software development. Builds can be started by various means, including being triggered by commit in a version control system, scheduling via a cron-like mechanism, building when other builds have completed,

GitHub is a web-based Git repository hosting service, which offers all of the distributed revision control and source code management (SCM) functionality of Git as well as adding its own features. Unlike Git, which is strictly a command-line tool, GitHub provides a web-based graphical interface

Continuous Integration (CI)



OpenStack and CoreOS

OpenStack is a free and open-source cloud computing software platform. Users primarily deploy it as an infrastructure as a service (IaaS) solution. The technology consists of a series of interrelated projects that control pools of processing, storage, and networking resources throughout a data center—which users manage through a web-based dashboard, command-line tools, or a RESTful API.

CoreOS is an open source lightweight operating system based on the Linux kernel. CoreOS provides no package manager as a way for the distribution of applications, requiring instead all applications to run inside their containers.

Why CoreOS?

At the present time there is an issue in Docker which prevents deleted containers to free mapped disk space. It's related to a kernel problem with DeviceMapper which affects the RedHat family:

https://github.com/docker/docker/issues/3182

Based on our experience, CoreOS is the most stable distribution for using Docker in production:

- based on btrfs file system
- kernel version (3.18)

StoRM deployment in CI with Docker

A deployment test implies:

A full installation of the StoRM application in a clean environment
A step to configure the application components
The execution of a dedicated testsuite against the deployed application

Two kinds of deployments:

Clean: install the last version of StoRM and test it
Update: install the previous version of StoRM, update to the last version and then test it

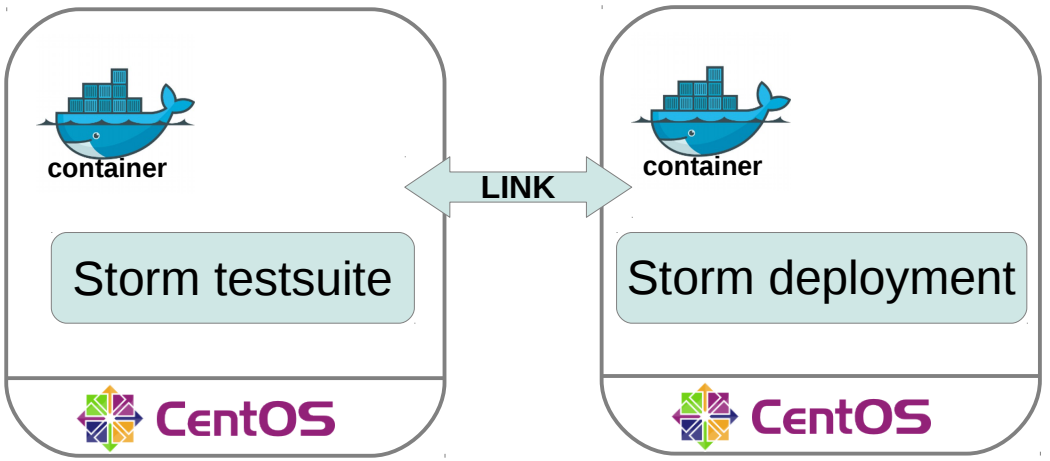
The goal is to perform deployment tests in Docker integrating the process with our Continuous Integration workflow

For our purposes, the containerazation of StoRM is achieved by including more services into a single container

StoRM deployment tests

A deployment test is executed
by linking two containers:

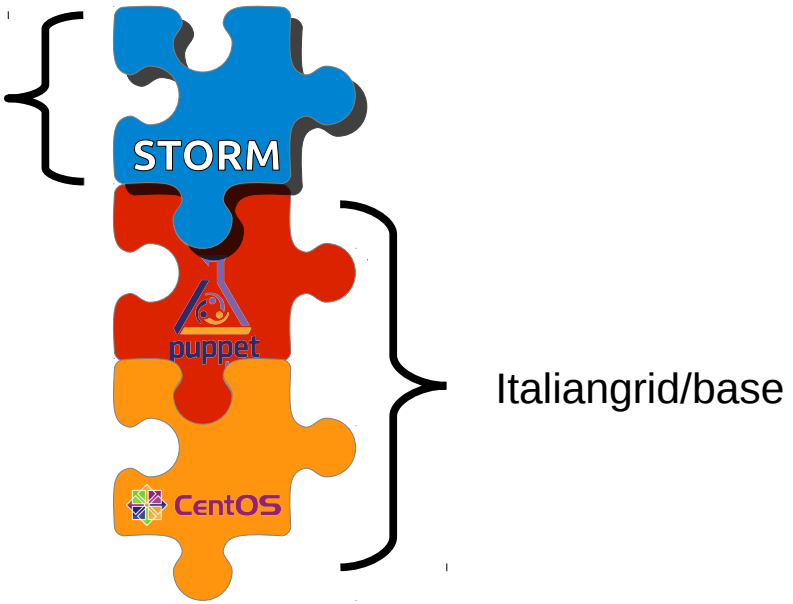
- The container which deploys the application runs in background
- The container which runs the testsuite starts in foreground and waits until the StoRM service is properly configured and active on the first container
- Inter-container communication
- Ports for required services exposed when the service image is defined



StoRM deployment image

Dockerfile

```
FROM italiangrid/base
ADD . /
RUN chmod +x /setup.sh
RUN /setup.sh
EXPOSE 8080 8085 8086 8443 8444 9998
# setup deployment script
RUN chmod +x /deploy.sh
# setup StoRM daemons
RUN chmod -R +x /daemons
```



StoRM testsuite image

Dockerfile

```
FROM italiangrid/base

# add and run setup

ADD . /

RUN chmod +x /setup.sh

RUN /setup.sh

# setup for the tester user

WORKDIR /home/tester

ENTRYPOINT /setup_testsuite.sh
```

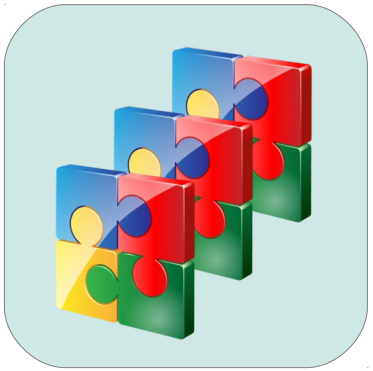
Images management in CI

Build slave for Docker images

Authentication required

PUSH

LOCAL IMAGES REGISTRY



PULL

Slave

Slave

Example of containers linking


```
# run StoRM deployment and get container id

deploy_id=`docker run -d -e "STORM_REPO=${STORM_REPO}"
-e "MODE=${MODE}" -e "PLATFORM=${PLATFORM}" \

-h docker-storm.cnaf.infn.it \

-v $storage_dir:/storage:rw \

-v $gridmap_dir:/etc/grid-security/gridmapdir:rw \

-v /etc/localtime:/etc/localtime:ro \

${REGISTRY_PREFIX}italiangrid/storm-deployment-test \

/bin/sh deploy.sh`
```

Example of containers linking


```
# get names for deployment and testsuite containers
deployment_name=`docker inspect -f "{{ .Name }}" $deploy_id|cut -c2-`
testsuite_name="ts-linked-to-$deployment_name"
# run StoRM testsuite when deployment is over
docker run --link $deployment_name:docker-storm.cnaf.infn.it \
-v /etc/localtime:/etc/localtime:ro \
--name $testsuite_name \
${REGISTRY_PREFIX}italiangrid/storm-testsuite
```

Jenkins slave setup for docker

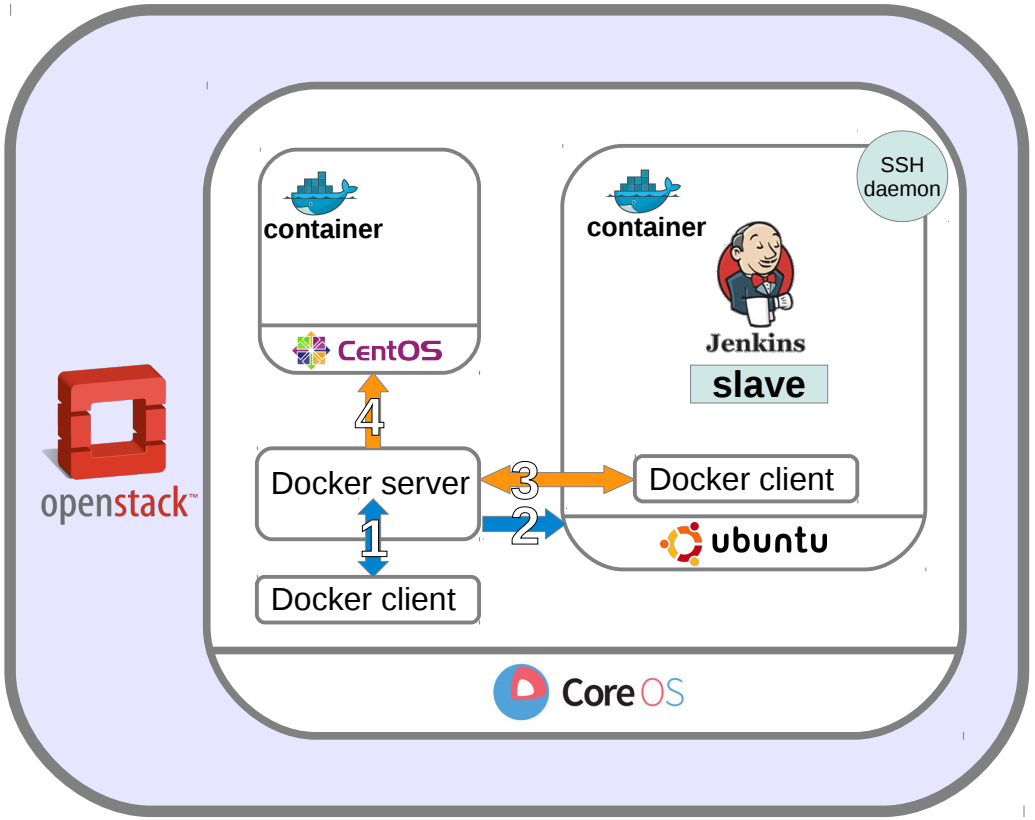
CoreOS VM provisioned by using OpenStack cloud facilities

In CoreOS everything runs in containers.
There is not a dedicate package manager

Docker client on CoreOS (the host) tells the server to download a prebuilt Jenkins slave image from our local docker registry

A container is created on the host where running the Jenkins slave:

- The Jenkins slave provides a ssh daemon to accept incoming jobs from the Jenkins server
- Docker client on the slave talks to the docker server which runs on the host
- The “containerized” slave runs jobs in turn as containers directly on the host



Next steps

Understand how to integrate GPFS and Docker

- Access GPFS filesystem from containerized StoRM (now ext4)

Containerized each service independently

- At the present time all StoRM services run in the same container

Thanks