Technologies Investigation:

Docker:

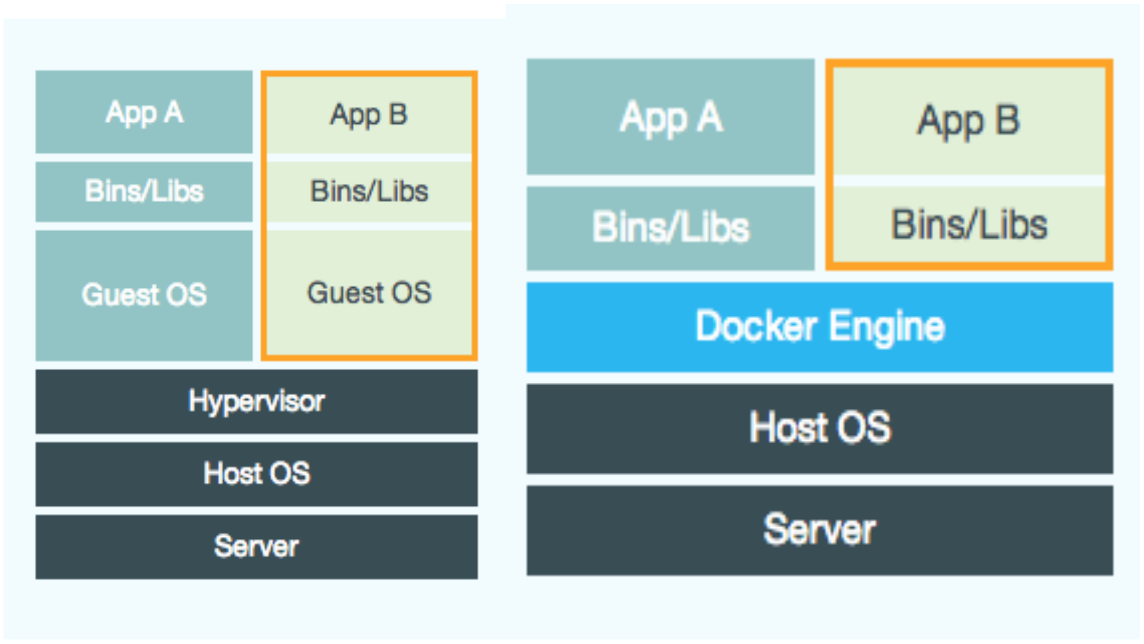
Docker is described as a lightweight virtualized environment for portable applications. It is supported on Linux only and is still in pre-release phase. We will describe how it works and how it can be used in the context of this thesis, in the next subsections.

How it works?

It leverages the ability of the Linux kernel to provide *cgroups* for resource isolation that doesn't require any instantiation of virtual machines. Cgroups is the product of the engineers at Google [1] and its primary purpose was to limit and isolate resource usage of process groups. More specifically it provides 4 features:

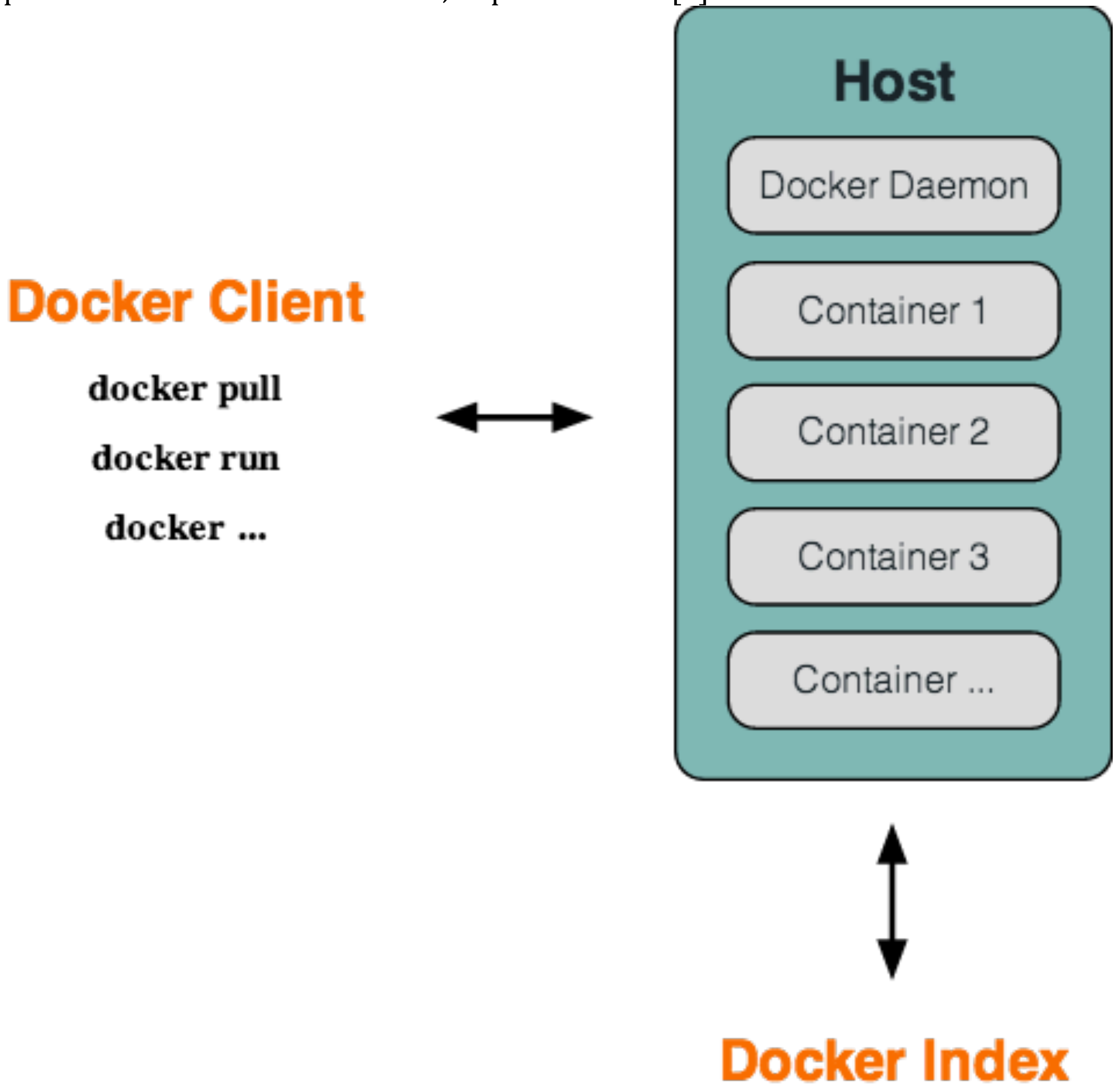
1. Resource limiting (i.e.: group can't exceed 52mb of ram).
2. Prioritization of the CPU and various buses.
3. Accounting for resource consumption.
4. Control.

Namespace isolation is also provided, which allows isolating completely an application from the operating system environment, and even from the mounted file systems. Worth mentioning is one of the applications of cgroups, namely LXC or LinuX Containers. *"LXC are an operating system-level virtualization method for running multiple isolated Linux systems on a single control host."*[1] As a matter of fact, Docker is actually built on top of LXC. It can be seen as an image manager, which offers deployment services. More specifically it extends the LinuX Containers by providing a high-level API that enables lightweight virtualization to run processes in complete isolation from the host system.



[1] Taken from Wikipedia: http://en.wikipedia.org/wiki/LXC
[2] Taken from Wikipedia: http://en.wikipedia.org/wiki/UnionFS

In the two pictures above, we can see the difference between a virtual machine (on the right) and Docker. We can observe that the Hypervisor and the guest OS for each application is absorbed into the Docker Engine, which provides a unique platform to run independent applications and their required Libs and Bins. This is achieved by running the application as an isolated process in userspace on the host operating system, thus sharing the kernel with other containers. In the following figure we will present the Docker's architecture, as presented on [2].



As we can observe, it uses client-server architecture, Docker Client being the client and the Docker Daemon being the server. The Daemon is responsible for building, running and distributing the Docker containers. It does not require to be executed on a dedicated host, nor for the client. The communication between the Client and the Daemon can be done through sockets or through a RESTful API.

According to their documentation, in order to understand the inner workings of Docker we need to introduce three concepts: Docker Images, Docker Registries, and Docker Containers.

Docker Images: They are the build component of Docker, and are in the format of a read-only template. These templates represent the internal composition of a Docker Container (i.e.: Ubuntu + Apache + custom-made web app). These images are used to create Docker containers.

More specifically, each image consists of a series of layers and via the use of union file systems; these layers are combined into a single image. Briefly, union file system (UnionFS) is a filesystem service that implements a union mount for other file systems[2]. Union mounts consist of multiple file systems being mounted at the same time, which appears to be one unique file system[3]. In other words, it allows branches (files and directories of separate file systems) to be overlaid transparently and forming a single coherent file system.

This is the primary factor that makes Docker so lightweight, since by leveraging the power of the union file systems, it is not necessary to replace an entire image when an update is made. Rather when you make a change to the Docker Image, a new layer gets built, and therefore to deploy the update, there is no need to distribute the complete image, rather just that layer (or the diff).

Docker Registries: Are simply repositories containing Docker images, where you can upload or download images. They can be private or public (i.e.: Docker Hub).

More specifically, it works as a version control would, you can push an image to a public repository, and/or pull one.

Docker Containers: Containers holds anything an application needs to run, and they are created from Docker Images. The following operations can be applied to Docker Containers: Run, Started, Stopped, Moved and Deleted. They are the run component of Docker.

More specifically a container consists of an operating system, user-added files and meta-data. In this context, the image from which the container is built contains the following information: what process to run when the container is launched, and the corresponding configuration data. Since as mentioned above, the image itself is read-only, when Docker runs a container from an image, it adds a read-write layer on top of the image, leveraging the union file system, in which the application can run.

In a nutshell, you can build Docker Images that hold your application, which can be shared on a public registry (Docker Hub) or private (local), and you can create Docker Containers based on those Docker Images. Finally you can run the application contained in the Docker Images, by running the corresponding Docker Container.

Flow of events of running a Docker Container:

1. Pull the image. (locally if it exists)
2. Create the container.

---

[2] Taken from Wikipedia: http://en.wikipedia.org/wiki/UnionFS

[3] Taken from Wikipedia: http://en.wikipedia.org/wiki/Union_mount

3. Allocate fs and mounts a read-write layer.
4. Allocate network/bridge interface.
5. Sets up IP address.
6. Execute the process specified in the run command.
7. Captures and provides application output.

DEIS:

DEIS is a lightweight, flexible and powerful application platform that deploys and scales Twelve-Factor apps as Docker containers across Clusters of CoreOS machines. DEIS can deploy any application or service that can run inside a Docker container, the only caveat is that in order to be scalable applications must follow Heroku's twelve-factor methodology and store state in external backing services. Currently it can run locally (using Vagrant), or in Rackspace, EC2 and bare metal.

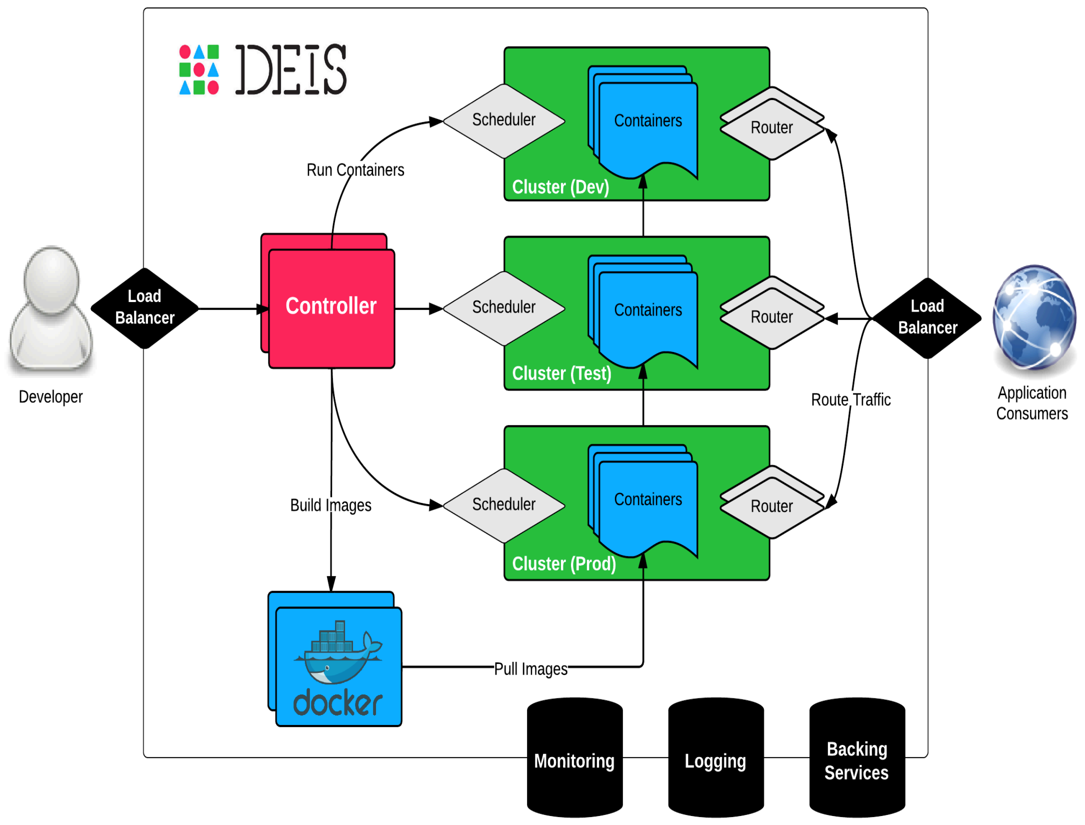
AppScale:

AppScale is an open source implementation of Google App Engine, the key feature of this, it that it decouples the GAE application from the Google Cloud platform and enables you to run on said application on any cloud provider.

How can it be used in the context of this thesis?

I think that the most practical perspective would be to extend the current DEIS implementation to allow deployment on volunteered computing resources. This would provide a mean to deploy any Docker application on volunteered cloud, and if they comply with the 12 factor methodology [5], leverage the scalability features provided by the DEIS platform.

As I can understand from my current knowledge, DEIS operates on dedicated hosts such as an instance of Rackspace or EC2. More specifically, they can be deployed on platform for which support of CoreOS is provided: as shown in the following figure.



Let's take a quick look at the current requirements we have set for ourselves:

Requirements:

1. Does not run on a dedicated host, preferably as a software daemon.
2. As of now only on Linux/OS X.
3. Homogeneous Nodes versus Heterogeneous Nodes.
    a. Homogeneous is easier from a management/monitoring point of view.
    b. Heterogeneous makes the separation between computations very clear.
4. Do not want to create a new API to build applications on top of (as PaaS), rather use an open source one.
5. No single-point of failure.
6. Makes use of volunteered computing resources.

From this list of requirements we can see that number 2, 3 and 4 are fulfilled. And in respect to number 3, all nodes are homogeneous. But requirement 1, 5 and 6 are not. DEIS do run on dedicated hosts, namely each node in a cluster refers to a CoreOS machine, therefore requirement 1 is not fulfill. There exist a single-point of failure, namely the controller; since it orchestrates everything that occurs in the PaaS, therefore requirement number 5 is not fulfilled. And since the idea of making use of volunteered computing resources is novel in this context, the thesis main objective is to fulfill this requirement.

Questions that arises from this investigation:

Within the context of this investigation, several key questions arose and it is imperative to reflect and to provide insightful answers. Here is a list of questions that we will discuss in the next meeting:

1. How modular and decoupled is the current implementation of DEIS? In the sense that the current architecture heavily relies on the CoreOS dedicated host platform, by substituting this platform for a software daemon, will we hit a massive wall? By this we mean do we have to modify the majority of the code base? If that is the case, it will be more attractive for the scope this thesis to write our own implementation, which will be heavily inspired from the current DEIS architecture.
2. Similar question towards how can we remove the Single-Point of failure that is the Controller? Can we effectively distribute the workload and application logic through independent homogeneous nodes?
3. Since it is used to deploy Docker applications, how do we create a software daemon that is isolated from the hostOS to provision resources to the Cluster? My initial idea was to use a Docker application as the software daemon, since it isolates itself from the hostOS and could provision an isolated set of resources. But in that context, it implies running Docker applications inside a Docker application software daemon… We need to assess the feasibility.

References

[1] Jonathan Corbet (29 May 2007). "Process containers". LWN.net.

[2] Docker's Website: www.dockers.io

[3] DEIS's Website: www.deis.io

[4] AppScale's Website: http://www.appscale.com/ or https://github.com/AppScale/appscale

[5] Twelve Factor's Website: http://12factor.net/