Is Docker container based virtualisation useful for Scientific purposes at DESY?

Joshua De Souza, University College London, United Kingdom

22 September 2015

Supervisors: Karsten Schwank & Andreas Gellrich

Abstract

The aim of this report is to present an overview of what Docker is and it's relevance to scientific computing at DESY. The report includes an example of how Docker may be used to "dockerize" the dCache application. dCache is a large data storage and retrieval system with storage resources located at several locations internationally [6]. ATLAS and CMS experiments both use dCache to archive data generated from experiments. In addition some other potential use cases where Docker may be useful are outlined. This report will focus on what Docker is and what it may be used for. However further exploration of how Docker may be used for scientific purposes at DESY is required to determine if it should be used and where it is appropriate.

Contents

1 Introduction

Docker is an open source platform for automating the deployment of applications [2]. It was developed as a project within dotCloud, a platform as a service company. This report is a brief overview of how Docker may be useful for scientific purposes at DESY. Some of the advantages and limitations of Docker were investigated. My project focused on how dCache was made into a ready made, ready to use package. The report summarises with the outcome of the project and the outlook for Docker.

1.1 Computer Virtualisation

Virtualisation can allow for better use of underlying computer resources. [3]. Physical hardware without any virtualisation layer can make managing computer resources effectively difficult. An example of this is having an internet server on a single machine, a mail server on another, a file server on yet another and so on. Each server carries out only one function. Consequently each server would have an excess of resources to allow for growth of the services and processes it carries out.

Virtualisation is using software to create an abstraction from underlying resources. It can be used to run multiple servers on a single machine. The machines resources are used more effectively as the underlying hardware resources can be managed more effectively using this intermediate software layer. This is as the virtualised hardware can be managed more flexibly amongst the different servers on the same machine.

There are three broad categories of virtualisation [3]. The first is server virtualisation and an example of a use case for this type of virtualisation is as in the above example. The second type of virtualisation is storage virtualisation.

The third broad category is client or desktop virtualisation, which makes managing the end users machine easier and provides greater security. The virtualised desktop can be hosted on a remote server or locally. One way of employing desktop virtualisation is known as application virtualisation. This is where an application is run on an end users computer using the host operating system to virtualise the application [3].

Application virtualisation can be achieved through container based virtualisation. Software known as containers control how the application that runs within it interacts with the desktop operating system [3]. Here instead of the physical hardware being virtualised, the containerised application shares the host operating systems kernel.

1.2 Docker - container based Virtualisation

Docker is an open source project released by a platform as a service company called dotCloud [4]. Docker implements container based virtualisation, also known as operating system level virtualisation. Two important components of Docker virtualisation are images and containers. Images can be thought of as being static while containers are dynamic and are instances. In effect an application, dependencies and configuration are encapsulated in a read only snapshot called an image. A read-write layer is added on top of the static image. Here the service is carried out and changes to the container can be committed to the image.

1.3 Containers vs. Virtual Machines

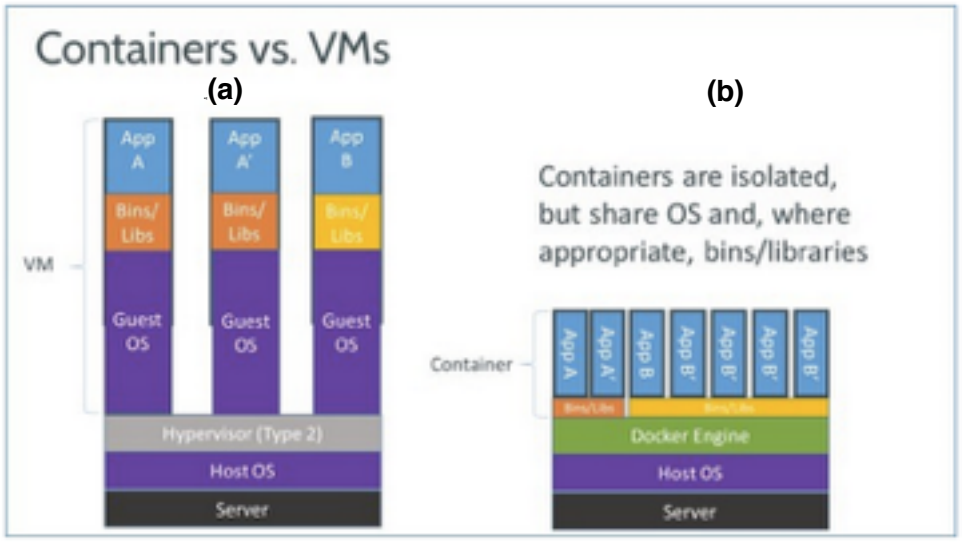

Figure 1: (a) Virtual Machine setup (**b)** Docker container based virtualisation setup

Virtual machines use hypervisors to emulate virtual hardware [5]. In the above model (figure 1a) a thin layer called a hypervisor emulates hardware allowing different guest operating systems to run on top of the Host OS. Comparatively the docker setup (figure 1b) uses shared operating system resources. The docker containers utilise the Linux instance allowing the docker environment to be composed of less. The advantage of this is that docker containers cost less processing overhead. This makes them inexpensive and means many containers can be run on the host operating system at the same time. A downside of this is that “dockerized” applications have to be supported by Linux as they must run on top of a Linux distribution host operating system.

2 Docker Architecture

2.1 Client-Server architecture

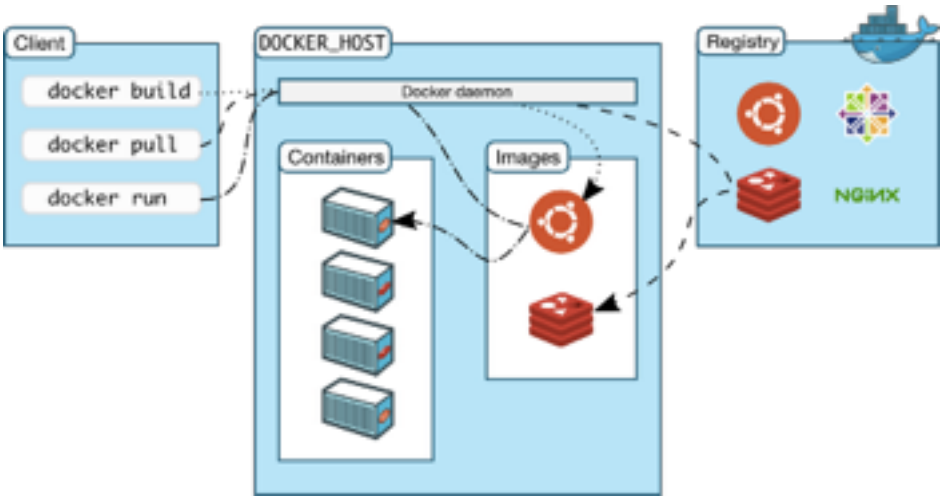

Figure 2: Client-server architecture with Docker registry

Docker uses a client-server architecture [6]. The Docker server can be on the host machine or can be hosted remotely. The Docker server is daemonized and so runs as a background service. The Docker client is the end users interface with the docker daemon. The client and daemon communicate by either using a RestAPI or through sockets. The docker daemon is responsible for building and distributing images and running containers. Running docker containers means managing them. This includes being able to create, start, stop containers and more.

2.1 Docker Components

2.1.1 Images

Docker makes use of three important components. An image is a read-only template from which an instance called a container can be created and started from. The base image must be based on a Linux operating system.

Docker uses a Union File System [6]. Files and directories can exist as part of a layer and several layers can be transparently overlaid forming the complete filesystem. A Docker image can be composed of several layers. The layered system allows for efficient version control and faster shipping and deployment.

A consequence of the layered structure is that new images can be shared over a network more efficiently. As an example, if an image is pulled from a public repository and changes are committed to the image, pushing the new image back to the public repository will only require the transfer of the change the new image is composed of.

A Dockerfile is used to create a new Docker image. The file contains a set of instructions which must start with the FROM command, which specifies the base image the new image will be built from. Some instructions are: ADD a file, RUN a command, EXPOSE a TCP/IP port within the internal container network. The image can be built using the Docker BUILD command. The BUILD command uses the Dockerfile along with the builds context to create the new image.

The docker build context are files that are used to build an image described by Dockerfile(s). The build context can be files located in the same directory as the docker file that contains instructions that act on these files, or at other specified locations [7]. For example the COPY <source>… <destination> instruction will act on files within the same folder as the Dockerfile that contains the instruction. However the WORKDIR instruction can be added before the COPY instruction in the same docker file to set a different folder for the COPY instruction to copy its <source> from.

2.1.2 Registries

Docker registries host repositories [6]. The repositories contain a collection of tagged images. The registries can be private or public. Dockerhub is the Docker public registry that contains a large collection of images that can be pulled to a local destination.

2.1.3 Containers

Containers are running instances of an image that are comprised of an operating system, user added files and meta-data [6]. When a container is started, a read-write layer is added to the image it is started from. The container has its own internal network interface that allows communication . Docker containers are designed to run isolated processes. The lifetime of a container is defined by the ENTRYPOINT service. The ENTRYPOINT instruction is written within the Dockerfile.

2.2 Linking containers and creating volumes

Aside from being able to communicate via mapping network ports externally, links can be used to share select information between containers [8]. Links are TCP/IP ports that allow connection information and service discovery to be shared between linked containers. The information is communicated through environment variables and updating the /etc/host file [9].

Volumes are free floating file-systems that evade the UnionFS [10]. A mount point can be created between containers or from a container to the host [11]. They are initialised when the container is created. They are designed to persist even after a container is removed. This allows volumes to be shared and reused among containers. Can be used to store state from computations between transient container. Multiple mount points can be created and the volume will be initialised when the container is created. In addition you can also mount a data volume from the host machine into a container. It is possible to mount volumes in read-write and read-only modes by specifying options.

3 Itemising dCache

3.1 Creating directory structure and dcache-docker git repository

The following directory structure was created in /space/workspace/dcache-docker/dcache on the local machine using the *mkdir* command.

```
.
l____master
   l l____base
   l l____db
   l l____standalone
   l l____dCacheDomain
```

The *dcache-docker* folder was made into a Git repository with a master branch called *master*. Git is a popular version control tool [12]. It was used to collaboratively to create the Dockerized dcache and postgresSQL database with my colleague Karsten Schwank. A Dockerfile was created in each sub-directory of the master directory. The making of the postgreSQL database image and container is not included in this report.

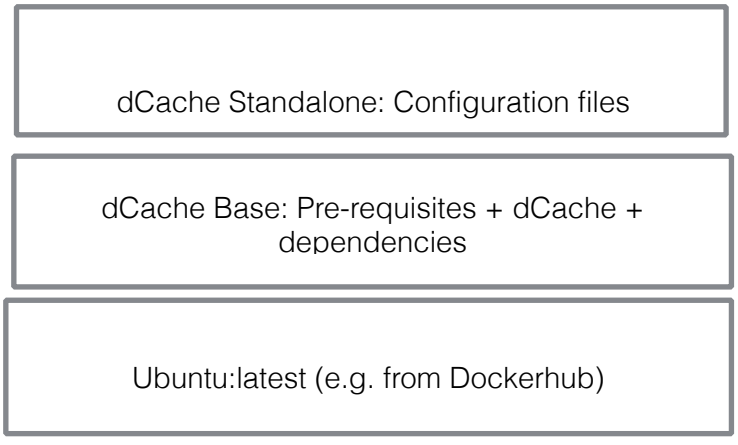

Figure 3: Images layered to create overall dCache Image

3.2 Creating the dCache Image

The dCache image can be thought of as three separate images. The first image: Ubuntu:latest which was pulled from Dockerhub. The image that is overlaid on top is the dCache Base image. The dCache base image contained dCache, its pre-requisites and dependencies. The dCache standalone layer was the top-most layer, which contained the configuration files required by dCache

3.2.1 dCache Base Image

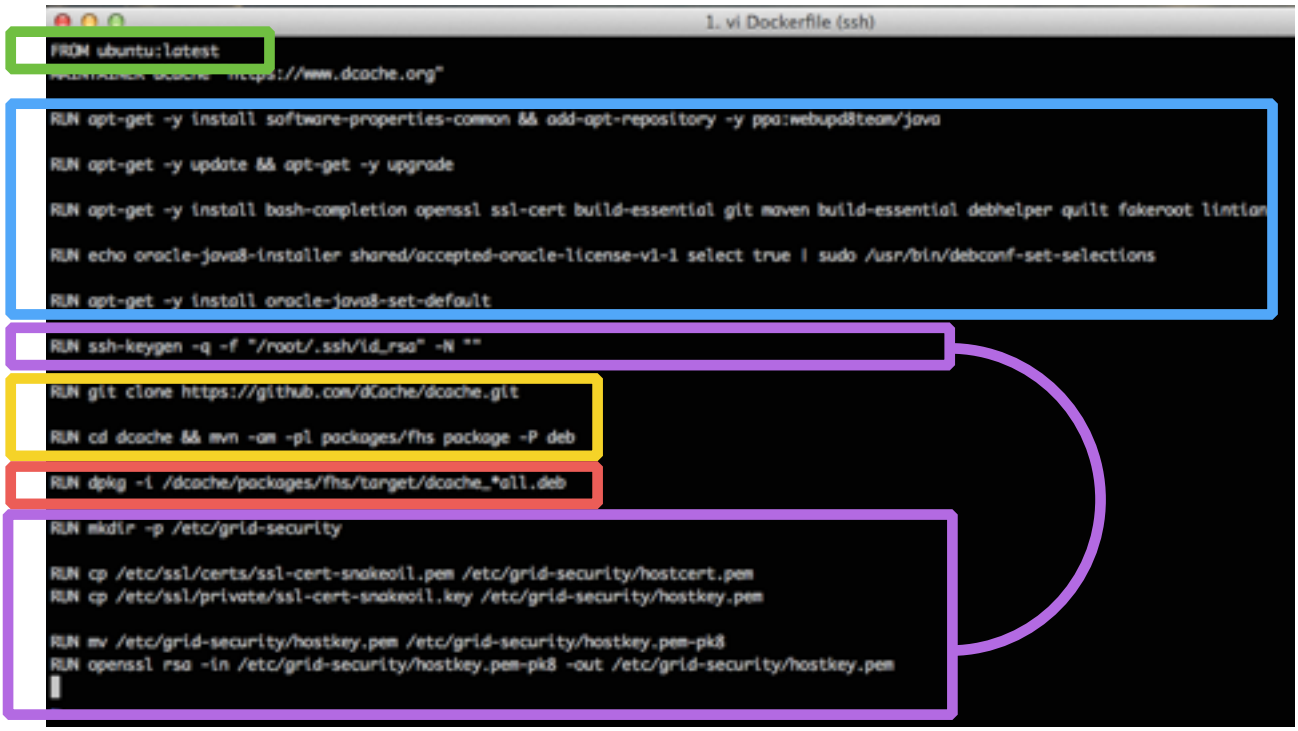

Figure 4: dCache Base Image

The FROM instruction tells Docker which Image to base the new image to be built on. Here the base image for the dCache Base image to be overlaid on is ubuntu:latest.

The other Docker instruction used in this dockerfile is RUN. This completes the command that follows it in a new layer, which is then committed to the image [13].

The *base* directory Dockerfile contained prerequisites for the dCache installation (blue in figure 4). This included installing the build automation tool Maven and other packages from Ubuntu repositories using the software user interface - Advanced Package Tool (APT). APT was also used to install Java 8 which is necessary to compile and run dCache.

Public and private user keys were generated and inserted into ~/.ssh/ by typing in ssh-keygen into command line (purple in figure 4). A SSL certificate was created and placed into /etc/grid-security/ (purple in figure 4).

The same Dockerfile also contained instructions for building dCache. The git clone command copies source code from the dCache development teams GitHub repository. This creates a sub-folder with the source code. The *cd dCache* command changes the working directory to the dCache folder and maven is called using *mvm -am -pl*.

Finally the .deb packages that were contained in the dcache folder were installed using the low level Debian package management tool dpkg [14].

3.2.2 Standalone Image

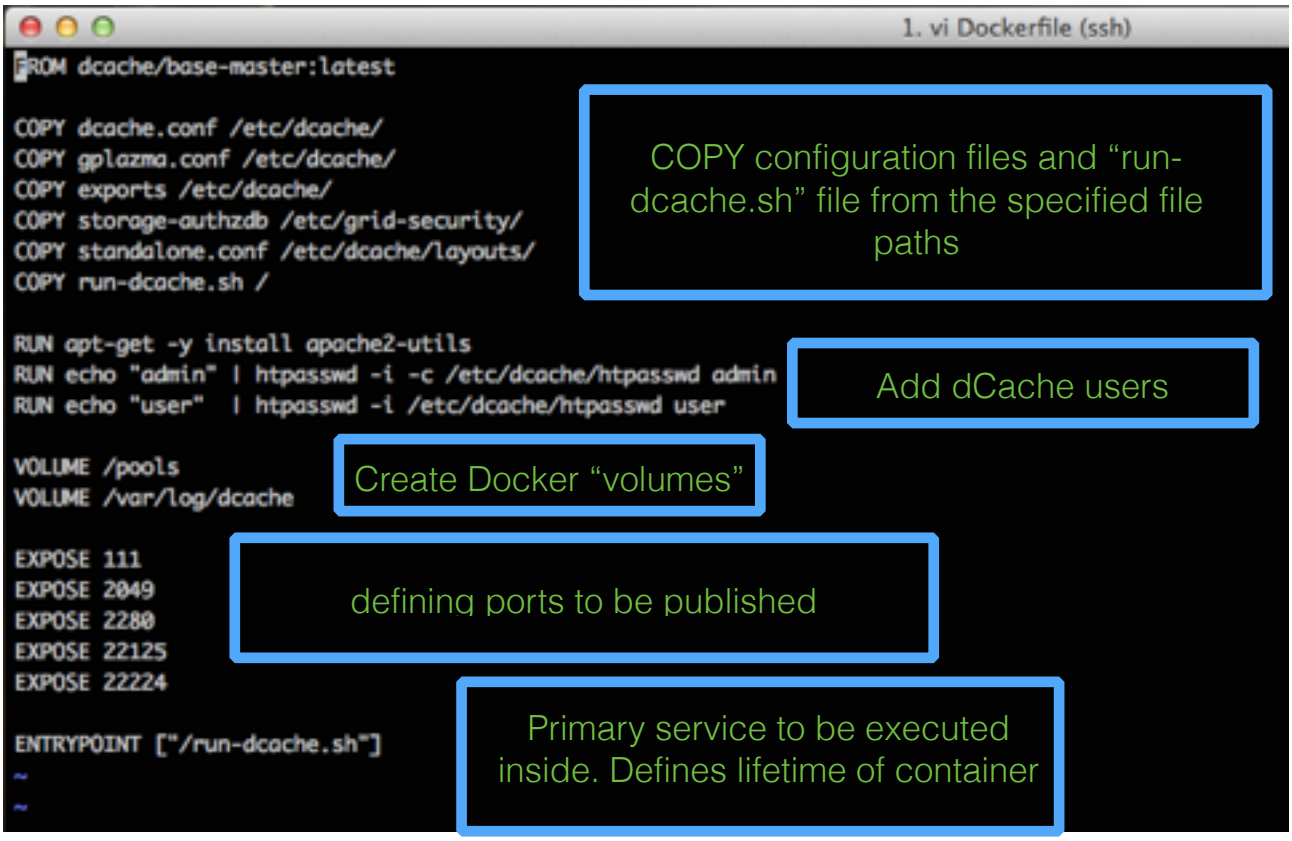

Figure 5: Standalone Image

The standalone directory Dockerfile contains instructions to COPY various configuration files for dCache, from the specified file paths.

The apache2 package was also installed using APT. The command *echo "admin" | htpasswd -i -c /etc/dcache/htpasswd* creates the username admin for the use of HTTP authentication. Likewise the subsequent command creates a username called *user*.

The VOLUME command was used to create a mount point between the host and running container The */pools* and *var/log/dcache* volumes are located on the host.The EXPOSE command was used to define ports to be published. This allows docker containers to communicate. The ENTRYPOINT instruction specifies the process that will be run inside the container. The process that is run is */ run-dcache.sh*.

The *docker build* command was used to build the *dcache/standalone-master* image.
The following command was typed into terminal to build the image: "*docker build --no-cache --rm -t dcache/standalone-master .".*

The --rm removes flag removes intermediate containers after the build. The -t flag is used to name the repository the image belongs to. The "." (full-stop) specifies that the BUILD instruction will build from the current working directory.

The *docker build* instruction works by using the instructions inside the chosen Dockerfile and the builds context [15]. The first instruction is the FROM command which defines the base image the new image will be layered on top of. If the base image is not already built, the docker daemon will first look locally to build the base image. If the base image is not present locally, it will search the Docker remote public registry Dockerhub. In this case the base image was not built, but the build source was local. So the daemon finds the directory that is the build source and builds the base image. This “base” image itself is built from an ubuntu:latest base image, and so here “base” is used as a relative term. The ubuntu:latest image is pulled from Dockerhub as the image is not present locally. The final image is named in the format <parent directory>/<directory Dockerfile is within>-<branch>. The build process sends the builds context recursively to the daemon [15].

3.3 Docker Compose tool for multi-container configurations

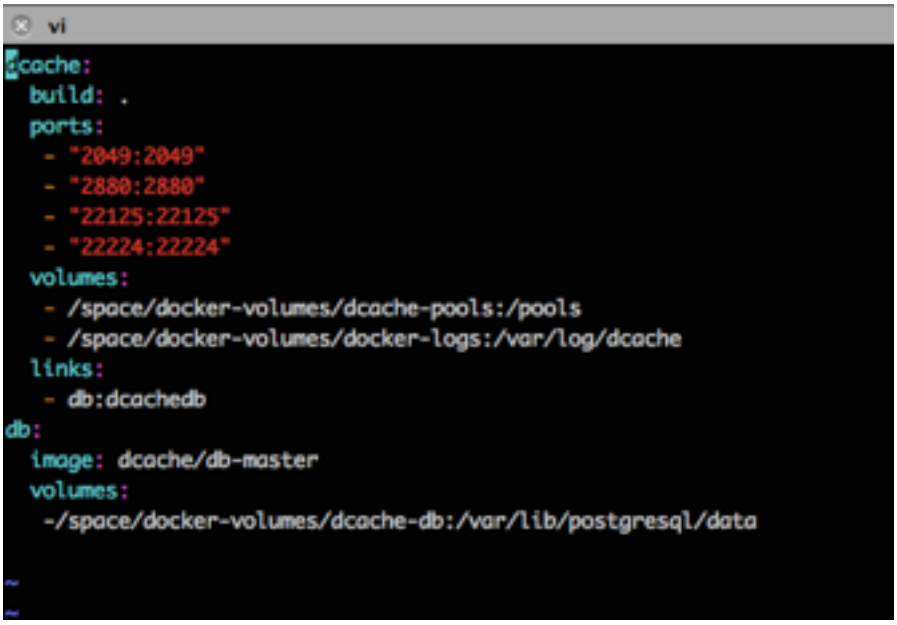

Figure 6: Yaml file for multi container-configurations

Docker Compose is a tool which can be used to define multi-container configurations [16]. Instead of creating and configuring containers individually, by defining a YAML file. The Compose .yml file is structured into paragraphs. Each paragraph describes a single container or service. Each service must contain either a BUILD or IMAGE instruction [17].

Ports and volumes in the format <host>:<container> were specified for the dCache service. The defined volumes create a mount point from */pools* container namespace to the host */space/docker-volumes/dcache-pools* namespace on the host. Similarly the */var/log/dcache* container namespace is mounted to */space/docker-volumes/docker-logs* on the host. A link was created between the dCache container and the PostgresSQL container in the format <container name to be linked too>:<link name>.

For the database image, the base image *dcache/db-master* was called. A volume was defined to mount */var/lib/postgresql/data* to */space/docker-volumes/dcache-db*.

Finally *docker-compose up* was run in command line which creates and starts both the dCache and database containers.

4 Summary and Outlook

A first version of a Portable, “lightweight” dCache service was created. This indicates Docker may be a viable way of virtualising applications for scientific purposes at DESY. A possible next step would be to split dCache even further. Doors and pools could be containerised within their own containers. This would allow for better management of dCache.

Other potential use cases worth exploring are “dockerizing” different versions of the same application and other services to improve restart time after hardware fixes. Different versions of an application on the same host may be incompatible. Containerisation of different versions may resolve any conflicts the versions may have. Currently restarting some services used in the IT department at DESY take a long time if they have to be shut down. A reason for restarting a service could be if a hardware fix is necessary. “Dockerizing” a service may provide a way of reducing the restart time of these services. Docker containers can be paused. The computer can then be shut down and the hardware fixes completed. It may then be possible for the container to be resumed from the state it was in when it was paused. This has to be tested to see if it actually works.

In summary there are advantages and disadvantages of docker container based virtualisation, which were briefly outlined in the introduction. Another potential downside of using docker is possible security risks due Docker using the host operating systems kernel [18].

To conclude Docker was successfully used to containerise dCache. It appears it may be appropriate for some services and applications at DESY. However the potential use cases above have to be tested and possible security issues that “dockerizing” applications might introduce needs to be investigated.

5 Acknowledgements

A special thanks to my supervisor Karsten for being incredibly generous with his time to both explain and create the project. In addition many others offered up their time and patience including Christian, Olufemi, Tigran, Paul and Patrick.

References

[1] https://www.dcache.org/manuals/dcache-whitepaper-light.pdf

[2] https://en.wikipedia.org/wiki/Docker_(software)

[3] http://www.losangelescomputerhelp.com/2010/04/the-different-types-of-virtualization/

[4] http://searchservervirtualization.techtarget.com/feature/A-brief-history-of-Docker-Containers-overnight-success

[5] http://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/]

[6] https://docs.docker.com/introduction/understanding-docker/

[7] https://docs.docker.com/reference/builder/

[8] https://github.com/wsargent/docker-cheat-sheet#links

[9] https://docs.docker.com/userguide/dockerlinks/

[10] https://github.com/wsargent/docker-cheat-sheet#volumes

[11] https://docs.docker.com/userguide/dockervolumes/

[12] https://en.wikipedia.org/wiki/Git_(software)

[13] https://docs.docker.com/reference/builder/#run

[14] https://en.wikipedia.org/wiki/Dpkg

[15] https://docs.docker.com/reference/builder/

[16] https://docs.docker.com/compose/

[17] https://docs.docker.com/compose/yml/

[18] https://zeltser.com/security-risks-and-benefits-of-docker-application/