Towards a Virtualization Testbed for Energy Usage Reduction on a Raspberry Pi 2 Cluster

Rob J. van Emous
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
r.j.vanemous@student.utwente.nl

ABSTRACT

Nowadays, companies move their software, infrastructure and data from private dedicated servers, to arrays of lightweight, single-purpose *Docker* containers. Although this allows for much quicker re-spawning of failed services and better scaling according to demand, server consolidation remains to be difficult. This results in idly running servers due to over-provisioning and therefore unnecessarily high energy usage in private and public data centers. Reducing the energy usage of data centers is a popular research topic, but modeling and implementing new techniques remains to be either inaccurate or difficult respectively. During this research, virtualization on a *Raspberry Pi 2* (RPi 2) cluster in a micro data center is proposed as a testing platform for energy usage reduction to bridge the gap between computer models and actual data centers. The requirements to the design of the *Container Hosting Environment* (CHE) were determined. Then, different designs for cluster software were identified and evaluated based on their ability to set up the cluster in the best way to be used as a virtualization testbed. Additionally, as a proof of concept, the best design was set up the RPi 2 cluster and showed reasonable performance when tested using a containerized *NodeJS* web app.

Keywords

RPi 2 cluster, energy usage reduction, testbed, virtualization, Docker, cloud servers

1. INTRODUCTION

Up until recently, most companies needed vast ICT infrastructures and the accompanying management departments. Hosting their own website for instance was done though having an array of *physical machines* (PMs) dedicated to handling a specific part of the job: a web server, database, caching and firewall. As described in [39], these servers can be called pets: they require a lot of manual attention to setup and "When they get ill, you nurse them back to health". This is shown in the left part of Figure 1. When more computing power was required, the server hardware was upgraded (vertical scaling), resulting in service downtime.
The next step involves hypervized virtualization. Instead of using dedicated servers, applications are deployed on *virtual machines* (VMs). Each VM has its own *operating system* (OS) and is responsible for a specific task [41]. These VMs can be called cattle according to [39]: when a virtual component of the service misbehaves, it can be shut down while an exact copy of the VM is started in the meantime. Every PM would now be allowed to have identical hardware as shown in the right part of Figure 1. Next to this, scaling could now also be done by increasing the number of servers in the cluster (horizontal scaling). This results in little to no downtime. A cluster is a number of PMs, also called *nodes*, working together as one computing unit.

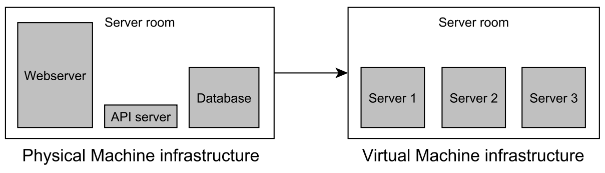

Figure 1: A server room: simplified change in hardware requirements from a PM to VM infrastructure. The height of the gray boxes resembles their computing power.

Although using VMs proves to be more flexible and scalable than using dedicated servers, a different virtualization solution in the form of containerization is getting more and more attention [40]. One popular software solution implementing this technique is *Docker*. As shown in Figure 2, containers do not require a guest OS like VMs do and therefore use much less storage.

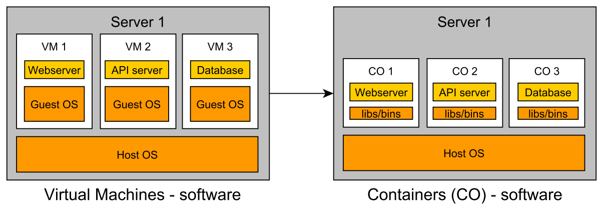

Figure 2: A server: simplified change in storage requiremtn from a VM to container infrastructure.

At the same time, an increasing number of companies move hosting and managing their software infrastructure to the cloud [8]. Instead of buying new servers themselves when needing more processing power, they just rent more and more powerful virtual environments from companies like Amazon (Web Services S3) [1], Google (Compute Engine) [16] or Microsoft (Azure) [30]. Although this has drawbacks in the area of availability and privacy compared to a private server, it is much more scalable, robust and has the insurance of meeting the performance agreed on [28]. The companies behind these cloud based services will provide a client with an environment tailored to its needs. In 2011, those data centers made up about 1.3% of the world's total electricity usage [36]. A large percentage of this is spent on servers running idle, because of underusage of servers due to the need for over-provisioning. Next to this, managing the distribution of VMs or containers over the PMs proves to be difficult. A number of solutions to managing VMs in the most optimal way concerning energy usage have already been researched, and this topic receives much attention nowadays. Developing, testing and applying new optimization techniques are difficult due to the requirement of a real data center for testing.

Next to this, as shown in [38], a RPi 2 cluster has proven to be similar to a typical data center. It lacks raw processing and I/O power, but has advantages in its pricing and energy usage. Therefore, the cluster seems to be able to bridge the gap between the model and reality. This makes the cluster useful as a cheap alternative to mimic the functionality of the data center in handling containers. Other research focuses on the feasibility of replacing existing data center hardware with the RPi 2 cluster, or compares its general energy usage and performance to that of a model [34]. The former proves to be not feasible, and the latter only tests the PMs of the cluster rather than VMs or containers running on the PMs. Therefore, trying to replace cloud services with a mini data center of RPi 2 clusters, does not seem to be useful. By combining this with the difficulties in testing new optimization techniques mentioned in the paragraph above, the use of the cluster becomes clear. A *Container Hosting Environment* (CHE) will be set up on the cluster for testing energy usage reduction strategies. Using a RPi 2 cluster for this purpose is an interesting research field which to our best effort seems to be unexplored.

The main research question that will be addressed in this paper can be formulated like: *How can virtualization on a RPi 2 cluster in a micro data center be set up as testing platform for energy reduction?*
I order to answer the main research question, the research is divided into the following sub questions:

I What are the requirements to the CHE to function effectively as a virtualization testbed?

II What designs for setting up a CHE on a RPi 2 cluster can be used to satisfy these requirements?

III What is a suitable proof of concept that demonstrates virtualization on the RPi 2 cluster?

First, in Section 4, the requirements to the testbed are described. Second, in Section 5, the most popular design options for the software required for setting up a CHE are shown and evaluated based on the proposed requirements. Last, the best design is set up on the RPi 2 cluster as a proof of concept in Section 6.

2. RELATED WORK

The use and qualities of the RPi 2 cluster have already been tested on particular parts of a cloud service. For instance, hosting a data center or video streaming service [38], and cloud based parallel computing (RSA) [43]. Next to this, its performance and energy usage have been tested and compared to a model [34].

Their research is mainly focused on replacing a certain part of the functionality of a data center with a RPi 2 cluster. It shows primarily that this relatively cheap cluster uses little energy, but lacks in performance compared to real data centers. In contrast, this research will focus on setting up a flexible container environment on the RPi 2 cluster, capable of hosting all services which were individually researched before. At the same time, instead of trying to replace current data centers, it proposes this cluster as testbed for energy usage reduction strategies.

In [2], the possibilities of running *Docker* containers on an individual RPi 2 has been researched and [27] even researched and built a software solution for easily running *Docker* with *Swarm* on the RPi 2. Next to this, some manuals have already been written about using Mesos [5], Kubernetes [35], or *Swarm* [19] with *Docker* on a RPi 2 cluster. The first paper focuses on first getting the software running on an individual RPi. Then, it compares the energy usage of applications running in this environment, with running them directly on the operating system. This research extends that by first discussing and designing the best virtualization environment, after which this design will be run on the RPi 2 cluster. Next to this, the focus of this research is not on the efficiency of virtualization, but on setting up a testbed for making virtualization even more efficient in order to reduce the energy usage of data centers. The second paper focuses on providing an easy private cloud setup for businesses and institutions in the form of an RPi 2 cluster. This research builds on that paper, but uses it to setup a testbed instead of a private cloud. The manuals focus on getting a certain CHE set up, but do not compare or mention other designs.

3. BACKGROUND

In this section, two main concepts of virtualization are worked out. First, an overview of virtualization is given by showing the available types of virtualization and discussing several software packages implementing this technique in Section 3.1. Second, the application of virtualization in the real world, like in private servers and cloud services is shown in Section 3.4.

3.1 Virtualization overview

Virtualization is any way in which a piece of software, like a browser, database, web server or even an entire OS, does not run on respectively the software or hardware it expects to run on. An OS expects to run directly on hardware, but hypervized virtualization allows for it to run as a guest OS on top of the host OS. In that same way, software expects to run on an OS directly, but by using container-based virtualization, it can be run inside the isolated environment of a container. Both techniques will be explained in more detail below. Finally, an implementation of containerization called *Docker* and software to run containers on a cluster of PMs called *Kubernetes* are described.

3.1.1 Hypervized virtualization

In hypervized virtualization like *Xen*, *KVM*, *Virtual box* or *VMware*, this environment contains the guest OS which assumes it is running on native hardware, while it is actually running on the host OS. These VMs have the advantage of hosting a completely isolated environment, which is preferable for security- or privacy-sensitive applications. Disadvantages are their large size of up to tens of Gigabytes and the fact that they take up to a minute to start, depending on the guest OS and hardware performance.

3.1.2 *Container-based virtualization*

In container-based virtualization like *Docker*, *LXD*, or *Rocket*, the environment only needs to contain the specific libraries it dependents on. The applications running inside these containers are still isolated from each other, but they no longer require their own guest OS or an emulator which degrades the performance. It shares the OS with the other containers and the host. For this reason, containers are only several Megabytes in size and are able to start up in milliseconds similar to a normal process [14]. Next to this, according to [15], the main advantages of container-based virtualization over hypervized virtualization, are performance improvements and reduced startup time. Containerization is especially useful for applications which can be split in several logical components like a database, web server and load balancer. Containers are less usable when legacy or larger (inseparable) systems need to be run in virtualized environments, or when complete isolation between the containers is required for privacy or security reasons. In [10, 17], it is made clear that isolation of the hardware resources, file system and access rights to containers have been implemented. Still, various relatively simple attacks to gain access to the contents of a container exist.

A simplified comparison of structure, CPU performance and storage usage of PMs, VMs and containers is shown in Figure 3. All servers are assumed to be equal in total CPU performance and the amount of storage. Also, the OS uses (nearly) no CPU resources and uses 20% disk space. As can be observed, we need four dedicated servers which are all heavily under-used. In contrast, we need only two VM hosting servers which are about maxed out in usage. Finally, we need an equal amount of *Docker* container servers, but those require a lot less storage than the second option, due to the lack of guest OSs.

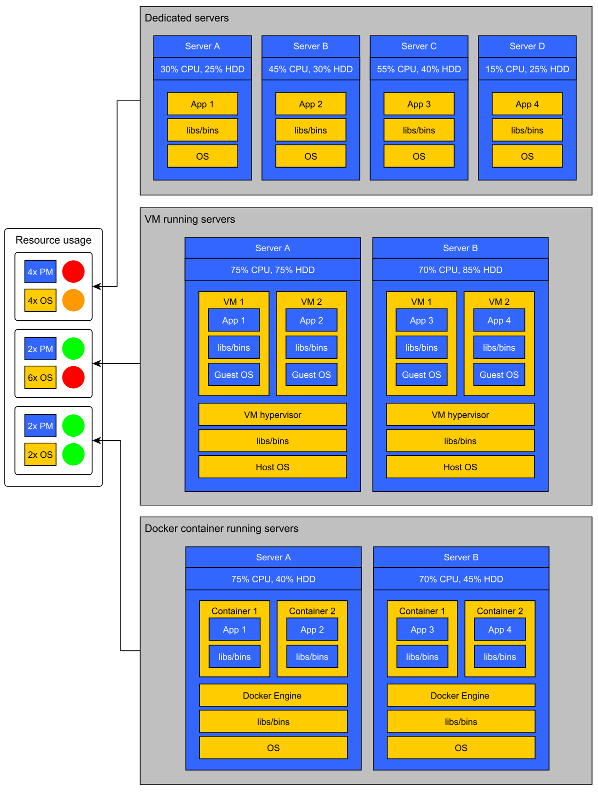

Figure 3: Structure, CPU performance and storage usage of PMs, VMs and containers.

3.2 Docker

Docker [14] is an abstraction layer used easily spawn lightweight containers. It is an extension to *Linux Containers* (LXC) [25], which has less features and is much more difficult to use. With *Docker*, users can build, ship and run containers quickly and from any location. All setup and data of the container is packed in an image. This is possible because of the *UnionFS* file system service *Docker* is based on [12]. It only needs one base image of a file system and will only save the changes to this in its own image. At run-time, it will merge the changes and image of a file system into the virtual file system available to the container. These images can be pushed to, and pulled from *DockerHub* [13], which is a *Docker* image host and version control system.

3.3 Kubernetes

Kubernetes [23] is a *Container Orchestrator* (CO) designed by *Google* and later donated to the *Cloud Native Computing Foundation* [42]. It is developed for distributing and running *Docker* containers on a cluster of physical machines. It takes care of starting the containers, duplicating them on multiple PMs, keeping them running or restarting them when necessary [23]. Kubernetes is also able to move containers around between PMs, but only in a static way. Thus, the program running in a container would first have to be shut down before it can be moved. This causes the service this container hosts, to be temporarily unavailable to the end-user. An extension to the software called *Ubernetes* even goes as far as managing multiple *Kubernetes* clusters with completely different hardware and possibly located around the world.
As an example of virtualization on a RPi 2 cluster, the general structure of the cluster running both **Docker** and **Kubernetes** is shown in Figure 4. It is a cluster made of 3 RPi 2s of which the one on the left is the master node and the ones on the right are slave nodes. The master can be controlled via its *REST API* and also hosts the shared data of the cluster. Both slaves are running four containerized applications spread over 2 pods: a container-holder accessible from the network.

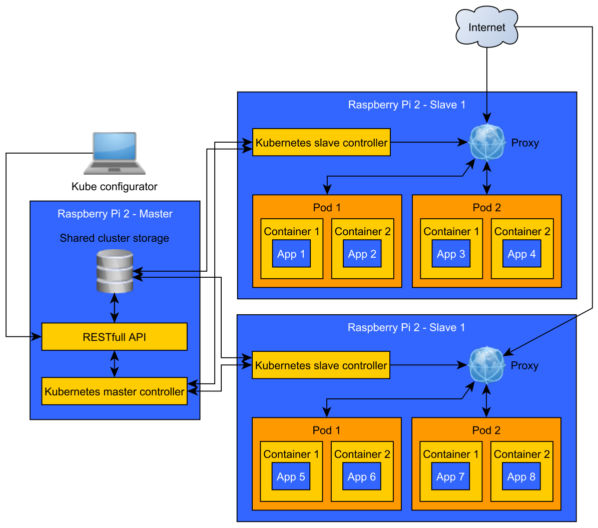

Figure 4: The simplified structure of the Raspberry Pi 2 cluster running Kubernetes and Docker. Inspired by [37].

3.4 Application of virtualization

Virtualization is used in several contexts ranging from home computers and company servers (private servers), to even data centers (cloud computing). These options are explained in more detail below.

3.4.1 Private servers

Private servers are PMs or VMs which are privately administrated, and deployed for a small target group and particular purpose, like hosting your own website or a game server. Next to this, companies can use private servers to host their own infrastructure and software for their employees or provide services to their customers. As already shown in the introduction, without using virtualization, every server will perform a specific task resulting in a lot of largely under-used servers constantly turned on to cope with sudden increases in traffic. Virtualization is able to increase server consolidation by allowing for some PMs to be fully used while others can be turned off. This is accomplished by running all services in VMs or containers in as few PMs as possible. Sudden increases in traffic could then be dealt with by turning on an extra PM and allowing for the VMs or containers to scale accordingly.

3.4.2 Cloud computing

The cloud no longer only encompasses the place for users to store their data available from any network location like *Microsoft 365*, *Google Drive* and *Dropbox*. It has evolved into a much bigger structure. In [18], cloud computing is described as: "*To outsource IT activities to one or more third parties that have rich pools of resources to meet organization needs easily and efficiently*". These activities can be separated into the types which are explained in [24]. Running corporate software online via subscriptions: *Software as a Service* (SaaS), moving most of the computing power and infrastructure away from companies: *Infrastructure as a Service* (IaaS), and finally building and hosting constantly changing software services like websites: *Platform as a service* (PaaS).

4. DESIGN REQUIREMENTS

This section describes the requirements of the software components that will be used to set-up the CHE on the RPi 2 cluster. To maximize the usability of the cluster as a test-bed, most requirements are based on requirements to actual data centers. In [4, p. 6] five criteria were stated of which scalability, performance and manageability of the cluster as a whole were already assessed in [38]. Especially, the performance of the cluster is lacking compared to actual data centers. Still, it allows for a software set-up able to mimic that of an actual data center. The other criteria stated below are based on practical usability of the cluster as a testbed for energy usage reduction in virtualization. The requirements are described in the next four Sections.

4.1 Compatibility

The usability of the software components is determined based on its compatibility with the hardware of the RPi 2 cluster, as well its mutual compatibility. Not all software is meant to be used alongside each other, while others were made with the combined usage in mind. The reason why the components should be as compatible as possible, is grounded in the fact that this makes it easier to set up the software, and it most likely is already done before. Because of this, most issues that would occur with combining the two software components have already occurred to others and solutions might have been found already.

4.2 Resource utilization & Scalability

The resource utilization of the CHE is evaluated while taking into account its scalability. Resource utilization encompasses a lot of different parts of the system, like the usage of the CPU, RAM, storage, network and number of PMs. It should be as low as possible when the system runs idle, but also be able to use the resources to their maximum potential when required. The requirement can be made concrete by estimating or measuring the utilization of these hardware components when running no containers, running a few containers and when running a large amount of containers.

4.3 Configurability & Controllability

The configuration and controllability of the CHE is assessed for the extent to which it is easy, extensible and allows for using the cluster as a testbed. The configuration of the cluster is every action required to get a (multi-) container application up and running. The software is assessed based on the former steps to be relatively easy, and flexible when (small) changes are required. After this, the controllability can be determined. Most components can be controlled from a client program on the master node or an external computer. This allows for starting, stopping and changing containers and the relations and communications between them. The extend to which the software provides this functionality is evaluated.

4.4 Reliability & Replication

Reliability is about the extent to which the software does what it is meant for, for extended periods of time, despite changes to, or failures in the system. This is required for researchers using the testbed, but also to clients of any data center which this testbed models. This requirement can be made measurable by running a service in a container for extended periods of time, by stressing the hardware at the same time, or even by physically changing the hardware during the execution. The latter can be done by removing the network or power cable of a RPi 2 for instance. Replication is the duplication of identical containers. This can be used to distribute the load among different containers to take advantage of multi-core or multi-PM (cluster) systems. It can also be used to cope with failing hardware, by moving a container to another PM in the cluster.

5. DESIGN COMPOSITION

This section will propose a design for setting up the CHE on the RPi 2 cluster. In Section 5.1, the required software will be split in groups based on their type of contribution to the total design. Next to this, a description is given of every group and possible design options (software or hardware) are discussed. In Section 5.2, a number of designs composed of one item from every group, are evaluated based on the extent to which they pass the requirements proposed in Section 4. This ultimately leads to a ranking of the designs, of which the best option is selected to be setup on the RPi 2 cluster in Section 6.

5.1 Design group overview

For setting up a CHE, a number of different hardware and software components are required. These are divided into four groups based on their type of contribution to the CHE. Figure 5 shows these groups and the design options available within every group. The first three groups are essential to virtualization on one PM and the CO group adds functionality to support a cluster of PMs. The list of hardware and software alternatives within every group is not meant to be extensive, although it does contain the most popular options currently available.

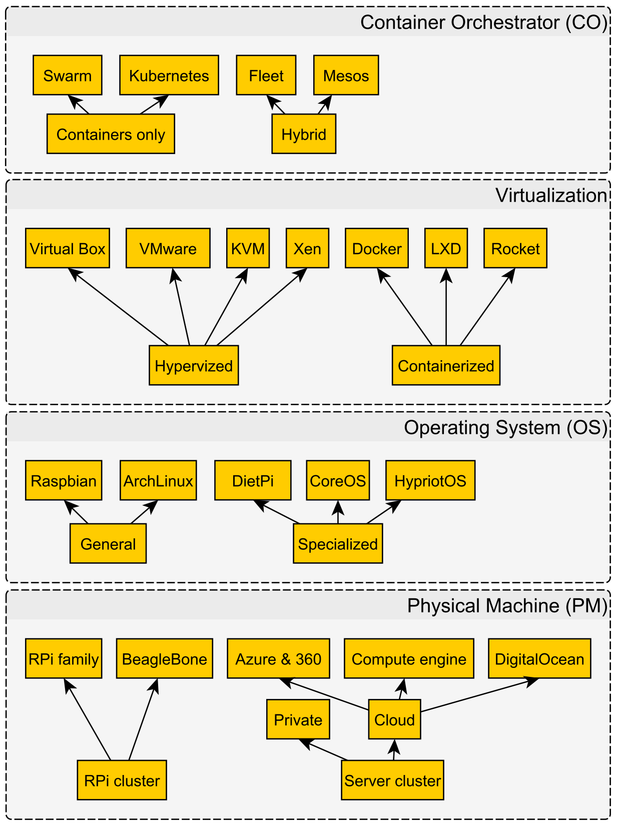

Figure 5: CHE design alternatives grouped by their type of contribution.

5.1.1 *Physical Machine*

A number of options exist for the hardware (PMs) of the testbed. Although this research will be based on a RPi 2 cluster, other options are evaluated as well to show its main advantages. The most important aspect of the cluster hardware is that it should make using the testbed more attractive than an actual data center: low purchase costs, energy efficiency and a large community of users (helpful for sharing experience). The RPi 2 is equal in price and about six times as powerful as the RPi B+, while using only slightly more power [26]. Price to performance and performance to energy usage of RPi B+ is the best among most popular micro-boards like the *BeagleBone* according to [6]. Therefore, these benchmark results can also be applied to the RPi 2. Next to this, as Section 2 shows, a lot of test clusters are based on members of the RPi family: version 1, 2 or 3. Finally, the community behind members of the RPi family is the largest among development boards [22]. A cluster of private servers as described in Section 3.4.1 could be used as a testbed, but is still a large and rather expensive option, compared to micro-boards like the RPi 2.

5.1.2 *Operating System*

The OS running on every RPi 2 in the cluster cannot be any OS, for it has to be ARM compatible and lightweight in CPU, RAM and disk usage. Like already mentioned before, the RPi 2 cluster has decent computing power, but this is still quite lacking compared to traditional servers. For this reason, only light variants of Linux based OSs should be used. *Raspbian* is made to be used with a desktop instead of just SSH and has a very large list of preinstalled packages. *HypriotOS* is an OS based on *Debian* (just as *Raspbian*), but with primarily essential packages installed. *DietPi* even takes this a step further by being made as small as possible. This result in the size of the OS being only about 500 MB and having just 11 running processes [7]. These OSs were all specifically designed for running on a micro-board like the RPi 2. Other options are a general OS called *ArchLinux* and also *CoreOS* which is based on *ChromeOS*. The latter focuses on distributed storage and easy software upgrading [9]. An alternative to that would be Microsoft's *Windows 10 for IoT* (Internet of Things) [29, p. 257-271], but currently does not seem to support container-based virtualization like *Docker*. *HypriotOS* is specifically made for running containers and *Docker Swarm* out-of-the-box, while *CoreOS* and *ArchLinux* can be easily made to work with containers and *Kubernetes*. *Raspbian* and *DietPi* can be made to support both options, but this is a rather tedious process [11].

5.1.3 *Virtualization method*

As shown in Section 3.1, virtualization can be achieved by using hypervized VMs or containers. Both having their own strengths and weaknesses. Regardless of this outcome, using VMs on the RPi 2 proved to be unfeasible. As this manual [33] points out, with some tweaking, it is possible to enable virtualization on the Cortex-A7 SoC of the RPi 2. Unfortunately, sharing the CPU with multiple OSs can only be done in a static way: by assigning one CPU core to the VM. This, of course, does not scale beyond the limited amount of cores of the system. In Figure 6, virtualization in the RPi 2 is compared to that of a server.

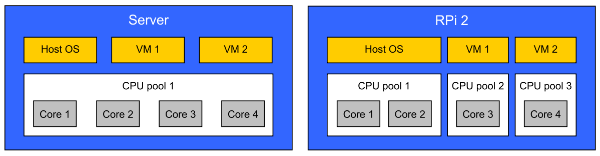

Figure 6: Comparison of hardware virtualization in a server and the RPi 2.

Containerization on the other hand, does not require special hardware to run and can therefore be used on the RPi 2. More advantages of using containers over using VMs is described in Section 3.1. Several options are available for running containers like *Docker*, *LXD* and *Rocket*. *Microsofts* container manager for Windows called *Drawbridge* will not be discussed, because we will only focus on running *Linux* based OSs. The advantages of *Docker* are already described in Section 3.2. It is also the only container software found to be used on the RPi 2 before, as shown in Section 2. *Rocket* provides features similar to *Docker* [44], while *LXD* is more of a hybrid between a VM hypervisor and a container runner [32]. The latter is primarily used for providing an additional layer of security around existing *Docker* containers.

5.1.4 *Container Orchestrator*

Container creation and deployment software like *Docker* confines itself to running containers on only one PM, a *Container Orchestrator* (CO) is the component which manages the containers across a cluster of PMs. It turns multiple PMs into one virtualized platform for running containers. Orchestrators mostly use a master-slave setup: one node (master) configurable through an API, controls all other nodes (slaves). The master can also be used for shared storage between the clients. Although this can also be done by duplicating the storage over all nodes. Most orchestrators are also able to scale applications to demand: spawn more instances to distribute the load, and deal with failing containers or PMs by spawning identical containers (on another node) in the cluster. Popular CO options are *Docker Swarm*, *Kubernetes*, *Mesos* and *Fleet* [31]. *Docker Swarm* is easy to use for small clusters, but not fully stable. It is made by the same company that developed *Docker*. *Kubernetes* is already described in Section 3.3. *Fleet* and *Mesos* are meant for running very large clusters. They are more of a framework for running VMs, containers and COs, than just a CO. Therefore, they are needlessly complex for deploying on a small testbed.

5.2 Design evaluation and proposal

All design groups and possible options have been described. For the first group, the RPi 2 (or its successor 3) currently seem to be the best testbed hardware to use. The second group could be filled in using either *HypriotOS*, *CoreOS* or *ArchLinux*. For the third group, *Docker* is the best candidate, and the last group is best represented by *Docker Swarm* or *Kubernetes*.
For the OS and CO groups, a number of options for designing the RPi 2 cluster are left over. In Table 1, these design options are evaluated based on the extent to which they pass the requirements proposed in Section 4.

Table 1: Design options requirement fulfillment

	Requirements			
	4.1	**4.2**	**4.3**	**4.4**
CoreOS + Kubernetes	++	+	+/-	+
ArchLinux + Kubernetes	+	+	+/-	+
HypriotOS + Swarm	++	++	+	+/-

4.1: The first and last OSs are build with the accompanying CO in mind and therefore are the most compatible. 4.2: The first two designs, although equally scalable as the last design, use more resources. 4.3: For using the CO in a small testbed cluster, *Kubernetes* has a unnecessarily complex structure and control-interface compared to *Swarm*. 4.4: *Kubernetes* is a more established and stable CO than *Swarm* resulting in easier replication and more reliability. Although all three designs could be used to setup the RPi 2 cluster, based on these requirements, the design consisting of *HypriotOS* and *Docker Swarm* seems to be the best alternative.

6. PROOF OF CONCEPT

The CHE is set up in Section 6.1 using the best design determined in Section 5. After this, a test application is run to demonstrate the abilities of virtualization on the RPi cluster in Section 6.2. A step-by-step set up of the system and the benchmarks is described in Appendix A.

6.1 Setup of design and NodeJS test

The RPi 2 cluster is setup using the design: *HypriotOS*, *Docker* and *Swarm*. One RPi 2 is configured to be the *Swarm* master and the other three nodes as slaves (or workers). In order to show the abilities and advantages of the CHE, a small eight-case test is performed on the cluster hosting a *SaaS* (see Section 3.4.2) web app in *NodeJS*. *NodeJS* applications normally only use one core of a PM. By scaling the number of instances distributed over the cluster, it is likely to have a significantly better performance. Figure 7 shows the test setup running the web app in 1 container (yellow), 3 containers (orange), or 7 containers (red). The containers are managed by 1 node (left part), or distributed over 4 nodes (right part). A 15-container setup is also tested, but not shown in the Figure for the sake of brevity. In this case, with the 4 node setup, every node runs two additional containers. In the cases where more than one container is deployed, a separate container running a load balancer is added.

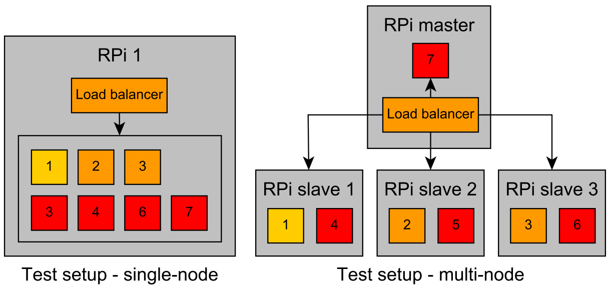

Figure 7: The single-node and multi-node test setup.

6.2 Testing the design

The setup is benchmarked using software from the *Apache Toolset* [3] which performs 10.000 requests to the web app with a concurrency of 50. The total performance of the system is determined based on the average number of successful requests per second. The results are shown in Table 2 and visualized in Figure 8. The full benchmark results can be found at the *GitHub* repository referred to in Appendix A.

Table 2: Apache Toolset benchmark results

		# of containers / web apps			
		1	**3**	**7**	**15**
# of nodes	**1**	72.03	159.7	133.3	106.6
	4	75.75	192.4	373.9	416.8

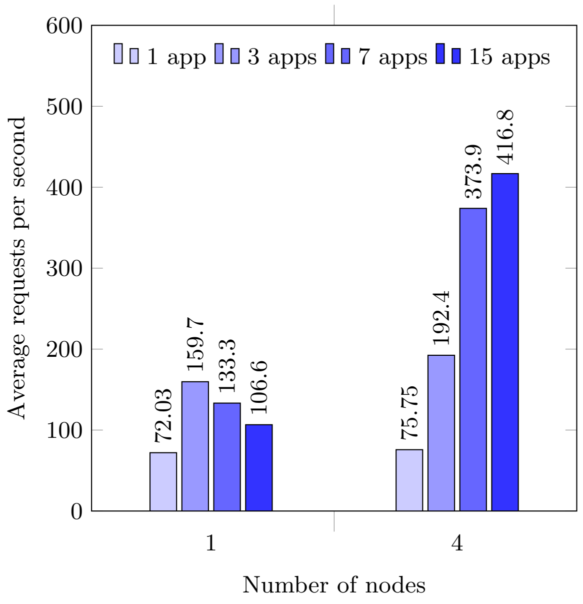

Figure 8: Apache Toolset benchmark results visualized

The results show that in the single-node test, scaling the web app is only beneficial up the number of cores of the system: 4 in the case of one RPi 2. By deploying even more containers on the node, the overhead of running containers starts to outweigh the scaling benefits and the performance drops. In the multi-node test, the performance is much higher than in the first test. It also increases somewhat linearly to the scaling. This increase also levels out, when the number of containers reaches the total amount of CPU cores in the cluster (which is 16). The reason for the performance increase being less than expected when scaling, could be either be due using containerized virtualization, the overhead in the load balancer, the network as a whole, or something else. This is not investigated further.

7. CONCLUSION

A list of requirements to the RPi 2 cluster is constructed based on prevailing requirements to other data centers or clusters. These are: compatibility, resource utilization & scalability, configurability & controllability and reliability & replication. Possible software designs are shown and evaluated based on the requirements. The design consisting of a RPi 2 cluster running *HypriotOS*, *Docker* and *Swarm*, currently seems to be the best option. The proof of concept shows that the chosen design can be set up on the RPi 2 cluster quickly and easily. The containerized *NodeJS* application run on it greatly benefited from scaling, but the advantage leveled out when the number of containers reached the number of CPU cores in the cluster.

The research question of this paper was formulated like: *To what extent can virtualization on a cluster of RPi 2s in a micro data center be set up as testing platform for energy usage reduction?* Using a design consisting of lightweight components, a virtualization cluster has successfully been setup. It showed to be easily configurable and performed well in the small test setup. Therefore, virtualization on the RPi 2 cluster in a micro data center can indeed be used as testing platform for energy usage reduction.

Future Work

This research can be extended by making a link to a computer model, as well as a real cloud host. For the cluster to be able to accurately serve as testing platform, the characteristics like energy usage and performance of the CHE should be compared to both ends.

In the first case, the characteristics of the CHE could be compared to an extension of the model described in [34], of a cluster of container hosting machines. Possibly, there are ways in which the workings of the test-bed are not quite reflected in the model and vica versa. The cluster provides data and answers, but those might be incorrect due to wrong assumptions. The model could therefore be used to verify or justify this data and suggest improvements when possible.

In the second case, the performance and energy usage of the CHE when running energy reduction strategies, could be compared to a real cloud host running comparable applications. There are large differences in cluster size and node hardware between them which could result in unexpected behavior. Most data centers for instance have x86/64 based CPU's, while the RPi 2 has an ARM based CPU. Furthermore, the CPU of the RPi 2 does not support hardware virtualization out-of-the box. This does not impose any problems to containerized VMs, but hypervized VMs will run significantly slower without hardware acceleration.

Acknowledgements

This research would not have been possible without the help and encouragement of Björn Postema from the Design and Analysis of Communication Systems group of the University of Twente. Next to this, the author would like to thank G.G.S. van Emous for his corrections and advice regarding the paper. Last, the author is thankful for the power and rest he believes God provided to him during the research. *"For I can do everything through Christ, who gives me strength."* (Phil. 4:13 - The Holy Bible (NLT)).

8. REFERENCES

[1] Amazon. Broad & Deep Core Cloud Infrastructure Services. `https://aws.amazon.com/`, 2016. [Last accessed: May 16, 2016].

[2] G. Anusooya and V. Vijayakumar. *Power Optimization System for a Small PC Using Docker*, pages 173–181. Springer International Publishing, Cham, 2016.

[3] Apache. ab - Apache HTTP server benchmarking tool. `https://httpd.apache.org/docs/current/programs/ab.html`, 2016. [Last accessed: June 19, 2016].

[4] M. Arregoces and M. Portolani. *Data center fundamentals.* Cisco Press, 2003.

[5] M. Asay. Let Docker Swarm all over your Raspberry Pi Cluster. `http://www.techrepublic.com/article/docker-and-mesos-like-peanut-butter-and-jelly/`, 2015. [Last accessed: May 16, 2016].

[6] M. F. Cloutier, C. Paradis, and V. M. Weaver. Design and analysis of a 32-bit embedded high-performance cluster optimized for energy and performance. In *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*, pages 1–8, Nov 2014.

[7] Cnxsoft. DietPi is Lightweight, Easy to Use Debian Based Distribution for Raspberry Pi, ODROID, and Orange Pi Boards. `http://www.cnx-software.com/2015/12/18/dietpi-is-lightweight-easy-to-use-debian-based-distribution-for-raspberry-pi-odroid-and-orange-pi-boards/`, 2016. [Last accessed: June 22, 2015].

[8] L. Columbus. Cloud Computing Adoption Continues Accelerating In The Enterprise. `http://www.forbes.com/sites/louiscolumbus/2014/11/22/cloud-computing-adoption-continues-accelerating-in-the-enterprise/#59c4f6325feb`, 2014. [Last accessed: May 12, 2016].

[9] CoreOS. Using CoreOS. `https://coreos.com/using-coreos/`, 2016. [Last accessed: June 22, 2016].

[10] Docker. Containers & Docker: How Secure Are They? `https://blog.docker.com/2013/08/containers-docker-how-secure-are-they/`, 2013. [Last accessed: June 15, 2016].

[11] Docker. Debian. `https://docs.docker.com/engine/installation/linux/debian/`, 2016. [Last accessed: June 19, 2016].

[12] Docker. Understand the architecture. `https://docs.docker.com/engine/understanding-docker`, 2016. [Last accessed: May 16, 2016].

[13] Docker. Welcome to Docker Hub. `https://docs.docker.com/docker-hub/`, 2016. [Last accessed: June 16, 2016].

[14] Docker. What is Docker? https://www.docker.com/what-docker, 2016. [Last accessed: May 4, 2016].

[15] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 610–614. IEEE, 2014.

[16] Google. Compute Engine - Scalable, High-Performance Virtual Machines. https://cloud.google.com/compute/, 2016. [Last accessed: May 16, 2016].

[17] U. Gupta. Comparison between security majors in virtual machine and linux containers. *arXiv preprint arXiv:1507.07816*, 2015.

[18] Q. Hassan. Demystifying cloud computing. *The Journal of Defense Software Engineering*, pages 16–21, 2011.

[19] Hypricot. Let Docker Swarm all over your Raspberry Pi Cluster. http://blog.hypriot.com/post/let-docker-swarm-all-over-your-raspberry-pi-cluster/, 2015. [Last accessed: May 16, 2016].

[20] Hypriot. How to use Docker Compose to run complex multi container apps on your Raspberry Pi. http://blog.hypriot.com/post/docker-compose-nodejs-haproxy/, 2015. [Last accessed: June 18, 2016].

[21] Hypriot. How to setup a Docker Swarm cluster with Raspberry Pi's. http://blog.hypriot.com/post/how-to-setup-rpi-docker-swarm/, 2016. [Last accessed: June 18, 2016].

[22] Iqjar. An overview and comparison of today's single-board micro computers. http://iqjar.com/jar/an-overview-and-comparison-of-todays-single-board-micro-computers/, 2013. [Last accessed: June 18, 2016].

[23] Kubernetes. Accelerate Your Delivery. http://kubernetes.io/, 2016. [Last accessed: May 4, 2016].

[24] A. Lenk, M. Klems, J. Nimis, S. Tai, and T. Sandholm. What's inside the cloud? an architectural map of the cloud landscape. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 23–31. IEEE Computer Society, 2009.

[25] Linux Containers. Infrastructure for container projects. https://linuxcontainers.org/, 2016. [Last accessed: June 16, 2016].

[26] MagPi. Raspberry Pi 3 specs and benchmarks. https://www.raspberrypi.org/magpi/raspberry-pi-3-specs-benchmarks/, 2016. [Last accessed: June 18, 2016].

[27] M. R. Marcel Großmann, Andreas Eiermann. Hypriot cluster lab: An arm-powered cloud solution utilizing docker, May 2015.

[28] C. McCoy. What to Consider Before Moving Your Business to the Cloud. http://www.pcworld.com/article/2045740/what-to-consider-before-moving-your-business-to-the-cloud.html, 2013. [Last accessed: May 13, 2016].

[29] P. Membrey and D. Hows. *Learn Raspberry Pi 2 with Linux and Windows 10*. Apress, 2015.

[30] Microsoft. The intelligent cloud for smarter business. https://azure.microsoft.com/en-us/?b=16.17, 2016. [Last accessed: May 16, 2016].

[31] A. Mouat. Swarm v. Fleet v. Kubernetes v. Mesos. https://www.oreilly.com/ideas/swarm-v-fleet-v-kubernetes-v-mesos, 2015. [Last accessed: June 22, 2016].

[32] M. Nestor. Infographic: LXD Machine Containers from Ubuntu Linux. http://linux.softpedia.com/blog/infographic-lxd-machine-containers-from-ubuntu-linux-492602.shtml, 2015. [Last accessed: June 22, 2016].

[33] S. L. Pascual. Enabling KVM virtualization for Raspberry Pi 2. http://blog.flexvdi.com/2015/03/17/enabling-kvm-virtualization-on-the-raspberry-pi-2/, 2015. [Last accessed: June 13, 2016].

[34] B. F. Postema and B. R. Haverkort. *Computer Performance Engineering: 12th European Workshop, EPEW 2015, Madrid, Spain, August 31 - September 1, 2015, Proceedings*, chapter An AnyLogic Simulation Model for Power and Performance Analysis of Data Centres, pages 258–272. Springer International Publishing, Cham, 2015.

[35] Raspberrypicloud. How to: Kubernetes Multi-node on Raspberry Pi 2s. https://raspberrypicloud.wordpress.com/2015/08/11/how-to-kubernetes-multi-node-on-raspberry-pi-2s/, 2015. [Last accessed: May 16, 2016].

[36] T. Renzenbrink. Data Centers Use 1.3% of World's Total Electricity. A Decline in growth. https://www.elektormagazine.com/articles/data-centers-use-1-3-of-worlds-total-electricity-a-decline-in-growth, 2011. [Last accessed: May 13, 2016].

[37] C. Sanchez. Scaling Docker with Kubernetes. http://www.infoq.com/articles/scaling-docker-with-kubernetes, 2014. [Last accessed: May 16, 2016].

[38] N. J. Schot, P. J. E. Velthuis, and B. F. Postema. *Capabilities of Raspberry Pi 2 for Big Data and Video Streaming Applications in Data Centres*, pages 183–198. Springer International Publishing, Cham, 2016.

[39] N. Slater. Pets vs. Cattle. https://blog.engineyard.com/2014/pets-vs-cattle, 2014. [Last accessed: May 11, 2016].

[40] D. Strauss. Containers-Not Virtual Machines-Are the Future Cloud. http://www.linuxjournal.com/content/containers%E2%80%94not-virtual-machines%E2%80%94are-future-cloud, 2013. [Last accessed: May 12, 2016].

[41] Techopedia. Virtual Machine (VM). https://www.techopedia.com/definition/4805/virtual-machine-vm. [Last accessed: May 13, 2016].

[42] The Linux Foundation. Cloud Native Computing Foundation Charter. https://cncf.io/about/charter, 2016. [Last accessed: July 23, 2016].

[43] M. van der Vegt. Raspberry pi 2 as an feasible alternative for cloud based parallel computing solutions. In *24th Twente Student Conference on IT, 2016*, 2016.

[44] K. Ward. Rocket Containers, an Alternative to Docker, Blasts Off. https://virtualizationreview.com/articles/2014/12/02/rocket-containers-announced.aspx, 2014. [Last accessed: June 22, 2016].

Appendix A

This section contains a practical guide in setting up the CHE and testing it according to the proof of concept proposed in Section 6. All files referred to in this section can be found at a *GitHub* repository: https://github.com/RobvEmous/DACS-BachelorReferaat. Commands should either be entered in a (Ubuntu 16.04) Terminal indicated by the **ubu>** prefix, or in a RPi 2 accessed via SSH indicated by the **rpi>** prefix. All used IP-addresses are placed in square brackets and should be replaced by your own.

Setting up HypriotOS v0.8 on the RPi 2

The guide is based on on [19], but is extended to support key-based root login via SSH. This setup is meant for one RPi 2 which we will call **rpi2** with static IP-address **192.168.100.102**. The procedure should be repeated (with an updated IP-address) for all nodes in the cluster.

Install HypriotOS

- Insert SD card into computer.

```
ubu> sudo apt-get -y install unzip curl pv hdparm
ubu> wget https://raw.githubusercontent.com/
hypriot/flash/master/$(uname -s)/flash
ubu> chmod +x flash
ubu> sudo mv flash /usr/local/bin/flash
ubu> sudo flash --hostname rpi2 http:downloads.
hypriot.com/hypriotos-rpi-v0.8.0.img.zip
```

- Insert SD card into the RPi 2 and wait about 2 min.

Key based root login

```
ubu> ssh pirate@[192.168.100.102]
```

- Type: `hypriot`

```
rpi> sudo su
rpi> passwd
```

- Type a new password

```
rpi> sudo nano /etc/ssh/sshd_config
```

- Change: `PermitRootLogin without-password` to `PermitRootLogin yes`

```
rpi> sudo service ssh restart
rpi> exit (2x)
ubu> ssh-copy-id -f -oStrictHostKeyChecking=no
-oCheckHostIP=no root@[192.168.100.102]
```

- Type same password

Test setup

```
ubu> ssh root@[192.168.100.102]
```

Setting up Docker Swarm on the cluster

The guide is based on on [21] and will setup *Docker Swarm* on the cluster using **rpi2** as master and the **rpi3** with static IP-address **192.168.100.103** as slave. Repeat the slave setup (with an updated IP-address) for all other nodes in the cluster. We first need *Docker Machine* to easily configure the masters and slaves from the computer.

Install Docker Machine

```
ubu> sudo apt-get install docker-machine
```

Generate Swarm token

```
ubu> export TOKEN=$(for i in $(seq 1 32); do echo
-n $(echo "obase=16; $(($RANDOM % 16))" | bc);
done; echo)
```

Create master node

```
ubu> docker-machine create -d generic \
--engine-storage-driver=overlay --swarm
--swarm-master \
--swarm-image hypriot/rpi-swarm:latest \
--swarm-discovery="token://$TOKEN" \
--generic-ip-address=[192.168.100.102] \
rpi2
```

Create slave node

```
ubu> docker-machine create -d generic \
--engine-storage-driver=overlay --swarm
--swarm-image hypriot/rpi-swarm:latest \
--swarm-discovery="token://$TOKEN" \
--generic-ip-address=[192.168.100.103] \
rpi3
```

Test setup

```
ubu> docker-machine ls
```

Creating and testing a NodeJS app

The guide is based on on [20], but is extended to a cluster of RPi 2s. First, the necessary files and *Docker Compose* are downloaded. *Docker Compose* is used for starting the multi-container application. This test requires the amount of *NodeJS* containers indicated by '[nr]', to be specified: 3, 7 or 15.
Setting up the test on the whole cluster of RPis could have been done automatically by using *Docker Compose* and *Docker Machine*, but would also require a service like *ZooKeeper* for creating a network overlay. This would allow for the containers on different nodes to communicate. Instead, containers were deployed statically on every RPi 2 in the cluster. A manual on setting this up can be found at the *GitHub* repository referred to earlier.

Install Docker Compose & Download files

```
rpi> sudo sh -c "curl -L https://github.com/
hypriot/compose/releases/download/1.1.0-raspbian/
docker-compose-‘uname -s‘-‘uname -m‘ > /usr/
local/bin/docker-compose; chmod +x /usr/local/
bin/docker-compose"
rpi> docker pull rjvanemous/nodejs-test
```

- Download the *DACS-BachelorReferaat* repository from github.

```
rpi> cd DACS-BachelorReferaat/NodeJStest
```

Starting & stopping - single-node, single-container

```
rpi> docker run -p 80:80 -name web -d
rjvanemous/nodejs-test
rpi> docker stop web
```

Starting & stopping - single-node, multi-container

```
rpi> docker-compose -f docker-compose-[nr].yml
up -d
rpi> docker-compose stop
```

Install & start Apache benchmark tool

```
ubu> sudo apt-get install apache2-utils
ubu> ab -n 10000 -c 50 http://[192.168.100.102]/
```