University of Southern Denmark

Data Center Demand Response Through Workload Migration

Master Thesis - Software Engineering

Nikolaj Wolder Steenberg

niste15@student.sdu.dk

Exam Nr: 474996

University of Southern Denmark

Morten Abrahamsen Olesen

moole15@student.sdu.dk

Exam Nr: 424143

University of Southern Denmark

Supervisor

Jakob Hviid

jah@mmmi.sdu.dk

University of Southern Denmark

Abstract

With the increasing adoption of renewable energy sources, the ever-lasting problem of stabilizing the power grid has to be found, where conventional methods of energy storage can not be applied. One of the large contributors to Danish energy consumption is data centers. These large processing facilities consume a vast amount of power, however, they possess an attribute that might be able to reduce the problem of power grid instability. Each workload found on a data center can potentially be moved, with the potential to balance the power consumption of data centers between each other. Within this thesis, the environment of data centers and demand response systems is explored, in relation to architecture and which workloads one might find within a data center, as well as demand response system types. Due to the nature of data centers, this migration of workloads between servers can potentially save power, through thorough planning of the location of each workload. The solution presented within this thesis asks how this workload migration can be executed, as well as how a demand response system based on this function can be developed. At last, the solution is evaluated on the performance of the workload migrations and the potential expansion of the demand response system is discussed.

Acknowledgements

The project is developed as a Master's Thesis in Software Engineering at the University of Southern Denmark. The authors would like to acknowledge the outstanding counseling and support by Jakob Hviid.

Abbreviations

RQ *Research Question*
A question which the research is based upon.

CPU *Central Processing Unit*
Core processing unit within electronic hardware.

RAM *Random Access Memory*
High speed storage close to the CPU, store data frequent in use.

DAS *Direct-Attached Storage*
Storage connected directly to compute server. No other server has access to this storage.

NAS *Network-Attached Storage*
Storage behind a network interface. Enables storage access through the network.

SAN *Storage Area Network*
Storage on a separate network.

DCT *Data Center Tiers*
Describes data center infrastructure in tiers of level of implementation.

VMM *Virtual Machine Monitor*
Manages virtual machines.

KVM *Kernel-Virtual-Machine*
Hypervisor build directly into OS kernel.

QEMU *Kernel-Virtual-Machine*
Hardware emulator/virtualizer.

DR *Demand Response*
A change in power consumption to match demand.

PBDR *Price-based Demand Response*
Demand response regulated by a pricing event.

ToU *Time-of-use*
Electrical pricing based on the time of use.

VPP *Viable-peak-pricing*
Variable electricity pricing defined as periods in advance.

IBDR *Insentive-Based Demand Response*
DR based on an insentive set by retailers.

AC *Air Conditioning*

VM *Virtual Machine*
A virtualization of a system/operating system.

TCP *Transmission Control Protocol*
A standard which defines a network conversation between applications/systems.

API *Application Programming Interface*
Software intermediary, allowing for interactions between multiple applications.

SDK *Software Development Kit*
Collection of developments tools.

URI *Uniform Resource Identifier*
Physical or logical resource defined by a sequence of characters.

HTTP *Hypertext Transfer Protocol*
Application protocol for transmitting resources.

JSON *JavaScript Object Notation*
Lightweigth standard file format.

RTP *Real-time-pricing*
Hourly electrical pricing defined in advance.

NFS *Network File Server*
A distributed file system.

iSCSI *Small Computer Systems Interface*
A protocol running on top of TCP. Used to linking data storage facilities.

SSD *Solid-state drive*
A non-mechanical hard drive using flash chips.

SATA *Serial ATA*
A storage interface bus.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Data centers are the engine behind the ever-increasing digitalization of the modern world. With the introduction of cloud computing, data centers are used to perform various tasks ranging from research to entertainment. However, while data centers are becoming increasingly prevalent, a problem that needs to be addressed and solved in the near future arises.

1.1 Domain

Renewable energy is becoming increasingly widespread, and Ørsted, the largest energy provider in Denmark, aims to provide green energy to more than 50 million people in 2030[1], as well as relying on a 99% renewable energy production in 2025. The adaptation to renewable energy is improving the sustainability of the power grid, but simultaneously requires stabilization of the power grid. Furthermore, the issue is amplified due to the inability of storing excess energy. When overproduction is occurring, Denmark trades energy with its closest neighbors, Sweden, Norway, and Germany, or stops parts of the production entirely[2].

Another facet of this problem is that whenever the consumption of energy is larger than the production of energy, sustainable energy is not as reliable as fossil energy in that it is not stored but traded with neighboring countries. When consumption of energy is larger than production, the access energy needs to be bought from another party. This is because it is not possible to fully stabilize the network by either storing the access energy production or modulating energy consumption.

A large part of the total energy consumption in Denmark is data centers. In 2019, a single data center could contribute upwards of 4% of the total Danish energy consumption, and by 2029, data centers are expected to have increased Denmark's electricity consumption by as much as 17%[3].

The estimated increase in renewable energy production does not entail that production will exceed the consumption of energy. The estimated increase in data center energy consumption will equate to more instability in the power grid. If this load that the data centers put on the power grid can be diminished by transferring or managing resources within and between data centers, the stability could increase.

1.2 Motivation

Renewable energy sources are being integrated into the power grid, providing a relatively steady amount of energy production, while the energy consumption fluctuates significantly during the day. This results in transfers of energy to neighboring countries, to accommodate this fluctuation, selling when there is an energy surplus, and buying when an energy deficit occurs. While data centers provide a natural strain on the power grid, the potential for energy flexibility exists. When an energy deficit occurs, data center workloads can be migrated to another region or country, which experiences an energy surplus, mitigating the need of transferring energy between the two locations. This system of altering power consumption to better match the demand for power with the supply of power is called a demand response (DR) system.

1.3 Project Scope

The domain of focus is the migration of workloads. The workloads which will be subject to migration, are tasks that are executed on servers in a data center. This definition of workloads is very ambiguous, due to the nature of workloads in a data center. No data center is the same, and therefore, both the purpose and the structure of the data centers processing determines which workloads are executed within. These workloads can take the form of abstractions, f.x. virtualization, and containerization, which both do not describe the task executed within, but how the task is executed. The migration of workloads entails moving a workload from one server to another. The attributes of the migration, i.e. if the migration preserved the state of the migration, as well as the downtime of the given workload which is subject to migration, are both factors that need to be explored. As data centers play a central role, only Danish data centers will be of concern as to limit the scope, while still preserving the general elements of data center. Therefore the scope of this project aims to focus on the migration of containerized and virtualized workloads, in the context of Danish data centers. While more elaborate abstractions and technologies will be discussed, not all will be implemented, and therefore will not be a part of the final product. All of this will be enveloped within a DR context. The DR aspect of the solution will illustrate the potential capabilities of such a technology, and therefore will only be used to realize the theory of a DR system, but will not, by itself, function as a complete system.

1.4 Problem

With the aforementioned problem space in consideration, a list of research questions are set to be answered throughout the thesis:

–***RQ1:*** *How can workloads be migrated to facilitate a DR system in a data center context?*

Furthermore, the project aims to answer a subset of research questions, which all assist in gaining a general understanding of *RQ1*.

–***RQ2:*** *Which attributes are important to a data center participating in such a DR system?*

–***RQ3:*** *How can a DR system in a data center context be designed to support multiple types of workloads?*

–***RQ4:*** *How can the state stay preserved while performing workload migration?*

1.5 Approach

Throughout the project, existing solutions and techniques will be explored through a literature review. This review will provide the foundation of the research and development of a potential solution. The output will be a constructed piece of software, which binds technology and theory together to illustrate a potential solution to the aforementioned problem. This software will then be evaluated and discussed based on the requirements of the solution.

1.6 Report Structure

The report is structured with historical linearity throughout section-1 to section-9, which describes the complete development cycle of the solution. Section-2 and section-3 aims to provide background knowledge of the topics which will be further used and explored throughout the thesis. Section-2 reviews literature that aims to provide a general understanding of foreign topics which aid in the further research of the domain. Section-3 then takes these topics and explores them in greater detail, as well as describing topics

that might not have been explored in the section-2, but can be used to further enhance the understanding of the domain. Section-4 through to section-7 follows the software development cycle of a waterfall model, starting with the gathering of requirement, then the design of the solution based on these requirements, followed by implementation of the solution, and at last an evaluation and presentation of the results. Section-9 then describes the finalizing thoughts on what further development of the solution, would be beneficial, based on findings throughout the research and development. Finally, in section-10 the general findings concerning the problem statement of the thesis are concluded.

2 Literature Review

To understand the domain of data center DR systems, a set of literature is reviewed. The goal of this literature review is to provide a foundation of which topics this literature addresses, as well as if there are aspects that need further exploration.

2.1 Problem formulation

As defined in chapter-1 the problem which is sought to be solved is the adaptation of renewable energy is improving the sustainability of the power grid, but simultaneously requires stabilization of the power grid. Alongside this, data centers are becoming increasingly prevalent, and their energy consumption is a large contributor to the total Danish energy consumption[3]. Because data centers provide promising attributes to participate in a DR system, this is the main focus of the literature review. The question which is sought to be answered by the literature review is as follows:

–How can a data center DR system solve the aforementioned problem to stabilize the power grid?

During the following review, different aspects of this problem will be explored, meaning that not all literature will have exactly this focus, but will provide a general understanding which can assist in solving the problem.

2.2 Literature Search

The literature review throughout this sections is as follows:

[4] A. Wierman et al. "Opportunities and challenges for data center demand response". In: *International Green Computing Conference*. 2014, pp. 1–10. DOI: 10.1109/IGCC.2014.7039172

[5] J. Li, Z. Bao, and Z. Li. "Modeling Demand Response Capability by Internet Data Centers Processing Batch Computing Jobs". In: *IEEE Transactions on Smart Grid* 6.2 (2015), pp. 737–747. DOI: 10.1109/TSG.2014.2363583

[6] H. Wang et al. "Proactive Demand Response for Data Centers: A Win-Win Solution". In: *IEEE Transactions on Smart Grid* 7.3 (2016), pp. 1584–1596. DOI:

10.1109/TSG.2015.2501808

[7] C. Tang and M. Dai. "Demand Response Control Strategies for On-campus Small Data Centers". In: *2012 9th International Conference on Ubiquitous Intelligence and Computing and 9th International Conference on Autonomic and Trusted Computing*. 2012, pp. 217–224. DOI: 10.1109/UIC-ATC.2012.97

Each paper or article describes an important aspect of deploying a data center DR system. In the following sections, the literature will be analyzed and discussed.

2.3 Literature Analysis

Throughout the following section, a thorough analysis of the aforementioned literature will be performed.

2.3.1 Opportunities and Challenges for Data Center Demand Response[4]

The article outlines opportunities and challenges for a potential DR system in a data center context. The two key challenges to overcome are specified as; information and communications technology become more crucial to society, the energy demands are skyrocketing, and the integration of renewable energy into the power grid courses challenges for managing the power grid and has the potential to increase costs considerably. These challenges are magnified by the fact that large-scale effective energy storage is not cost-effective at this point. The idea of solving these challenges by developing a data center DR system is that these challenges are symbiotic. Data centers are flexible, meaning the load can potentially be transferred to another data center. This flexibility is a crucial tool to enable a DR system. This flexibility combined with techniques such as geographical load balancing, load shifting, and more, a data center DR system could prove a viable solution to the aforementioned challenges.

Other than the challenges of implementing a DR system, challenges regarding the adoption of such a system are identified. A list of five challenges regarding regulations and market maturity, risk management, control, market complexity, and market power, describes adoption risks for implementing such a system.

2.3.2 Modeling Demand Response Capability by Internet Data Centers Processing Batch Computing Jobs [5]

The motivation behind this article by data centers being a promising sector for adopting a DR system. This is due to the increasing energy consumption of data centers, and that 41.6% of the costs are energy-related within the described scenario[8], within which 56% is consumed by IT-infrastructure i.e. servers and storage, 30% for cooling, and the remaining for power conditioning, network equipment, lighting, and physical security systems.

In CPU-intense batch computing systems, which is the focus of this article, it is common that a large number of jobs are executed in bursts. It has been shown that these bursts start at 8:00 AM and end at 03:00 PM, after which they transition into an idle state. Because a CPU in idle can consume as much as 60% of the peak-power, another power-saving technique is presented: To consolidate the remaining batch computing jobs, while turning off the idle systems. This, however, presents a problem. The wake-up and shutdown of a server cannot be done too frequently. If this is done extensively, the reliability of the server will degrade. Therefore, it is desirable to have a constant minimum time that the server is either ON or OFF.

The total energy consumption of a data center is described to include two significant parts: The power consumption of the IT computing equipment i.e. servers, storage, etc., and the cooling system infrastructure. Within the IT computing system, the CPU is responsible for the majority of the power consumption, as well as being the most dynamic part of a typical server's power model. This dynamic behavior, as well as the high idle power consumption, makes CPU-intense tasks an ideal subject to a DR system.

To be able to dispatch workloads from a data center, three major topics are described:

- Workload Balance: Each workload submitted to the system should be processed within the scheduled time frame, and each server assignment should satisfy the requirements of that workload. This topic could be realized through live migration techniques.

- QoS Requirement: The users of the system needs to sign an SLA contract with the cloud provider. This SLA contract will specify the QoS requirements of each job and has to be satisfied by the data center DR system.

- Workload Continuous Execution: A batch computing job can be either continuous

or interrupting. If a workload has to provide continuous execution and has already been executed, it should not be interrupted during the workload time frame.

2.3.3 Proactive Demand Response for Data Centers: A Win-Win Solution[6]

The article explores the possibility of reducing data center energy consumption by distributing computation geographically. It is common that large-scale cloud providers like Amazon, Google, Microsoft, etc., build multiple, geographically distributed data centers. The article describes the huge amount of energy data centers consume and that most existing studies have focused on energy management techniques to minimize energy cost from the viewpoint of the data center. However, this study focuses on the energy cost minimization of data centers, based on their impact on the power grid.

Throughout the article a solution is proposed, which connects the smart grid to the data centers, to enable a DR system, in which workloads are assigned to a specific geographical location. Contrary to the previously discussed articles, no workloads are migrated within or between data centers. However, the study of suitable algorithms to optimize energy consumption, as well as the proactive DR system architecture displays a solution that provides useful background information about developing a DR solution for data centers.

2.3.4 Demand Response Control Strategies for On-Campus Small Data Centers[7]

The article has another focus than the previously described articles. While the motivation and problem that is sought to be solved through a data center DR system are that cloud computing is increasing in popularity and usage, and with that increase comes an increasing energy consumption. However, the reliability of commercial data centers has the be a top priority, which a DR system might degrade. The focus of this article is small on-campus data centers, which does not have to have this extreme focus on reliability, which makes them, a more suitable candidate for a DR system than commercial data centers. The article does not propose a specific DR system but discusses and analyses the energy consumption of a traditional on-campus data center.

2.4 Literature Findings

Through the literature search, it became apparent that stabilizing the power grid through a data center DR system, is not a new idea. Because data centers account for such a large percentage of the total national energy consumption as stated in section-1.1, and use flexible workloads i.e., workloads that can, in theory, be transferred between data centers with relative ease, they are an obvious candidate to implement into a DR system. The architecture and deployment of this DR system are not bound and determined. Small on-campus data centers are an obvious candidate for a DR system [7], however, because of the potential impact of a DR system impacting commercial data centers, as well as on-campus data centers, no specific type of data center was chosen. The reasoning for only including small on-campus data centers was because of the rigorous reliability and mission-critical nature of commercial data centers. If the reliability can be persisted, even if the data center participates in a DR system, the commercial data centers can be included in the system. The methods of implementing a DR system in small-campus data centers[7] can, with modifications, be applied to larger data centers, if the reliability is kept intact.

2.5 Literature Review Conclusion

To facilitate a data center DR system, the problem of workload migrations has to be solved. The migration is what facilitates flexibility within the DR system, and allows workloads to be transferred between or within data centers. While workload migration has been discussed already[4], the practical solution to how this migration might work has not. Because there is no definitive standard for what a workload entails in a data center, and each workload has a different migration process, this topic has to be analyzed. This will be discussed and analyzed in section-3.2 section-3.3. The act of migrating workloads is identified as a critical feature for a data center DR system, due to the limitations of action without this feature.

3 Theory & Analysis

In order to prepare a solution to the problem, some elements which drastically impact the problem space have to be addressed. These elements are derived from the literature review in section-2.3, as well as elements surfaced during the reviewing process.

3.1 Data Center

In order to develop a solution that is aimed to be used at data centers, the domain of data centers has to be addressed. Cisco who is one of the networking hardware and software manufacturers defines a data center as:

> *"At its simplest, a data center is a physical facility that organizations use to house their critical applications and data."*
> - CISCO[9]

IBM, who manufactures and sells both computing hardware and software, shares a somewhat similar definition:

> *"A data center is the physical facility that makes enterprise computing possible.."*
> - IBM[10]

Both of these definitions describe the core functionality of a data center. It is essentially a facility that enterprises can use for their critical application and data processing. Other than performance, the aspect of criticality is what significantly sets the commercial data centers apart from traditional data processing.

3.1.1 Data center core elements

A data center consists of three core elements:

- Compute
- Storage
- Networking

Compute is the processing power used to run the applications housed within the data center. Storage is the storage of the data used by the applications, and the network covers the transport of data to, from, within, and between data centers[10]. Due to the criticality of the data center's workloads, additional equipment is used to enhance the security and reliability of the three core elements:

- Environmental controls
- Power

Environmental controls monitor the environment inside the data center, adjusting the temperature, humidity, and airflow of the facility. The aspect of power refers to the electricity supply to the facility. Most data centers use battery backup systems to counteract the adverse possibility of a power outage[10].

3.1.2 Data Center Anatomy

The data center core components have been discussed in section-3.1.1 and are quite similar to regular desktop computers. However, the difference is the scale and management of these components. Instead of the three core components being a processor, hard disk, and a simple networking interface, each component in a data center is an intricate combination of multiple software and hardware solutions. The compute element represents the servers of the data center. The servers are often stored in what is known as racks. The physical dimensions of the server are described by rack units (U), determining the height of the unit. One rack unit is 1.75 inches, with racks being able to house upwards of 50 U, or about 2 meters in height. The typical serving size is between 1U and 4U, meaning that each rack can store between 12 and 50 servers. Each server is equipped with the typical computing hardware, including a central processing unit (CPU) and random-access memory (RAM), which has more processing potential than a traditional computing system, including more processing cores and a greater amount of memory. The amount of servers and racks housed in a data center varies based on the individual data center. Garner estimated that Google housed 2.5 million servers in their data centers back in 2016[11]. However, not all servers in a data center handle the computation. The other two core components are also housed in the server environment. Networking and Storage servers occupy the same space as the compute servers. Network servers handle the connection between compute servers and enable access to each server from

fx an administration portal. The network servers also handle the connection between compute and storage servers, enabling the compute servers to access the shared storage of the storage servers. The architecture of the storage server depends on the use-case of the storage. Sometimes a storage server is not needed and only local storage is used on the compute server. The different storage solutions within data-center context is as follows[12]:

- Direct-Attached Storage (DAS) is the setup in which the storage is connected directly to the compute server. No other server has access to this storage. The advantage of such a solution lies in the storage is directly attached to the unit which interacts with it, it has a performance advantage. No network bottlenecks can happen on a DAS because of the lack of a network interface.

- Network-Attached Storage (NAS) is much like DAS, however behind a network interface. This enables the storage to be accessed through the network interface and accessed by multiple machines at a time. This requires a storage server and has the advantage of convenience. The most used protocols for a NAS storage server are Network File System (NFS) and Common Internet File System (CIFS). The disadvantage of NAS is that the addition of a network storage solution introduces a potential bottleneck due to the storage being network attached. If the network transfer is slower than the storage speeds, the storage speeds will match the network speeds at best. Second, depending on the network, the storage server has to compete with other traffic on the network.

- Storage Area Network (SAN) is a similar solution to NAS. The main difference being that SAN operates on a separate network, eliminating the potential downside of competing for network traffic.

In the case of NAS and SAN, multiple servers are connected to the storage solution and therefore a storage server can supply entire or multiple racks with storage simultaneously.

Although the structure and content of a data center vary from data center to data center, almost every data center has these described attributes. The reason these attributes are common between data centers is that they either increase productivity, security, or reliability, which are all part of the main selling point of data centers. The

combined elements of a data center are used to describe a data center in terms of tiers, also known as Data Center Tiers.

3.1.3 Data Center Tiers

Data center tiers (DCT) are used to describe specific data center infrastructures. The lowest DCT defines the simplest infrastructure implementation. Tier 1 DCT could entail simple cooling and few, if any, redundancy and backup, furthermore the system availability is also affected and is the lowest of the tiers. Each DCT level defines the system with better specifications and performance.

3.1.4 Service Level Agreements

A Service Level Agreement (SLA) is a contract between data center operators and customers which defines several metrics the service provider guarantees. Common metrics range from "System availability", "Acceptable Data Loss", "Recovery Time" and "Performance". Every category defines certain metrics which impact the hosted application. Within the SLA, the DCT is defined as a subsection of the following categories.

- System Availability - Defines the availability of the servers. The availability is defined by an Availability Target that identifies the rough availability for the server in %-uptime and in an approximate of downtime per year. Example: An availability target of 99.99% equates to a 52-minute downtime per year.
- Acceptable Data Loss - The amount of data that is acceptable to lose over a period of time. Recovery Time - Defines the amount of time from some disaster it would take for the system to back up and running. Could be defined as an hourly period from disaster to active again.
- Performance - The transactional time between server and client systems. (Could be a non-functional requirement)

Furthermore, the SLA defines "Exceptions and Limitations" which define any exceptions to the SLA conditions, scope and application. These could be "System Availability" at the holiday period and "Service provider liability" at natural disasters etc.

The SLA defines "Responsibilities and responses" which define several requirements the customer must follow to help the service provider in their operations of the customer's service, and likewise, the service provider also has several requirements they must follow.

Lastly, the SLA should include the pricing models and charges for each service type. The models should include the hardware specifications like storage capacity, CPU frequency, data throughput of the storage controllers. This defines a rough example of an SLA.

The SLA for our solution would impact the "System Availability" as the workload migration will include downtime. The downtime will thereby depend on the migration threshold and the migration method.

3.2 Data center workloads

The definition of a workload within a data center is a complex definition. This is because of abstractions. Virtualization and containerization are both abstractions of the traditional server environment, and therefore the individual workload running within this environment is irrelevant. The workload is the environment itself. These workloads are then managed within the data center, either by hypervisors in the case of VMs or by container management software like Docker for containers.

3.2.1 Virtual Machines

A virtual machine (VM) is an abstraction of a computer system, which has its own virtualized CPU, memory, network interface, and storage. This virtualization of devices is created based on a physical system. This virtualization of resources for the VM is done by a hypervisor, which separates the virtualized systems resources from the physical system. The physical machines, also known as host machines, are equipped with the hypervisor and can facilitate multiple VMs. The hardware of the host is treated as a pool of resources, which each VM can allocate and use. This makes VMs highly scalable, in that resources can easily be relocated between existing or new VMs. [13]

3.2.2 Hypervisors

Also known as VM Monitor (VMM), hypervisors are what manages VMs. It does this by assigning each VM a portion of the host machine's hardware resources, including CPU, memory, and storage. This prevents the VMs to interfere with each other, running each VM in an enclosed environment.

The type of hypervisors can be separated into two groups; Type 1 and Type 2.

- Type 1 hypervisors run directly on the underlying physical hardware of the host, interacting directly with the CPU, memory, and storage, and replaces the host's operating system. For this reason, Type 1 hypervisors are also known as bare-metal hypervisors. Due to the fact that Type 1 hypervisors interact directly with the machine's hardware, these hypervisors are highly efficient. However, it is not all Type 1 hypervisors that replace the operating system of the host machine. Hypervisors such as KVM (Kernel-Virtual-Machine) are built directly into the OS kernel, which enables the hypervisor to emulate the physical machine's processor entirely in software[14].

- Type 2 hypervisors run as an application within the operating system. This type is rarely used in a server-based environment and is instead suitable for PC users which wish to run multiple operating systems on their machine.

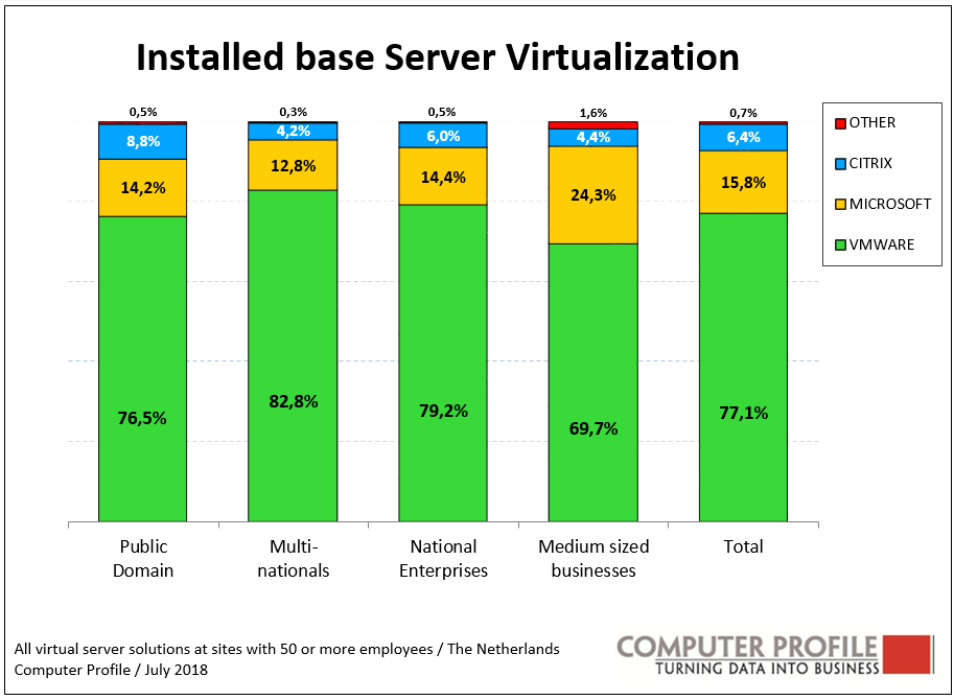


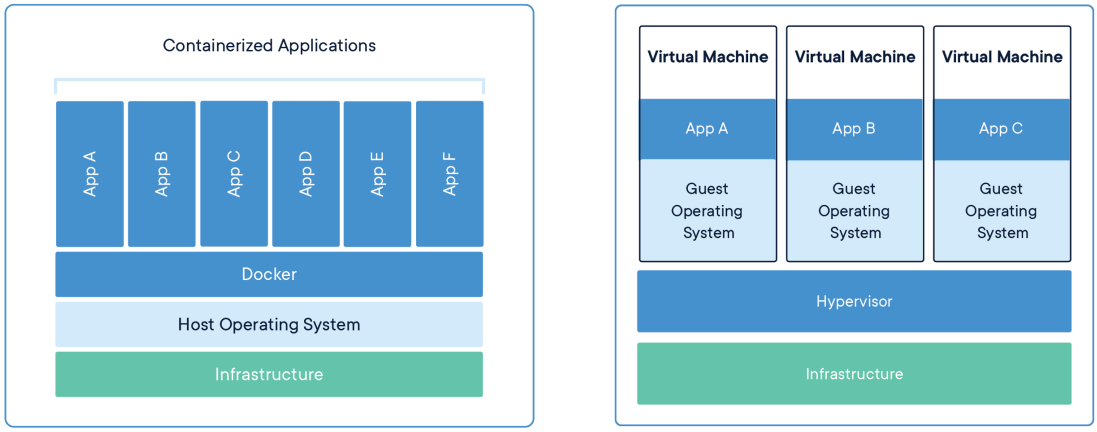
Figure 1: Industry server virtualization market share [15]

The main difference between the two types of hypervisors is speed and convenience. Type 1 is generally faster than Type 2, due to the reduced overhead, while Type 2 does not require a separate machine to manage and administer the hypervisor. However, as mentioned before, Type 2 hypervisors are rarely used in a server-based environment.

As seen in figure-1 the vast majority of server-based virtualization software used in the Netherlands in 2018 was made by VMware, Microsoft, and Citrix. VMware provides a variety of hypervisors, however the most common being vSphere for server-based environments, which is a Type 1 hypervisor. Microsoft's Hyper-V, as well as Citrix's XenServer, are also both Type 1 hypervisors. This confirms the notion that Type 1 hypervisors are more dominant within server-based environments.

3.2.3 Docker

The workload found in a Docker environment is called a container. Docker containers are packaged up code and dependencies, which allows the application to run reliably from one computing environment to another. Docker containers are created based on a container image. This image contains all the information needed to run the application i.e. code, system tools, system libraries, settings, etc.[16]. This provides a simple way of creating multiple applications, across multiple environments, without running into dependency issues. Just like a VM, Docker virtualizes an environment, but in a different way. VMs virtualize the hardware, whereas a docker container virtualizes the operating system. This makes containers portable and efficient. [16]. The difference is shown in figure-2 where the structure is clearly different. Where both of the setups have an underlying host (infrastructure) which provides an operating system, and both have an orchestrating layer, in containerized environments docker and a hypervisor in a VM environment, the Docker environment virtualizes the operating system of the host and therefore does not have a separate operating system for each container.



(a) Docker (b) VM

Figure 2: Docker and VM Structure[16]

3.2.4 Kubernetes

Kubernetes is a tool for automating containerized applications. Docker and Kubernetes are both containerization tools, but the main difference is that where Docker runs on a single node when creating a Docker container, Kubernetes is meant to run on a cluster. Systems like this also exist in Docker called Docker Swarm, but Kubernetes does this more extensively.

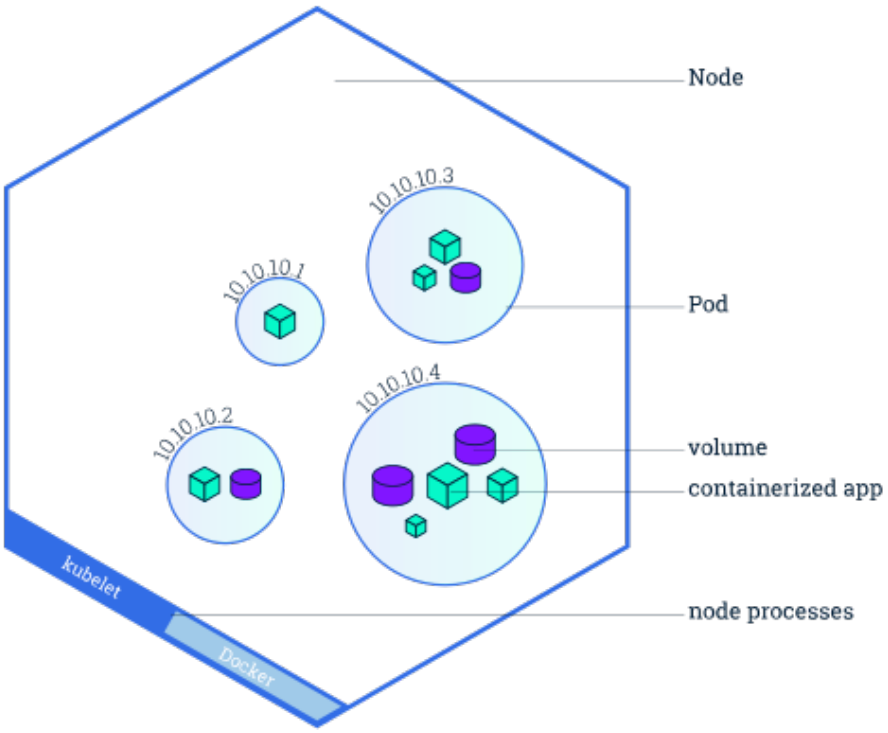


Figure 3: Kubernetes architecture structure including containers with volumes running in pods. All pods are grouped within a single node. Multiple nodes can run at the same time.[17]

The environment of Kubernetes is also different from that of Docker. Each container is contained in a pod. Within each pod, different nodes can communicate with each other on localhost, because within the pod the containers are utilizing the same network namespace. A number of these pods are contained in a node. Communication between pods on a node is done by a virtualized network orchestrated by Kubernetes. Much like a network connection, each pod has access to an ethernet device called eth0, which in this case is virtualized. This facilitates communication between pods in a node possible. Communication between pods within different nodes is done by each pod getting a designated IP, which will then facilitate communication. [18]

These nodes are all contained in a cluster which is the primary workload of a Kubernetes system. This is a highly scalable system in that new containers, pods, and nodes can be added to the existing system, without disruption to the existing elements. The container is still, as in the Docker environment, virtualizing the operating system

and not the hardware like VMs. The structure of a Kubernetes cluster is illustrated in figure-3.

3.3 Data Center Workload Migrations

With the structure and definitions of the workloads established, the topic of migration can be discussed. With each type of workload having different functionality, framework, and use-case, each type of workload has its own migration techniques. The definition of a workload migration differs from workload to workload, but in general, it involves relocating resources from one host to another. Within each workload type different migration types also exist, each with its own advantages and disadvantages. These types of migration, as well as a general definition of what migration entails for that workload, will be discussed in the following section.

The type of migration that will be in focus is that which moves an entire workload, or enables the instantiation of a workload, from one host to another. The main priority is preserving the state of the workload, as well as minimizing downtime.

3.4 Docker

The migration of Docker containers can be done in several ways. Docker does not provide a straightforward way of migrating containers while preserving the state, however, some techniques can be used to achieve this goal. It is important to note, that not all containers have a state, and therefore migration of stateless containers, requires only migration of volume if any.

3.4.1 Export/Import

It is possible to import and export containers in Docker. This simply copies the container and does not preserve the state of the previously running container. While this is a simple solution it is not complete. The export command does not export the contents of volumes associated with the container [19]. In the following example 1, a container is exported and compressed using a compression method, in this case, gzip. The container is then uncompressed by another host and imported by using docker import. The container can then be started by running the container using docker run.

```
1 // Export from host
```

```
docker export container-name | gzip > container-name.gz

// Import to host
zcat container-name.gz | docker import - container-name

// Run container
docker run container-name .
```

Listing 1: Docker export: Export a container's filesystem, Docker import: Imports the exported container with preserved file system state.

This technique preserves the file system state of the container, and nothing more. The export command saves the container's file system, and import imports the filesystem as a container. Therefore, the container's state is not preserved. However, by recreating the tasks running within the container, based on the filesystem, a similar container can be built.

3.4.2 Save/Load

Another alternative is to use commit, save and load. Docker save stores the image as a tar file, which can then be loaded onto another machine through the load command. However, this only preserved the image, not the file system of the container. If the container does not alter the file system from its initial creation, a save and load are sufficient to preserve the entire state of the container. However, if this is not the case, the commit command can be used. Docker commit creates a new image based on a container's changes. When a commit is used before the save and load command, the container's changes to the filesystem are preserved, and the container can be migrated between machines without inherent file system loss. A simple example of this usage is shown in listing-2.

```
// Commit image
docker commit container-name image-name[TAG]

// Save from host
docker save image-name[TAG] > file.tar

// Load to host
cat file.tar | docker load
```

```
9
10 // Run container
11 docker run image-name[TAG] .
```

Listing 2: Docker save: Save one or multiple images to tar file, Docker load: Load image from tar file.

Throughout these two techniques, the preserved state has only included the filesystem, however, if a container is currently processing a task, there is an aspect of memory that has to be preserved as well. If the memory state of the container is restarted every time a migration occurs, long-lasting tasks may never finish due to the constant resetting of the memory state. However, docker does not, by the initial configuration, provide such a feature. However, an experimental feature of Docker exists implementing the tool CRIU.

3.4.3 Checkpoint/Restore

CRIU is a project which implements a checkpoint/restore functionality for Linux systems[20]. This checkpoint/restore functionality makes it possible to freeze a container and checkpoint the memory state of the container to a disk. The CRIU project has been implemented in a series of container software including Podman and Docker. CRIU has a vast variety of use-cases[21], many of which enables workload migrations. However, the functionality that will be of focus during this thesis is the Docker implementation, which enables the "docker checkpoint" command. Listing-3 shows a potential use of the docker checkpoint feature.

```
1 // Create checkpoint
2 docker checkpoint create container-name checkpoint1
3
4 // Restore container from checkpoint
5 docker start --checkpoint checkpoint1 container-name
```

Listing 3: Docker checkpoint: Create a checkpoint for running container preserving state

This command creates a checkpoint of a running container and stores it on the disk. By combining this checkpoint with a container filesystem, the complete state of a container can be stored, and potentially migrated to another machine, i.e. combining commit, save and load with the checkpoint feature the complete state of the container is preserved.

A complete example of this can be seen in listing-4.

```
// Commit image
docker commit container-name image-name[TAG]
// Save from host
docker save image-name[TAG] > file.tar
// Create checkpoint
docker checkpoint create container-name checkpoint1


// Load to host
cat file.tar | docker load
// Run container
docker run image-name[TAG] .
// Restore container from checkpoint
docker start --checkpoint checkpoint1 container-name
```

Listing 4: Docker Commit: Generates new image based on container changes. Saves image to tar file. Creates checkpoint to preserve state, load docker image. Run image and restore with checkpoint.

3.5 Docker Swarm

When running a Docker Swarm, every node which is actively running has the ACTIVE availability tag. This means that the node is active and that it can receive new workloads. When migrating a node the process is simple. Because a Docker Swarm is decentralized and all nodes contribute to the workload at hand, a node's availability tag can simply be set to Drain. The Swarm manager will then stop the initiation of new tasks on the node, as well as replicate running tasks on nodes that are active. An example if this is shown in listing-5.

```
// Set the Docker node availability to drain
docker node update --availability drain <NODE-ID>
```

Listing 5: Docker Swarm Drain command. This is based on each node individually.

When the node is shut down, if desired, a new node can be created on another machine to re-instantiate the previous capacity of the swarm.

3.6 Kubernetes

Much like a Docker Swarm, the nodes of a Kubernetes cluster are relatively easy to migrate to another. Because each node is part of a greater system i.e. the cluster, each node is not solely responsible for the workload. The process of migrating nodes to a new machine is also, just like Docker Swarm very simple. Each node will need to be drained of its tasks, then stopped, and to reinstate the previous capacity, a new node can be started on a new machine.

To do a Kubernetes node migration a few steps must be performed[22]. First, the existing nodes need to be cordoned, marking each node as unscheduled. This stops the scheduling of new tasks on the nodes. Second, the nodes need to be drained. This evicts the workloads of the existing node pool, disabling them from receiving new tasks.

After this has been executed the old nodes can be deleted and new nodes can be added to the Kubernetes cluster, reinstating the previous capacity.

3.7 Virtual Machines

VMs have a different migration technique than containers, both due to the inherent architectural difference, but also because of the lack of a standardized method of VM migration. This is also true for containers, in that, a Docker containers migration process is not the same as a Kubernetes pod migration. Although it is not a one-to-one comparison, the same concept applies to VM migration, and how each hypervisor migrates a VM.

Even though migration techniques are different between hypervisors, the types of migration can be categorized into two main categories:

- Cold Migration occurs when a VM is migrated while inactive.
- Live Migration occurs when a VM is migrated while active.

The migration process is slightly different between these methods of migration, in that the state of a VM experiencing live migration might still undergo changes while the migration process is happening. During a cold migration, the migration is as simple as copying the filesystem to another hypervisor and reinstating the VM on that hypervisor. During a live migration, this process is not possible, due to the current state of the VM which also has to be migrated. The live migration process copies the running system

onto the new hypervisor, just like a cold migration. When the migration of the system is close to being completed, the last part of the active system, as well as potential changes to the system during the migration is copied again, to ensure that the state of the VM is preserved. Due to this behavior, live migration is more time-consuming than cold migration.

However, this increase in migration time does not impact the user as much as a cold migration. To do a cold migration, the VM has to be shut down, and when migrated, started on the new hypervisor to enable usage of the VM. During this time the system is unavailable. During a live migration, however, the user does not experience much, if any downtime of the system, and can keep working while the migration process is occurring.

3.7.1 Live Migration

The major difference between a cold and live migration is that the memory state has to be transferred during a live migration and not during a cold migration. The conventional live migration process of memory can be described with three phases[23]:

- Push phase: The source VM while certain parts of the memory are pushed to the new destination. Memory edited during this phase must be re-sent.
- Stop-and-copy phase: The source VM is stopped, and the remaining parts of the memory are copied across to the destination, where the destination VM is started.
- Pull phase: If the destination VM tries to access memory that is not yet copied, the file is tagged as "pulled" and is copied from the source VM to the destination VM immediately.

This process ensures that the memory is copied reliably across source and destination, however, not every phase is typically used in the typical migration process. Most practical solutions select one or two of these phases to be incorporated in the migration process. The potential downtime of the VM happens during the stop-and-copy phase, where the source VM is stopped and the destination VM is started.

3.7.2 Virtual Machine Migration Techniques

Two main techniques for VM migration, both involve the phases described in section-3.7.1. These techniques are known as pre-and post-copy[24].

- The pre-copy technique uses the push phase as well as the stop-and-copy phase. Due to this technique not including the pull phase, the entire memory state has to be transferred within the first two phases. The way this is done is through convergence. Convergence is when the copying of pages is done through iterations, and with each iteration, an increasing amount of the source VM's memory is copied to the destination VM. When either the iterations reach a certain value, or the number of uncopied pages reaches a certain threshold, a phase-change to the stop-and-copy phase begins, and the migration can happen. The memory is copied before the actual instantiation of the source VM, hence the name of the technique; pre-copy.

- The post-copy technique, like the pre-copy, utilizes the stop-and-copy phase, however, the push phase is exchanged with the pull phase. The post-copy technique starts with the stop-and-copy phase, suspending the source VM and transferring a minimal state of the memory to the destination VM. When this initial migration is done, the destination VM pulls the remaining memory from the source. The system is therefore undergoing a small initial migration, followed by a pull phase which will fetch the remaining memory state, hence the name of the technique; post-copy.

3.8 Energy

Throughout the energy section, different types of DR will be explored with accompanying energy-saving techniques.

3.8.1 DR System Types

DR defines a cost-effective way of maintaining grid reliability and security. DR is a global technique to lower electricity consumption by either reducing demand or utilizing on-site energy storage in response to an incentive or signal. DR can then be divided into two types:

- Price-based
- Incentive-based

3.8.2 Price-Based

Price-based demand response (PBDR) influences customers to shift their electricity usage behavior in accordance with certain pricing policies, where time-of-use (ToU) is conventional and most widely applied. ToU incentivizes customers to shift their consumption to alleviate the market energy costs and reduce risks on the supply side. As ToU defines the pricing in accordance to multiple factors, such as time of day, holidays, and season. Besides ToU other pricing policies exist such as Real-time-pricing (RTP) and Variable-peak-pricing (VPP). The RTP policy releases the prices ahead of time, therefore allowing a DR system to adapt for the day ahead. VPP functions as a hybrid of ToU and RTP, by defining a day into On-peaks and Off-peaks. In the summertime, the On-peak hour's range could be 6 hours with the rest being Off-peaks. The On-peak hours represent an elevated price range due to higher grid demand, therefore incentivizing customers to load shift the Off-peak.

3.8.3 Incentive-Based

Incentive-based demand response (IBDR) offers customers an incentive to reduce/shift their load. IBDR contains several programs used to apply load reduction/shifting.

- Direct load control - Agreement between customers and utility company. Allows utilities to remote control appliances such as AC and water heaters and is mostly used in the residential sector.

- Interruptible/Curtailable service - Customers are enquired by utilities to reduce the load by a certain level to ease the congested system load, in return the customers are rewarded with a bill discount or rate discount. Customers will receive a penalty if they do not meet the tasked energy reduction level.

- Demand bidding - Customers are not required to partake in DR programs, however decide themselves. Based on demand and generation, utility companies release the total amount of electricity for bidding. Customers bid on the electricity curtailment, utilities accept the bid, and customers are required to follow the curtailment. If customers fail to comply a penalty is issued.

- Capacity market - Customers are to reduce their consumption if the system reserve is short of electricity. This announcement is usually released a day ahead.

- Ancillary service market - As demand bidding, customers bid for curtailments. Once accepted, customers are paid at spot price and the curtailments are used as an operational reserve.

Each IBDR solution brings its own benefits which depended on the application can serve as a great solution for saving on their energy cost while benefiting the retailers with a more stable grid.

3.9 Energy Saving Techniques

With two DR definitions, it is important to clarify the techniques used to achieve a functional DR. Within DR, there are several techniques used to shift or reduce consumption. Each technique offers its own benefits and applies to different DR scenarios. The three most common techniques are:

- Load-shifting

- Peak-shaving

- Valley-filling

Each technique will be described in more detail below.

3.9.1 Load-shifting

As flexible load management solutions become more prevalent and beneficial for multiple industries, it is important to understand how load shifting works. It refers to down- or upscaling the load in accordance with the current peak consumption pricing signals. Therefore, load shifting downscales the load of the system at higher peak pricing, followed by an upscaling period of consumption with lower pricing to counter the downscaled period. Load shifting, therefore, does not result in less consumption, however, the consumption is consumed at a different time, which potentially could have higher generation levels with a lesser price point. [25]

3.9.2 Peak-shaving

The energy industry refers to power-saving techniques such as Peak-shaving or Peak-clipping as a method of leveling peak loads, by "shedding" the peak consumption for a short period of time, avoiding prolonged consumption at higher energy pricing. Implementation of peak shaving utilizes different approaches for regulating/transforming the energy efficiency of a given system. By monitoring/tracking the energy load based on a threshold, the system regulates the load, based on outputs generated by the monitoring system and its ability to predict the accumulated peak load. Other possible approaches such as energy storage serve as a load reducing technique, by providing supplementary power to the system, which absorbs the additional load. [26]

3.9.3 Valley-filling

Valley-filling is a load-balancing technique, by moving the expected peak-load of a system, to off-peak valleys, it is expected to equate to a total average lower cost. Generally, valley fillings are used to shift energy consumption to earlier in the day, where production levels are higher and energy pricing is lower. This results in a streamlined consumption curve, where the average total price equates to lower costs than without any load balancing techniques. In practice, it is used in multiple sectors such as Electric Vehicles, Home appliances, heating etc.[27]

4 Requirements

In the following section, the collection of requirements, as well as the requirements themself will be described. The format for the requirements will be divided into two categories, functional and non-functional requirements.

4.1 Collection of Requirements

The collection of requirements were based on which extensions the system needs to be able to execute in order to answer the research questions presented in section-1.4. This includes the ability to migrate a workload from source to destination. Together with the foundation of the requirements being based on the research questions, the requirements are also based on the theory and analysis of the topic of data centers and DR systems. This includes the critical aspects of downtime and state preservation, which also have to be a part of the requirements. Other than the elements drawn from previous sections, general requirements of usability, as well as which type of DR system to implement are also specified in the requirements.

To rank the requirements a prioritization technique has to be applied. The MoSCoW priority[28] system was picked because it provides a better description of each priority than a simple numerical system and enables scope indication with the "Will not have" priority.

All requirements, both functional as well as non-functional, are prioritized with the MoSCoW priority system. Further explanation of the functional- and non-functional requirements are listed in table-1 and table-2 respectively.

4.2 Functional Requirements

The functional requirements describe the core functionality of the system. Because the system handles migrations, most of the requirements in table-1 are to do with migrations. F01 and F02 state that migration between two servers, sharing the same storage is a "Must". This is due to this being the core of the system. If no migrations are possible, there is not a DR system. F03 and F04 are similar to F01 and F02, with the only difference being the distance of the migration. Being able to migrate between servers with different geographical regions, means that a workload can be migrated from one data center to another. F05 is again a "Must" priority due to the criticality of the

requirement. Not being able to preserve the state between migrations entails that data loss will happen on the workloads which are migrated. This is of course not true for every workload, but the vast majority of tasks have a state which needs to be preserved. F06 describes the type of DR system which will be implemented. The Price-based system is further described in section-3.8.2. When migration happens the downtime of the workload should be documented as described in F07. This information can be used to prove that the SLA is complied with. The last requirements F08 and F09 are to do with interactions with the system. Because the system is not completely autonomous, a data center authority has to be able to add and remove workloads from the system, as well as monitor the active workloads in the system.

ID	**Functional Requirement**	**Priority**
F01	Migration of Docker containers between two servers, sharing the same storage.	Must
F02	Migration of VM's between two servers, sharing the same storage.	Must
F03	Migration of Docker containers between two servers, at different geographical locations.	Should
F04	Migration of VM's between two servers, at different geographical locations.	Should
F05	The migrations of workloads preserves the state of the workload.	Must
F06	The DR system is driven by a Price-Based system.	Could
F07	Downtime of workloads is documented.	Should
F08	Add/Remove workloads from the DR system	Must
F09	Monitor active workloads in the DR system	Should

Table 1: Functional Requirements

4.3 Non-Functional Requirements

As described in section-3.1.4, the downtime of data centers is a critical attribute. Because of this, the requirement N01 states that the downtime of workloads has to meet this critical attribute. However, the performance of the DR system cannot be ensured at this point, hence the "Should" prioritization. NO2 depends on the source of input data which will be used to determine the migrations in the DR system. The reasonable

length entails that the interval in which input data is collected needs to be sufficient to do impactful migrations. If new input data is available each hour, then it needs to be sampled each hour. However if the input data is available each second, then the input data can be sampled each hour with fx a running average. NO3 and NO4 describe user experience. The system needs to be transparent in which actions are about to be executed and are about the execute. These are prioritized as "Could", because it does not inherently increase system functionality, but provides overall transparency of the system.

ID	**Non-Functional Requirement**	**Priority**
N01	Downtime of workloads is acceptable in relation to the data center tier system.	Should
N02	Responses to changes in the energy input data is in intervals of reasonable length.	Should
N03	The user interface is providing a clear understanding of which actions are about to be executed.	Could
N04	The user interface is providing a clear understanding of which actions that has recently been executed.	Could

Table 2: Non-Functional Requirements

4.4 Use-cases

With the requirements established it is now possible to determine the use-cases of the system. To illustrate the requirement interactions a use-case diagram was made. This diagram shows the use-cases of the system, and together with the requirements in table-1 and 2, provides a general understanding of the actors interaction with the system. The use-case diagram of the system is illustrated in figure-4.

The use case diagram has three primary actors:

- The Customer of the data center, who has a workload running on the system. The customer has to be able to access this workload directly, but not much more within the system.
- Operator is the actor who is responsible for data governance within the data center. The operator interacts with three use-cases; initiation of workloads, monitoring of

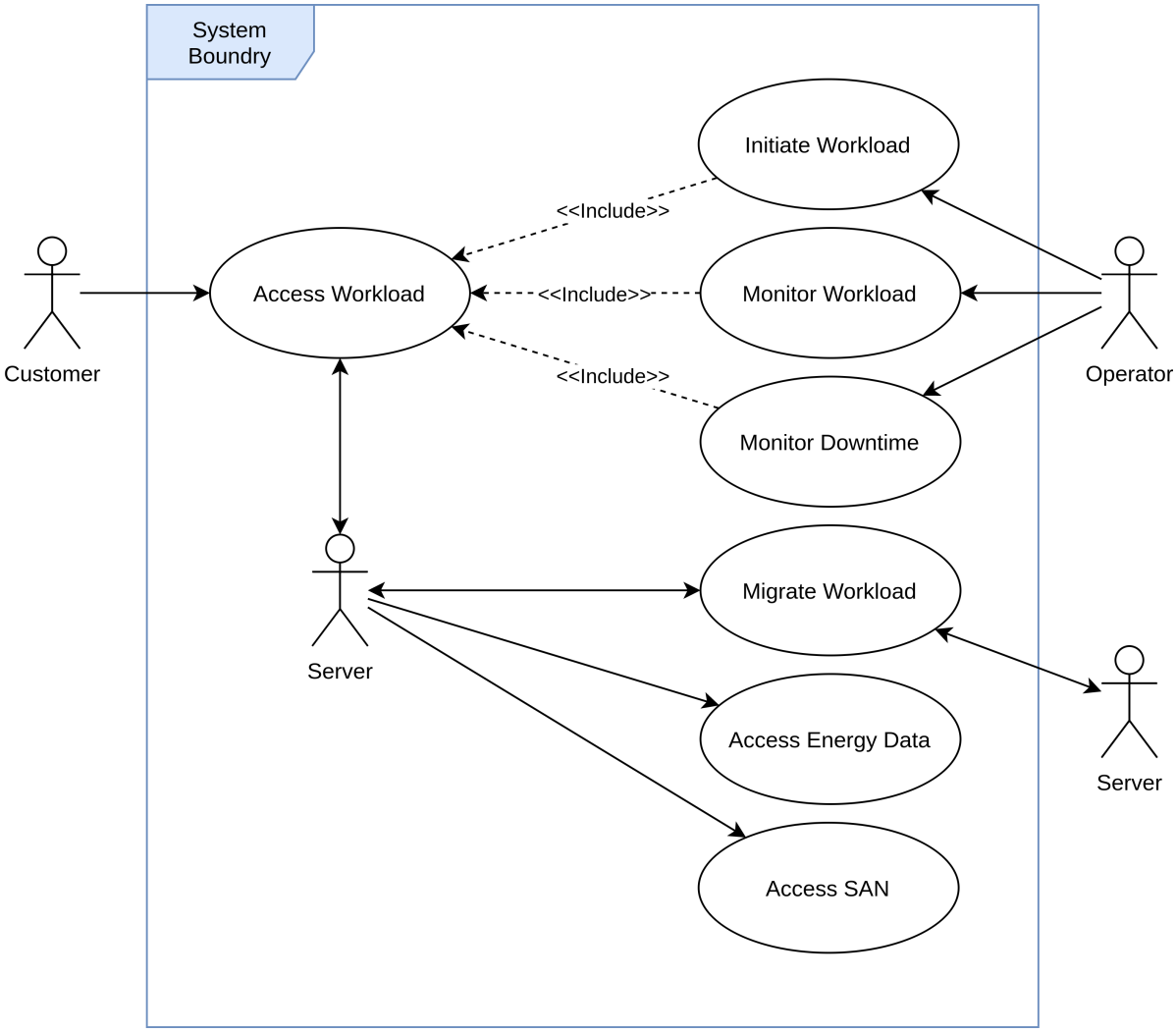


Figure 4: Use-case diagram illustrating the potential use-cases of the system. Within the system boundry is the system which is in development, and outside lies the actors interacting with the system.

workloads, and monitoring downtime of workloads, which includes the use-case of accessing the workload. Whenever a new workload is added to the system, the operator is the responsible entity and makes sure that the workload is running currently by monitoring it through the system. The Operator is also responsible for the overall downtime of the system. Each workload has a documented downtime, and it is the operators responsibility that this downtime is acceptable.

- The last actors are the servers. The server within the system boundary contains the workload and has some additional use-cases that can be initiated by the system.

This server can access the storage, in this case, a Storage Area Network (SAN), which holds the storage of the workloads, as well as accessing the Energy Data. The Energy Data access is needed to complete the last use-case which is workload migration. The server within the system boundary can migrate a workload to another server, based on the Energy Data. The other server is located outside the system boundary and therefore does not interact with the rest of the system in this instance.

The use-case diagram shows the requirements in action as well as their relations with each other. Through the use-case diagram, it is also possible to see access levels, which indicate which actor has access to what functionality. In the case of the operator can be further split into more roles, based on the individual organization. The important element is that the operator represent the commercial data center which the servers is located and the Customer actor is a customer of this data center. The server outside the system boundary does not have to be a part of the commercial data center, however it has to have an implementation of a compatible DR system.

5 Design

In the following section, the design and design decisions will be realized based on the stated requirements in section-4.

5.1 System Components

As seen in figure-5 the system is divided into three main layers; client, guest, and workloads. The client layer hosts the Controller software. The Controller controls an unlimited amount of guests through their GuestController software, however, only three guests are depicted in figure-5. The Controller can be run on any system and does not have to have access to control each workload. This is because the Controller is communicating with each of the GuestControllers, which has access to the workloads directly. The goal of the Controller is therefore to orchestrate the GuestControllers, which then orchestrates the workloads.

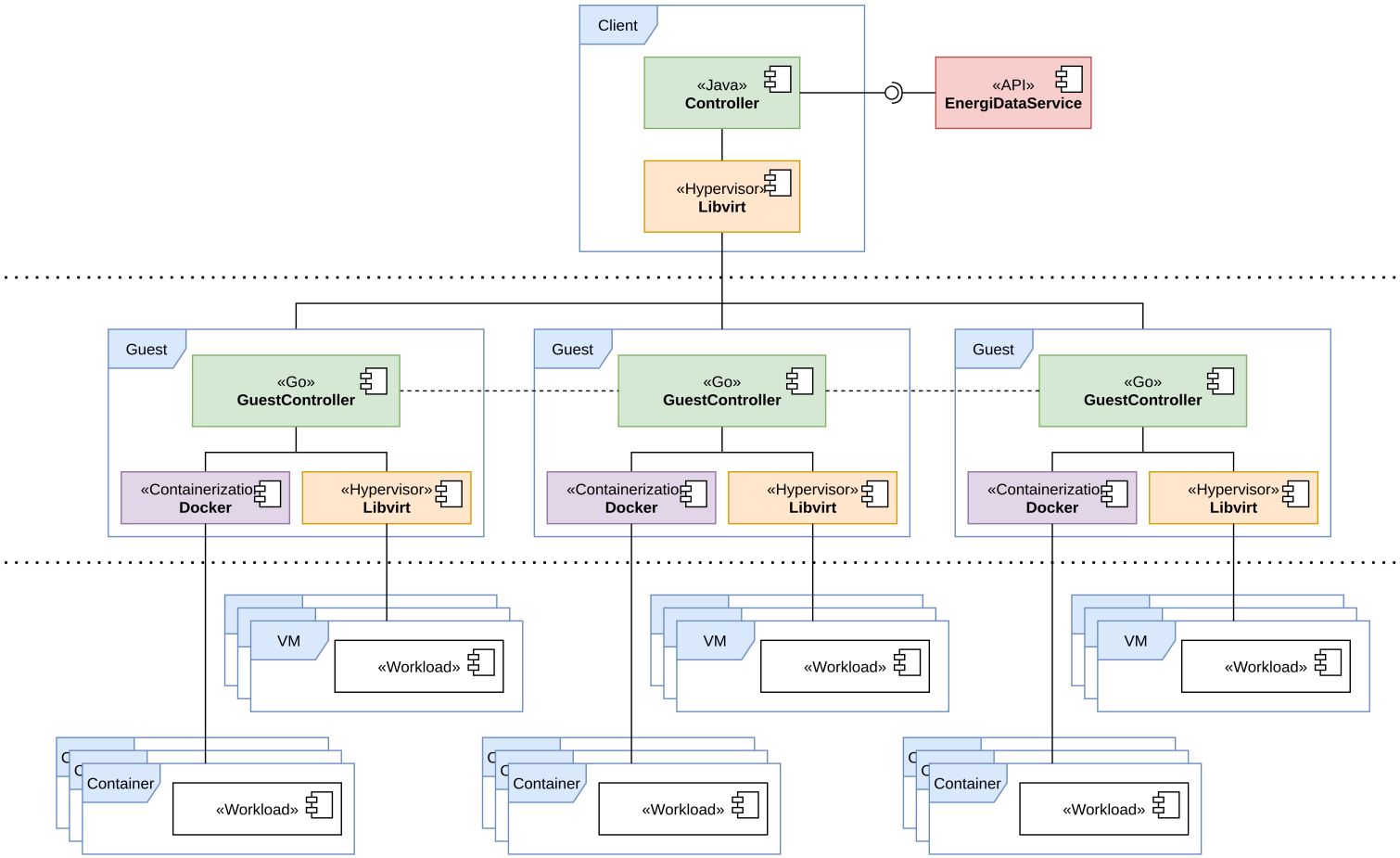


Figure 5: System design architecture: Tier 1: Client layer: Controls workload migration and energy data fetching, Guest layer: Manages migration operations executed by the client controller, Workload layer: Workloads running on guests which can be migrated between guests.

5.1.1 Controller

The client controller is comprised of two managers and a energy data fetcher;

- VMManager
- GuestManager
- EnergyDataFetcher

Each manager has its responsibilities, which will be clarified in deeper detail in the implementation section.

5.1.2 VMManager

The VMManager is responsible for maintaining a list of workloads received from the GuestManager. Since the VMManager acts as a "Master Controller" the responsibilities are quite vast. As the GuestManager is responsible for workloads from the GuestController, the VMManager maintains the complete list of workloads. When workloads are to be migrated, the VMManager handles the connectivity to the GuestControllers and ensures the migration is complete.

5.1.3 GuestManager

The GuestManager is responsible for maintaining workloads from the GuestControllers, and therefore acts as a middle-man between the GuestControllers and the VMManager. As the GuestController transmits new workloads to the client application, the GuestManager ensures workloads are active on the guest and thereafter adds the workloads to the VMManager workload list.

5.1.4 EnergyDataFetcher

To enable the systems DR on a price-based incentive, the EnergyDataFetcher is responsible for collecting electricity prices based on RTP. Afterward, the collected price data are used for generating a migration threshold, to which each workload will adhere unless a specific threshold for the workload is set or auto migration is disabled.

5.1.5 GuestController

The GuestController acts as a controller for the workloads. It contains a set of workloads that it has control over, as well as a connection to the Controller software. The responsibility of the GuestController is to act upon the instructions given by the Controller and nothing more. These instructions need to be logged and executed based on inputs from the Controller, via fx. a TCP connection. This makes the GuestController act like a server, carrying out commands whenever the Controller instructs it to do so. These instructions include:

- Get information about workloads, which participate in the DR system.
- Post information about workloads, to introduce them into the DR system.
- Post a request to migrate a workload from this GuestController to another.

The information about workloads is specific to the workload type. To get the information about Docker containers, the Docker API has to be used. The same goes for the VMs where a hypervisor API is used to manage the VMs.

5.2 Language Support

5.2.1 Docker

To manage and migrate Docker containers, it is vital that the various controllers can access Docker. This is done through the Docker API. Docker provides an official Python and Golang SDK, but these are not the only option. Since the Docker API is public, a lot of unofficial Docker libraries can be used for the same purpose as the official SDKs. In table-3 is a compilation of the Docker libraries which can be used to orchestrate containers.

Name	**Description**
Docker Python SDK	Official Docker SDK.
Docker Golang SDK	Official Docker SDK, supports experimental features.
Docker-client	Unofficial Docker Java API

Table 3: Docker Language Support

The main difference between the libraries listed in table-3, is that the Docker Golang SDK supports experimental features. These features are experimental, as the name implies, and are not a part of the stable release of Docker. These are not supposed to be run in a production environment, due to either instability, lack of support or contain unsolved problems. The experimental feature that is of interest is the "docker checkpoint" feature. This is the feature that runs on CRIU and enables containers to be checkpointed and restored from that checkpoint. The checkpoint can then be transferred to another machine and therefore aid in the migration process.

5.2.2 VM Orchestrators

The management of VMs depends greatly on which hypervisor is in use. Most hypervisors have a designated API that manages their VMs and does not provide native inter compatibility with other hypervisors. However, there exists plenty of APIs which can manage a variety of VMs from different vendors[29].

Name	**Description**
Libvirt API	Orchestrate and manage KVM, QEMU, Xen, etc. API language bindings for several mainstream programming languages.
Xapi Project	Orchestrate and manage Xen. API language bindings for several mainstream programming languages.
vSphere	Manage and Orchestrate VMs through VMware vSphere. First-party SDK in form of vSphere SDK, as well as language bindings for several mainstream programming languages.

Table 4: VM Orchestators Language Support

The most interesting API in table-4 is Libvirt. Libvirt provides an API to manage KVM, QEMU, Xen, VMware ESX, and more [30], as well as several API bindings to most mainstream programming languages, making it an obvious candidate for an all-around VM manager. Libvirt is open-source and provides thorough documentation to the API. Because both Xapi Project and vSphere only provide support for their corresponding hypervisor, implementing support for another hypervisor would significantly alter the system. Libvirt on the other hand, provides support for these hypervisors, with the use

of custom URIs based on the hypervisor used, effectively simplifying the implementation needed to support multiple hypervisors.

Several other APIs and SDKs for other mainstream hypervisors, and the hypervisors already addressed, do exist. However, the majority of the APIs provide equal, if not less, functionality and interoperability, than the three APIs located in table-4.

5.3 Migration

The main functionality of the design is the migration of workloads. Each type of workload has a specific order of action, to perform a migration. These tools and techniques were all discussed in section-3.3. However, the described workloads are not a complete collection of what can run within a data center. The workloads which will be of focus are Docker containers, as well as VMs, due to them being relatively complex to migrate.

5.3.1 Containers

As described in section-3.4 the migration of a Docker container can be done in several ways, however, the technique of most interest is the save and load method. In combination with CRIU, this technique preserves both the file system as well as memory state of the container during the migration process. However, as discussed in section-5.2.1, Docker does not provide the feature of CRIU within most of their SDKs or APIs. To utilize the potential of CRIU together with the Docker SDK, the only option of implementation language is Golang. Golang is known for being simple, reliable, and effective, however, the obvious upside to using Golang is concurrency, which makes it a language well suited for writing server applications, which serves well as if multiple migrations were to be executed at once, or a lot of guests are operating together concurrently.

The GuestController component, as described in section-5.1, is the component that handles the migrations of the Docker containers. This part of the implementation will have to be written in Golang to utilize and automate the CRIU checkpoint feature to enable loss-less migrations.

As previously mentioned in section-3.4, a series of Docker commands described in listing-4, can be used to perform a migration. The first 6 lines of the listing are what needs to be executed on a source guest, and the last part needs to be executed on the destination guest. Due to this some form of communication between each guest need to be present. This can be enabled by simply setting up an HTTP server on

the guests, each with a registry of which other guests are on the network. Each guest can then communicate with each other, to perform a partial migration on the source guest, and perform the last part on the destination guest. The last part which needs to be addressed is storage. When saving a Docker container, as well as when creating a checkpoint, shared storage between the source and destination guest needs to be present. This can be either a NAS or SAN, providing a simple shared folder between the guests.

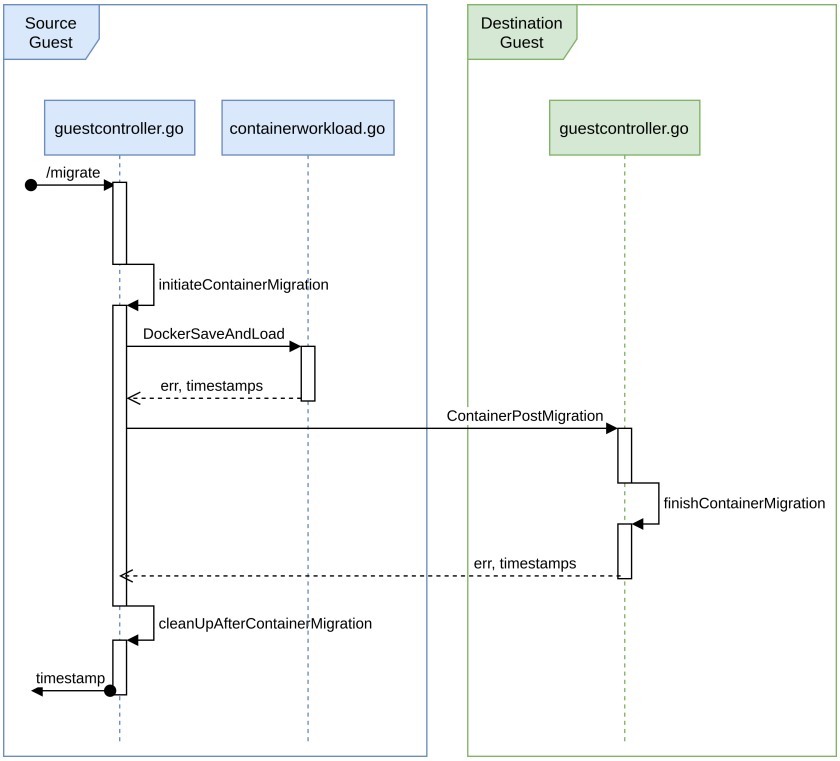


Figure 6: Sequence Diagram of Container Migration

During the migration process of a container, both the source and destination have to be involved. The sequence diagram illustrated in figure-6 shows this interaction. The source starts by preparing the migration by saving the container image on a shared storage device, specified by the host. When this is done, the source transfers the information about the workload to the destination, through a post-migration method. Then the destination continues the migration, loading the container through the specifications received by the source guest. Finally, the destination responds to the source, telling the source guest that the migration was successful.

5.3.2 Virtual Machines

The VM migration process varies greatly from hypervisor to hypervisor, however, the core principle is the same. A combination of the three phases of live migration as described in section-3.7.1, is used for the migration of VMs throughout hypervisors. The focus of migrating VMs will be on live migration and not cold migration due to the decreased downtime and the general similarity in implementation. The following discussed hypervisor migration techniques will all be functionally the same for cold and live migration.

The hypervisors of focus are VMware, Xen, together with the virtualization API Libvirt. This selection is based on both general popularity as well as accessibility. VMWare and Xen are both hypervisors that occupy the vast majority of the market share of hypervisors deployed in a data center environment as illustrated in figure-1. Libvirt provides control over the two aforementioned hypervisors, as well as a long list of others. This makes Libvirt a very attractive tool to implement, due to its interoperability with a long list of hypervisors.

5.3.3 VMware

As described before, VMware is the most popular solution for data center virtualization. However, VMware is the only selected hypervisor that is neither open-source nor free. With an entry cost of around 1200 USD[31] at the time of writing, this tool is simply not accessible through this project. However, the process of migration through VMware vSphere is done through the tool vMotion. This is all done through the user interface and is as simple as selecting a VM instance and selecting the migrate option. However, much like other VM migration techniques, some prerequisites need to be established. These prerequisites major are that the destination and host need to have a valid subscription to vMotion, that the host has access to the destination storage. The migration technique used by vMotion is what is previously described in section-3.7.2 as pre-copy. Although additional steps are performed within the migration process, the essential steps for performing the migration is pre-copy[32].

5.3.4 Xen

Xen is, just like VMware, a hypervisor that is readily used in the data center environment. However, Xen is open-source and has sparked a variety of tools which is used to

manage Xen. To orchestrate the Xen hypervisor, the vSphere counterpart to Xen called Xen Orchestra can be used, although this is not the only option. Just like orchestration, the list of available APIs is also vast. As described before in section-5.2.2, Xapi can be used, but this is not the only one. The migration of Xen VMs is done either through the orchestration software, APIs or through terminal commands. Migrating Xen directly from the hypervisor, has, just like VMware, prerequisites. Both the source and destination machine has to be running Xen. Shared storage in form of, fx a SAN, is also needed in some form to perform the migration. A simple command-line argument is needed to migrate a Xen VM to a destination and is described in listing-6

```
1 xl migrate <domain> <host>
```

Listing 6: Xen Migration through command-line

Migrating through Xen Orchestra is a simple drag and drop interface. Dragging a VM from one hypervisor to the other performs a live migration of the VM. This of course requires the same pre-requisites as the standalone Xen migration through the command line.

5.3.5 Libvirt

Libvirt is unlike VMware and Xen, not a hypervisor but a virtualization API used to manage a long list of hypervisors, including both VMware and Xen. This is desirable in a project like this, due to the interoperability of multiple hypervisors, without much additional implementation. Libvirt provides language bindings for multiple mainstream programming languages, and the prerequisites for using Libvirt is that the libvirt-dev package is installed, as a minimum on the machine which performs the Libvirt commands. Libvirt-dev is a bare-bones implementation of Libvirt and only supports a subset of hypervisors, in this case, not VMware. Because Libvirt is open source, and the migration techniques vary from hypervisor to hypervisor, the options of the migration can be specified to more than the two previous counterparts. Libvirt also provides thorough documentation of each of these options, including how exactly each migration is executed.

To migrate a VM through Libvirt a few options are available[33]:

- Hypervisor native transport is the process of migration through the used hypervisor. If fx, a Xen VM needs to be migrated to another Xen hypervisor, this method will be used.

- Libvirt tunneled transport, is just like the native transport, however, now the connection is tunneled through the Libvirt connection. This method is more capable of strong encryption, due to the encryption methods built into Libvirt. The downside is that extra data will be copied from Libvirt to the destination hypervisor, due to both the destination and source VM being transported between Libvirt and the hypervisor.

- Managed direct migration utilizes a manager Libvirt instance to facilitate the migration. The source and destination do not need to communicate with each other, only the manager. If migration is not successful, the source VM will be restarted, and another attempt at the migration can be facilitated.

- Managed peer-to-peer migration is like managed direct migration, however, in this instance, the manager does not communicate with the destination. The source receives the migration details and migrates directly to the source. This is potentially safer due to the destination only closing the VM when the migration is successful.

- Unmanaged direct migration is just like the hypervisor native transport, in that Libvirt does not control the migration process. The difference is that Libvirt merely facilitates and initiates the migration. This is beneficial in that if Libvirt crashes, the migration can still occur, due to the only responsible entity within the migration process being the individual hypervisors.

- Offline migration, is what is previously described as cold migration. An offline instance of a VM is moved from one hypervisor to another.

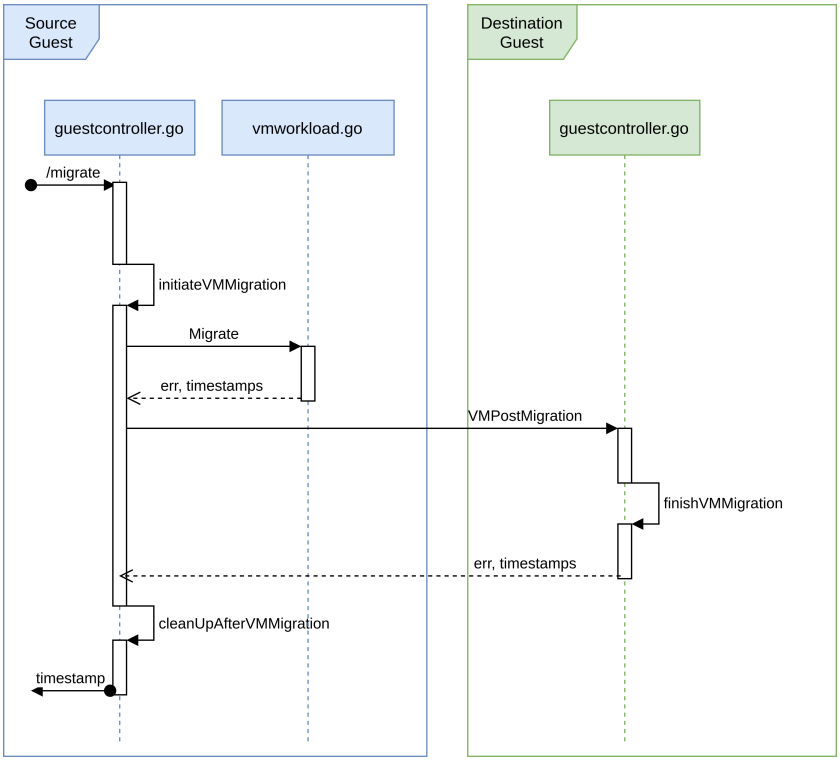


Figure 7: Sequence Diagram of VM Migration

Like the container migration process, when mitigating a VM, both the source and destination must be involved. The sequence diagram showing the VM migration interactions is illustrated in figure-7. The same three-step process is applied to the VM migration, as it was applied to container migration. First, the source starts the migration. Second, the destination is notified that the migration is occurring, and third, when the migration is successful the destination responds to the source to indicate that the migration was successful.

With all of the features of the solution designed, discussed, and illustrated, the foundation of the implementation is created. Each design decision within this chapter was based on the requirement specifications and will found the basis of the implementation which will be described in the following sections.

6 Implementation

This section covers the implementation of the developed solution, as well as important development decisions important to the development of the solution. The class-diagrams of the GuestController and the main Controller, are both available in the appendix-A.3 and appendix-19 respectively.

6.1 Github repository

`https://mortenabra.github.io/Masters-Datacenters/`

6.2 Client

6.2.1 Controller

As described in section-5.1, the controller is comprised of multiple subsystems, VMManager, GuestManager and EnergyDataFetcher. Each component of the controller will be described in further detail in the following sections.

6.2.2 Managers

The VMManager functions as a core object in the system which handles core functionalities such as migration calls, manual workload instantiation, and workload handling. As the GuestManager receives a JSON response from the GuestController, the data is parsed from a JSON object to a workload object. Whenever workloads are fetched and instantiated as workload objects, they are individually added to the corresponding Guest.

```
1 public void getWorkloadsFromGuests(VMManager vmManager) {
2         for (Guest g : guestList) {
3             if (g.isOnline()) {
4                 HttpClient client = HttpClient.newHttpClient();
5                 HttpRequest request = HttpRequest.newBuilder().uri(URI.
    create(g.getURL() + "/workloads")).build();
6 
7                 HttpResponse<String> response;
8                 try {
```

```
9                       response = client.send(request, HttpResponse.
    BodyHandlers.ofString());
10                      if (response.body().length() != 0) {
11                          Object obj = JsonParser.parseString(response.
    body());
12                          JsonObject jsonObject = (JsonObject) obj;
13
14                          JsonArray jsonArray = (JsonArray) jsonObject.
    get("Workloads");
15                          for (int i = 0; i < jsonArray.size(); i++) {
16                              Workload workload = vmManager.
    parseWorkloadObject((JsonObject) jsonArray.get(i));
17                              vmManager.getWorkloads().add(workload);
18                              g.getWorkloads().add(workload); // Add
    workload to guest.
19                          }
20                      }
21                  } catch (IOException | InterruptedException e) {
22                      System.out.println("Workloads not available: " +
    toString());
23                  }
24              }
25          }
26      }
```

Listing 7: GuestManager, getWorkloadsFromGuest()

Hereafter, the individual workloads are added to a complete list of workloads, by iterating each workload list within each Guest and thereby adding them to the complete list. Before each workload are added to any list however, they will need to be parsed to an applicable workload type. Hence a parseWorkloadObject method, which interprets the JSON data, and instantiates the correct object, based on its input parameters.

```
1 public Workload parseWorkloadObject(JsonObject jsonWorkload) {
2         String wl_name = (String) jsonWorkload.get("Identifier").
    getAsString();
3         String wl_ip = (String) jsonWorkload.get("AccessIP").
    getAsString();
4         int wl_port = (int) (long) jsonWorkload.get("AccessPort").
```

```
    getAsLong();
5       boolean wl_status = (boolean) jsonWorkload.get("Available").
    getAsBoolean();
6       boolean wl_autoMigration = (boolean) jsonWorkload.get("
    AutoMigrate").getAsBoolean();
7       String wl_sharedDir = (String) jsonWorkload.get("SharedDir").
    getAsString();
8       String wl_type = (String) jsonWorkload.get("Type").getAsString
    ();
9
10      // Determines the type of the workload
11      Workload.WorkloadType type;
12      switch (wl_type) {
13          case "Container":
14              type = WorkloadType.CONTAINER;
15              JsonObject containerProps = (JsonObject)jsonWorkload.
    get("Containerproperties");
16              String containerID = (String) containerProps.get("
    ContainerID").getAsString();
17              String containerImage = (String) containerProps.get("
    Image").getAsString();
18              boolean checkpoint = (boolean) containerProps.get("
    Checkpoint").getAsBoolean();
19              ContainerWorkload c_wl = new ContainerWorkload(wl_name,
     wl_ip, wl_port, wl_status, wl_autoMigration, wl_sharedDir, type,
    containerID, containerImage, checkpoint);
20              return c_wl;
21          case "VM":
22              type = WorkloadType.VM;
23              JsonObject vmProps = (JsonObject)jsonWorkload.get("
    VMProperties");
24              String domainName = (String) vmProps.get("DomainName").
    getAsString();
25              String connectionURI = (String) vmProps.get("
    ConnectionURI").getAsString();
26              VMWorkload vm_wl = new VMWorkload(wl_name, wl_ip,
    wl_port, wl_status, wl_autoMigration, wl_sharedDir, type,
    domainName, connectionURI);
```

```
27                  return vm_wl;
28              default:
29                  Workload wl = new Workload(wl_name, wl_ip, wl_port,
    wl_status, wl_autoMigration, wl_sharedDir, null);
30                  return wl;
31          }
32      }
```

Listing 8: VMManager, parseWorkloadObject()

As seen in listing-8, there are three workload types, Container, VM, and a default workload if neither applies.

6.2.3 GUI

At application startups, the managers are initialized with all GUI elements. Since the GUI needs to display all workloads and be able to view their details and migrate them, a JList is used. As workload objects need to be displayed in the JList, a DefaultListModel is used, which is an implementation of the ListModel interface. The DefaultListModel allows for the supply of an object list.

```
1  listModel = new DefaultListModel<>();
2      for (Workload workload : manager.getWorkloads()) {
3        listModel.addElement(workload);
4      }
5
6      vmList = new JList(listModel);
7      vmList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
8      vmList.setSelectedIndex(0);
9      vmList.addListSelectionListener(new ListSelectionListener() {
10       @Override
11       public void valueChanged(ListSelectionEvent e) {
12         if (!e.getValueIsAdjusting()) {
13           if (vmList.getSelectedIndex() != -1) {
14             Workload object = (Workload) vmList.getSelectedValue();
15             if (object instanceof Workload) {
16               selectedWorkload = object;
17               vmNameResult.setText(object.getWl_name());
18               vmIPResult.setText(object.getWl_ip());
```

```
19              vmStatusResult.setText(String.valueOf(object.isWl_status
    ()));
20              vmAutoMigrationSwitch.setSelected(object.
    isWl_autoMigration());
21            }
22          }
23        }
24      }
25
26    });
```

Listing 9: App, initialize()

After the GUI elements have been initialized, two TimerTasks will execute. The first TimerTask is responsible for continuously fetching workloads from Guests at a fixed rate. The second task continuously fetches energy pricing data at a fixed rate too. Both tasks are executed by a ScheduledExecutorService which initialize a ScheduledThreadPool of 10 threads.

```
1   private void setupWorkloads() {
2     int delay = 0;
3     int period = 5;
4     AtomicInteger workloadIteration = new AtomicInteger(0);
5     TimerTask updateWorkloadsTask = new TimerTask() {
6       @Override
7       public void run() {
8         workloadIteration.incrementAndGet();
9
10        guestManager.getWorkloadsFromGuests(manager);
11
12        checkListModelDuplicates(manager, initDone);
13
14      }
15    };
16    es.scheduleAtFixedRate(updateWorkloadsTask, delay, period, TimeUnit
    .SECONDS);
17    try {
18      Thread.sleep(delay + period);
19    } catch (InterruptedException e) {
```

```
20        e.printStackTrace();
21      }
22    }
```

Listing 10: App, setupWorkloads()

As depicted in listing-10, a initial delay and period is defined. The delay defines a time from thread start until TimerTask starts the encapsulated methods. The period defines the time between each iteration of the task. During the task, the ListModel elements will be replaced multiple times, therefore we must ensure no duplicate elements will be in the list. The workloads are stored in a HashSet in the VMManager, as a Hashset do not allow for duplicate elements, however a ListModel does, therefore its import to remove elements already in the ListModel when replacing.

```
1  public void checkListModelDuplicates(VMManager manager, boolean done) {
2      if (done && listModel != null) {
3        // Add workloads that are not in the listModel
4        for (Workload workload : manager.getWorkloads()) {
5          if (!listModel.contains(workload)) {
6            listModel.addElement(workload);
7          }
8        }
9
10       for (int i = 0; i < listModel.size(); i++) {
11         if (!manager.getWorkloads().contains(listModel.get(i))) {
12           listModel.removeElementAt(i);
13         }
14       }
15       vmList.setModel(listModel);
16     }
17   }
```

Listing 11: App, checkListModelDuplicates()

6.2.4 EnergyDataFetcher

To allow for workloads to migrate based upon electrical pricing and therefore utilizing load-saving techniques, the system must be able to collect electrical pricing. The pricing model used for determining if the workload needs to be migrated is based on RTP (Real-

time-pricing). The dataset is updated a day in advance, therefore allowing the system to determine a threshold based on past data. Since the electrical data is delivered as JSON, the EnergyDataFetcher object parses the data to a befitting model.

```
1    private void getEnergiData(int days) {
2
3        int responseCode;
4        try {
5            this.url = new URL(fp.getAPIURL(days));
6            this.urlConnection = (HttpURLConnection) url.openConnection
     ();
7            urlConnection.setRequestMethod("GET");
8            urlConnection.setRequestProperty("Content-length", "0");
9            urlConnection.setUseCaches(true);
10           urlConnection.setAllowUserInteraction(false);
11           urlConnection.connect();
12           responseCode = urlConnection.getResponseCode();
13
14           switch (responseCode) {
15               case 200:
16                   bufferedReader = new BufferedReader(new
     InputStreamReader(urlConnection.getInputStream()));
17                   convertJsonToObjects(readChaString(bufferedReader))
     ;
18                   setThreshold(32);
19                   break;
20               case 400:
21                   System.out.println("400 - Bad reqeust! Check syntax
     ");
22                   break;
23               case 404:
24                   System.out.println("404 - Not Found!");
25
26               default:
27                   break;
28           }
29
30       } catch (Exception e) {
```

```
31              e.printStackTrace();
32          }
33      }
```

Listing 12: EnergyDataFetcher, getEnergiData()

The data is then stored in a list which is used to calculate a threshold for migration. However, the threshold can also be set by the operator in the GUI. As the system is launched, the EnergyDataFetcher is instantiated and run in a separate thread, and executed at a fixed rate, similarly to listing-10.

6.3 Guest

To allow for migration functionality on each guest running within the DR system, each guest has a GuestController tied to it. This controller is written in Golang and is designed to be as autonomous as possible, enabling the main Controller software to orchestrate and monitor each guest. Throughout the following sections, the overall functionality of the GuestController will be described in detail.

6.3.1 GuestController

The GuestController is what orchestrates all migrations within the system. It consists of the main class which is called GuestController. The functionality of the software can be divided into two main parts and one which facilitates them. The first two are the migration implementations of Docker and Libvirt, and the facilitator is the HTTP server setup. All three elements will be discussed further in their respective sections.

6.3.2 HTTP Server

The backbone of communication between entities in the system is the HTTP servers of the GuestController. The Controller has to be able to send requests to the guest, telling it to migrate a workload, get information about the workloads running on the guest, access logs, and gather other relevant information about the guest. This is done by a simple HTTP server running on the GuestController. Within the setupServer() method of guestcontroller.go, each handler function of each URL for the HTTP server is initialized, as depicted in listing-13

```
1 // Setup http listener
```

```
2  func setupServer(serverIP string, serverPort string) error {
3    drlogger.DRLog(drlogger.INFO, "Setting up handlers")
4    http.HandleFunc("/guest", guestHandler)
5    http.HandleFunc("/guest/workloads", guestWorkloadsHandler)
6    http.HandleFunc("/workloads", workloadsHandler)
7    http.HandleFunc("/migrate", migrateHandler)
8    http.HandleFunc("/transfer", transferHandler)
9    http.HandleFunc("/logs", logsHandler)
10
11   logInfo("Listening on " + serverIP + ":" + serverPort)
12   err := http.ListenAndServe(serverIP+":"+serverPort, nil)
13
14   return err
15 }
```

Listing 13: GuestController, setupServer()

Within the method, each of the HTTP server URLs is initialized, each with its unique functionality.

- \guest is used to get current information about the current guest. The request only supports GET, and when a GET request is handled by the guestHandler, the current guestInformation of the GuestController is returned as a JSON string. The guestInformation is the Guest object held by the guestcontroller and holds the current IP, Port, shared storage location, and Libvirt URI of the guest. This can be used to either verify the information about a guest from the Controller or to check whether or not a guest is online by pinging this address.

- \guest\workloads return the current workload information about the guest. This handler uses the Docker SDK and the Libvirt API to retrieve information about active and inactive containers and running or inactive VMs running on the guest. This functionality is used to retrieve information about workloads that can be introduced to the DR system, without accessing the guest directly. Just like the guestHandler, this handler does not support other requests than GET.

- \workloads hold the current workloads initiated in the DR system. This handler supports both GET and POST requests. To initiate a workload in the system, a simple POST request with a workload JSON object has the be posted to the

system. The template for the workload object can be seen in listing-14. When a post request with this structure, the JSON object is stored in a JSON array which is locally stored on the guests file system. This JSON array is the backbone of information throughout the guestcontroller, due to it holding all the relevant information about each workload within the system. Whenever a GET request is requested, the contents of the JSON array are returned as a JSON string. This can be used to get insight into what workloads are running on the system.

- \migrate is where the functionality of the system starts to come together. The migrateHandler supports only POST requests. The functionality of this handler is to, as the name implies, handle migrations. To migrate a workload in the DR system, a POST request is handled by the system. An example of the JSON structure of the POST request is illustrated in listing-15. The Identifier parameter specifies the identifier of the workload which will be subject to migration, and the Guest parameter specifies the destination guest information. When the migrateHandler retrieves this JSON information the migration process starts. The workflow of this function is further described in section-6.3.4 and section-6.3.5 for container and VM migration respectively.

- \transfer is a handler for internal migrations between guests, which will be described further together with the migrateHandler in section-6.3.4 and section-6.3.5.

- \logs provide the functionality of logging the state of each guest. Every internal log, warning, or error is written to a logs file, which is then returned when the logsHandler receives a GET request. This enables system monitoring of the guests without accessing the guests directly. Unless a fatal error that breaks the system occurs, the errors can be monitored through the logs.

```
{
    "Identifier": string,
    "AccessIP": string,
    "AccessPort": string,
    "AutoMigrate": boolean,
    "Available": boolean,
    "SharedDir": string,
    "Type": "Container" | "VM",
```

```
    "Containerproperties": {
        "ContainerID": string,
        "Image": string,
        "Checkpoint": boolean,
        "NetworkSettings": object,
        "CheckpointID": string
    }
    "VMProperties": {
        "DomainName": string,
        "ConnectionURI": string
    }
}
```

Listing 14: GuestController, JSON Structure Workload

```
{
    "Identifier": string,
    "TargetGuest": {
        "Ip": string,
        "Port": string,
        "StoragePath": string,
        "LibvirtURI": string
    }
}
```

Listing 15: JSON Structure Migrate, guestcontroller

With the HTTPServer described the main functionality of the guestcontroller is left to be described. The migration is divided into two parts, container- and VM migration. Although these two functions have a lot in common and are structurally identical, their functionality is very different.

The structure of the guestcontroller is constructed as it is to accommodate new implementations of workload types. If fx, Docker Swarm, or Kubernetes need to be implemented into the system, a new set of functions to migrate, as well as an implementation of their properties into the system has to be constructed. The migrateHandler checks which workload needs to be migrated, and therefore a new check for the new workload also has to be implemented. With these changes, a new workload type can be added to the system, with all of the other functions being dynamic and can therefore

accommodate the new workload type.

6.3.3 Migration

Whenever a migration request is done through the migrateHandler of the guestcontroller, the type of workload is first identified through the JSON object which structure is described in listing-15. To get the workload type, the identifier is used to locate the workload in question, and then the identified workload type is used to identify which migration process is used.

6.3.4 Container Migration

The container migration functionality of the guestcontroller is supplied by the Docker SDK. The Docker SDK for Golang provides functionality such as; create, remove and monitor containers, but also experimental features like checkpoint/restore which are needed for the container migration process. The implementation of the migration function can be seen in listing-16 with the sequence of operations shown in listing-6.

```
1  // Init docker environment
2  ctx, cli, err := initDocker()
3  ...
4  ...
5  newImage, err := cli.ContainerCommit(ctx, wl.Properties.ContainerID,
       types.ContainerCommitOptions{})
6  if err != nil {
7    return 0, 0, 0, err
8  }
9
10 wl.Properties.Image = newImage.ID
11
12 resp, err := cli.ImageSave(ctx, []string{newImage.ID})
13 if err != nil {
14   return 0, 0, 0, err
15 }
16 defer resp.Close()
17 ...
18 ...
19 // Write resp to file
```

```
20 outFile, err := os.Create(wl.SharedDir + "/" + wl.Properties.
       ContainerID + ".tar.gz")
21 if err != nil {
22   ...
23 }
24 ...
25 ...
26 if restore {
27   err = cli.CheckpointCreate(ctx, wl.Properties.ContainerID, types.
       CheckpointCreateOptions{
28     CheckpointID:  wl.Properties.CheckpointID,
29     CheckpointDir: "",
30   })
31 ...
32 ...
33 defer outFile.Close()
34 _, err = io.Copy(outFile, resp)
35 if err != nil {
36   ...
37 }
```

Listing 16: GuestController, DockerSaveAndStoreCheckpoint()

First, the Docker environment is initialized. This returns a client cli object and a context ctx object. These objects are needed to access the Docker environment. Then the Docker image is committed, then saved using the ImageSave method. This returns a resp object which is an io.ReadCloser. An io.ReadCloser is an interface for the basic read and writes functions of a file. This file represents the output of the ImageSave function, and therefore needs to be saved to be transferred to the destination guest. Next, a new tar.gz file is created, which is used to store the output. If the migration requires a checkpoint, which is specified in the workload JSON specification, a checkpoint is created with the checkpoint ID also specified in the JSON specifications. At last, the output tar.gz file is written to with the resp object from the ImageSave function, effectively writing the file information.

When the DockerSaveAndStoreCheckpoint method is completed, the migration is essentially done on the source guest. The next step is to inform the destination guest that the migration is ready to be finished at the destination. The method ContainerPostMi-

gration is then called which again uses the HTTP server to facilitate communication. This is where the \transfer comes in. The workload object is posted to the destination guest through the \transfer, which triggers the transferHandler.

The transferHandler receives the workload JSON from the source and adds this to the locally stored workload JSON array. This works just like a POST request to the workloadsHandler but with more functionality. When the workload is added to the locally stored JSON array, the method for finishing container migration called finishContainerMigration is called. This method contains the functionality of migration on the destination end as discussed in section-3.4, i.e. use Docker Load to create the container on the destination, restore the checkpoint, and start the container. The functionality is illustrated in listing-17.

```
1  // Init docker environment
2  ctx, cli, err := initDocker()
3  ...
4  ...
5  tarFile, err := os.Open(wl.SharedDir + "/" + wl.Properties.ContainerID
       + ".tar.gz")
6  if err != nil {
7    ...
8  }
9
10 resp, err := cli.ImageLoad(ctx, tarFile, true)
11 if err != nil {
12   ...
13 }
14 defer resp.Body.Close()
15
16 ...
17 ...
18 out, err := cli.ContainerCreate(
19   ctx,
20   &container.Config{Image: wl.Properties.Image},
21   nil,
22   &network.NetworkingConfig{EndpointsConfig: wl.Properties.
       NetworkSettings},
23   nil,
```

```
24    wl.Properties.ContainerID)
25  if err != nil {
26    ...
27  }
28  ...
29  if restore {
30    err := cli.ContainerStart(ctx, out.ID, types.ContainerStartOptions{
31      CheckpointID:  wl.Properties.CheckpointID,
32      CheckpointDir: "",
33    })
34    if err != nil {
35      ...
36    }
37  } else {
38    err := cli.ContainerStart(ctx, out.ID, types.ContainerStartOptions{})
39    if err != nil {
40      ...
41    }
42  }
```

Listing 17: GuestController, DockerLoadAndStartContainer()

To start with the Docker environment is initialized just like the source guest. Then the tar.gz file which was the output of the source migration is opened and used as an input in the Docker Load function. This creates an image based on the image which was saved by the source guest. Even though the image is migrated, the container still needs to be created and started, which is the function of the last part of the code. The container is created, based on the configurations of the workload JSON, which was retrieved from the source guest, and then started based on a checkpoint if a checkpoint was created. At last, the container is started.

Now that the migration essentially is done, the source workload is copied to the destination and started on the destination, the workload needs to be added to the new system, and the source needs to be cleaned up. First the method AddContainerWorkload is called on the guest, which adds the workload information sent by the source is added to the destination guest's workload JSON array. When this is done, the original POST request sent by the source guest is returned with a non-error response. This indicates to the source that the migration was successful and that the cleanup can start on the

source guest. For this, the method cleanUpAfterContainerMigration is called which is described in listing-18.

```
1 func cleanUpAfterContainerMigration(container containerworkload.
    ContainerWorkload) error {
2   err := jsonmanager.RemoveContainerFromWorkloadFile(container)
3   if err != nil {
4     return err
5   }
6   err = container.DockeRemoveContainer()
7   return err
8 }
```

Listing 18: GuestController, cleanUpAfterContainerMigration()

This method simply removes the workload from the workloads array contained on the guest and removes the Docker container from the Docker environment. Additional functionality like cleaning up the file system could have been implemented, however, this will be discussed further in the discussion in section-8

6.3.5 Virtual Machine Migration

The migration of VMs is almost entirely handled by the virtualization API Libvirt. Libvirt is an ideal implementation, due to its interoperability with mainstream hypervisors. Because the container migration has a strict requirement of being implemented in Golang to support CRIUs checkpoint/restore functionality, and because Libvirt has language bindings for most mainstream languages, including Golang, the implementation of VM migration is also done in Golang.

Even though the steps of migration are different compared to the Docker container migration implementation, the structure of the methods is close to identical. The sequence diagram of the VM migration process is illustrated in figure-7, and it shows how similar the two implementations are. The main difference however is the internal calls within the guestcontroller on both the source and destination guest.

First, just like the container migration, the workload which is about to be migrated is identified and logged on the source system. If this workload is an instance of the VMWorkload class, the method initiateVMMigration is called. This effectively starts the migration of the workload from the source guest to the destination guest. The first method called within initiateVMMigration is Migrate.

```
func (wl VMWorkload) Migrate(destinationURI string) error {
  conn, err := libvirt.NewConnect("qemu:///system")
  if err != nil {
    ...
  }
  defer conn.Close()

  doms, err := conn.ListAllDomains(libvirt.CONNECT_LIST_DOMAINS_ACTIVE)
  if err != nil {
    ...
  }

  fmt.Printf("%d running domains:\n", len(doms))
  for _, dom := range doms {
    dname, err := dom.GetName()
    if err != nil {
      ...
    }

    if dname == wl.Identifier {
      err = dom.MigrateToURI3(destinationURI, &libvirt.
   DomainMigrateParameters{}, libvirt.MIGRATE_ABORT_ON_ERROR)
      if err != nil {
        ...
      }
    }
  }
  return err
}

return err
```

Listing 19: VMWorkload, Migrate()

The Migrate method described in listing-19 uses the VMWorkload object to gather the relevant information ended to perform the migration, as well as a destination URI which is used to determine the destination guests URI. First, a connection to the local Libvirt instance is made. In this case, the QEMU hypervisor is used, however, this

can be changed to suit whatever hypervisor Libvirt supports. When the connection is established, all domains are iterated though to determine which domain is the one that needs to be migrated. This is determined by matching the domain name with the identifier of the workload. This brings issues if either identifier has identical names, or if, fx, a container workload has the same name as a VM workload. A more secure implementation would be to add yet another field to the VM workload, which handles the domain name, however, this would not eliminate the case of duplicate domain names, due to the domain name not being a unique identifier. A further discussion of this topic will be covered in section-9.

After the workload is identified through the Libvirt API, the MigrateToURI method is called which initiates the actual migration through Libvirt. If the migration is successful, the migration is essentially done.

After this, the guest receives an HTTP request on the \transfer, which reaches the transferHandler. This identifies that the workload is a VMWorkload, and calls the finishVMMigration method. This method simply adds the workload to the workload JSON array, to make it visible to the rest of the DR system.

When the workload is successfully added to the system, the cleanup process can start on the source guest. This simply removes the workload from the workloads JSON array.

The implementation of the container workload migration and the VM workload migration is processed through essentially the same steps; initiate, finish, and cleanup. Initiate performs the actual migration on the source. Finish processes the migration on the destination end and adding the workload to the destination controller, and cleanup cleans the system through removal of workloads that are not relevant anymore. This three-step system acts as a framework for migration through the DR system. With the potential implementation of further workloads, new initiate, finish, and cleanup methods need to be implemented, as well as accommodating workload objects to store the information needed to perform the migrations.

7 Results

With the implementation done, the system is ready to be evaluated based on the results produced by the system.

7.1 Test Strategy

To test the requirements of the solution, first, a test strategy has to be established. Each of the requirements from table-1 and 2 has to be evaluated during these tests, however, before the evaluation is conducted the test strategy has to be established.

7.1.1 Data center setup

To emulate a data center environment without having direct access to the resources of a data center, a few compromises were made. These compromises do not impact the functionality of the setup but inhibit the potential speed and performance of a traditional data center. The results of tests can therefore be scaled up to adhere to the level of data center performance, to achieve close to realistic experiments, with substantially less performance. In the experimental setup, two sets of tests were done on two different computers, each with different specifications. Each computer will emulate a data center. The purpose of having two computers is to replicate the experiments and compare the results, thereby, being able to scale the performance of the results to the performance of a traditional data center.

7.1.2 Server

To simulate multiple servers within the setup, a series of VMs were created to each emulate a server in a data center. This VM acts as its own environment and is isolated from the rest of the servers. Each computer has two VMs hosted, both with the same configuration. Each VM has two cores, as well as 1GB of RAM allocated, running a Linux-based operating system. This is mainly because CRIU is only available in the Linux environments, as well as Linux is the dominant operating system within server environments[34].

7.1.3 Storage

To provide shared storage to the VMs, a simple NAS setup is used. The VM host will act as a NAS server, and provide storage to the VMs through NFS. NFS is a simple implementation of a shared file system. The host contains a folder which it exposes to the VMs, which enables them to access the folder through a very simple process. This folder is then shared between all three systems, the two VMs, and the host, enabling the systems to share files. Another storage option is to set up a SAN with an iSCSI target. iSCSI is a popular protocol within SAN networks, and is just like NFS, easy to set up. The SAN server can either be executed, like the NAS solution, from the host, or through an external server.

7.1.4 Workloads

The workloads running on the VMs include a variety of containers and VMs. The workloads will need to vary in both functionality and size of their file system, to evaluate the system on varying workloads.

7.1.5 Course of action

The functional requirements regarding the migration of workloads between two servers, both sharing the same storage (F01 and F02), as well as at different geographical locations (F03 and F04), will be executed through the implemented solution. To test if the migration was successful, the workloads will be monitored both before and after the migration, to ensure that the workloads state is preserved (F05). The type of DR system used i.e. price-based system (F06) does not have to be tested for its usage but tested for its functionality. The input energy data will be analyzed both before and after processing of the implementation. The last three functional requirements (F07, F08, and F09) are all functionality of the user interface (UI) and can be tested either through monitoring the logs or accessing the UI after the implementation.

The non-functional requirements require a thorough analysis of the implementation. By documenting the downtime of each workload and analyzing the processing done by the implementation, the downtime can be evaluated, and determined if it is acceptable (N01). The input data intervals (N02) have to be of reasonable length, and because this is done in the implementation, the length of the intervals can be discussed and analyzed during the experimentation. To test the UI (N03 and N04), a short discussion of the UI

will be performed. Because user experience is deeply subjective, the UI is evaluated by functionality and not by aesthetics.

Initially, the system has to be loaded with workloads ready for migration. Whenever the workloads are initiated and added to the system, migration of each workload is performed from the source system to the destination system. When the migration process is done, a second migration process is performed to migrate from the destination system back to the source system. Finally, the attributes of the systems are documented and analyzed to evaluate the system performance.

To do the experimental tests the following actions will be executed:

1. Initiate workloads on a guest VM.
2. Workloads are added to the system.
3. Migration of workloads from one guest VM to another.
    - Migration time is documented
    - State preservation is documented
4. Downtime is analyzed to determine if it is acceptable.
5. UI is evaluated
6. Price-based system is evaluated

With this course of action, all of the aforementioned requirements will be either discussed or evaluated. If a requirement is not included in the evaluation, a solution to the requirement will be discussed in either Discussion of the results in section-8, or Further Work in section-9.

7.1.6 Experimental Setup

As described in section-7.1, the experiments are executed on two computers, each running two VMs. The utility of two computers is to get a broader set of results based on different hardware. The goal is to differentiate between the two computers, based on their performance in both processing and storage speeds. If this task can be completed, a direct comparison between the performance and output of the migration process can

be done. Each system will be differentiated by hardware and performance tested to determine the difference between the systems.

To emulate additional systems on each setup a series of VMs will be added to each computer. Each computer will run two VMs, each emulating a server in a data center. The plan is to migrate workloads between the VMs and record the output of every migration.

7.1.7 Hardware

As the server hardware configuration in data centers tends to vary, depending on the usage of each machine, it was important to diversify the hardware used in the experiments. The configuration of each laptop is comprised of at least 4-cores and 8GB of RAM, however, in data centers, the number of CPU cores can range upwards of 128-cores.

The specifications of the two computers are described in table-7. As seen in table-7,

Model	Lenovo 720S-14IKB
CPU	Intel i5-8250U 1.6GHz 4-Core
RAM	8GB DDR4
Storage	256 GB SATA3 SSD

Table 5: Laptop #1

Model	Huawei MateBook 13
CPU	Intel i5-8265U 1.6GHz 8-Core
RAM	8GB DDR4
Storage	256 GB PCIe SSD

Table 6: Laptop #2

Table 7: Hardware specifications of the two test setups.

the specifications of each laptop are quite similar, however, the SSDs (solid-state drives) are is the variation that lies between the two setups. The storage connectivity interface used by the *Huawei* laptop compared to a SATA interface tends to be almost 2.5x-3x the speed, which will impact the results[35]. This is confirmed by testing each laptop's read and write speeds. The test is conducted by reading and writing a single large file in a folder, simulating the read and write functions used in this thesis. When migrating a workload typically, a large file has to be saved and loaded to a filesystem, hence the one large file format. The speeds of the two systems are displayed in table-8

The ratio of reads and writes exceeds the 2.5x-3x speeds expected. Laptop #2 read speeds are 3.7x faster, while the write speeds are 3.6x faster.

Setup Name	Read Speed (MB/s	Write Speed (MB/s)
Laptop #1	241	115
Laptop #2	905	417

Table 8: Laptops benchmark read and write speeds.

7.1.8 Software

The setup for each VM within the system is close to identical. To perform the migrations, and facilitate the necessary controllers a list of software is required to be installed on each system.

In terms of the operating system hosted on each VM, the choices are very limited. To access the packages needed, without building every tool from source, Debian 11 Bullseye was chosen. Linux operating systems are a necessity, due to the compatibility with CRIU. The specific operating system requirements for CRIU are however a bit vague. Through testing, the newer versions of Ubuntu did not support CRIU, due to dependency issues. However, CRIU experienced no issues running on Debian. At the time of writing, the latest stable version of Debian is version 10, Buster. However, due to the desired version of Libvirt not being available in version 10, the testing branch of Debian was used instead. Debian 11 currently supports every tool needed to facilitate the migration process of both VMs and Docker containers.

Docker Engine has to be installed on the guest as well. This process is fairly simple and is without any modifications to the official guide to installing Docker Engine on Debian[36].

To enable the functionality of CRIU on the VM, experimental features have to be enabled. This is done by modifying the experimental flag in the file daemon.json normally located in /etc/docker. This flag enables the Docker experimental features, however, CRIU is not a part of the Docker installation and has to be installed separately.

To install CRIU on the guest, simply clone the GitHub repository[37] and build the tool using Make. A guide to this is located within the INSTALL.md file also located in the CRIU GitHub repository. When the installation process of CRIU is done, the requirements for Docker migration are essentially done.

Libvirt is used to facilitate the migration of VMs and is needed to execute the con-

troller for the guest. The installation process, however, is very simple, and as discussed before, Debian 11 Bullseye was used due to its native support of Libvirt. Libvirt can therefore be installed directly from the default package manager. Everything needed is contained in the libvirt0 package, which can be installed through apt-get.

To facilitate migrations through Libvirt, a couple of additional prerequisites need to be addressed. To migrate a VM through Libvirt, the two computers, in which the migration happens, have to have shared storage. This is done through Network File System (NFS). NFS is relatively simple to set up and only requires that the host which shares the filesystem has access to the destination, and vise versa. A guide to setup NFS on two systems can be found here[38]. The NFS setup can also be used to share storage between the VMs for container migration.

7.1.9 System Setup

To set up the system for experimentation, some prerequisites have to be made.

- Workloads have to be added to the system. The workloads are the entities that will be tested throughout the experimentation.
- Shared storage between source and destination has to be constructed. This will be used to share files between source and destination.
- Each guest participating in the system has to be added to the master controller by their IP and port, which is specified by each guest.

The choice of workloads for migration is done by selecting a list of popular Docker images from Docker Hub, with varying size and functionality. This is to represent the largest amount of use-cases in which containers like this are a part. Each container's image size is logged in megabytes (MB), varying from 1.3 MB up to 936 MB. A list of the containers used in the migration process is described in table-9.

With the selection of images done, each image has to be containerized. This is done by starting a Docker container based on the image. An example of this is done in listing-20.

```
1 docker run -d -i -t --name <NAME> <IMAGE>  /bin/sh
```

Listing 20: Start container from image, Docker

ID	Image Name	Size (MB)
IM1	Busybox	1.23
IM2	Alpine	5.61
IM3	Ubuntu	72.9
IM4	Redis	105
IM5	Mongo	449
IM6	Wordpress	550
IM7	MySQL	556
IM8	Python3	885
IM9	Node	936

Table 9: List of images used for experimentation

The command starts a container based on an image and then is given a name. The -d option ensures that the container is created in detached mode. The -i and -t option allows the container to be run in interactive mode, allowing interaction with /bin/sh. The last parameter /bin/sh opens up a shell within the container, effectively never stopping the container. The key idea is that to do live migrations the container has to be active while the migration is happening. If the container was shut down, the migration process would be considered cold migration. Functionality like checkpoint/restore does not work on inactive containers, due to there being no active memory to perform a checkpoint of.

For the VMs, a disk image needs to be downloaded and installed as a VM through the hypervisor. The hypervisor used for this evaluation is Virsh, which is a simple hypervisor for managing KVM and QEMU VMs. The choice of VMs was picked based on popular Linux distributions. A complete list of the VMs used throughout the evaluation can be found at table-10.

As all dependencies are installed it is important to set up the NFS server and the NFS clients. This enables a shared storage folder between clients and the server, from which all clients can extract their workloads. The NFS server must allow all clients through its firewall, declare a storage environment, and declare all client IP addresses within the guest JSON. This allows the clients to have access to the NFS shared storage and being recognized within the DR system.

This is done by adding their information within the guests.json file located in Master-

ID	Operating System	Size (GB)
VM1	Ubuntu 20.04	1
VM2	Oracle Linux 6	1
VM3	CentOS 8	20
VM4	Alpine 3.13	0.7

Table 10: List of VMs used for experimentation

Controller/drenerginetMaster/mastercontroller/src/main/java/mastercontroller/Guests folder from the source directory. The guests.json file is a JSON file that contains a JSON array called Guests. Each guest object within this file should follow the structure as depicted in listing-21. Each guest within the system has to be added to this file, to make them visible to the controller.

```
{
    "Ip": string,
    "Port": string
}
```

Listing 21: Guest Structure, guest.json

When the guests are added to the system the guest.json file should look something like what is depicted in listing-22.

```
{
    "Guests": [
        {
            "Ip": "192.168.122.162",
            "Port": "8080"
        },
        {
            "Ip": "192.168.122.109",
            "Port": "8080"
        }
    ]
}
```

Listing 22: Guest Example, guest.json

Now that every prerequisite is done, the setup can be finalized through the UI. As seen in figure-8 the left pane which contains the workloads, is empty. This is because although the workloads are running on the guest, they are not yet added to the DR system.

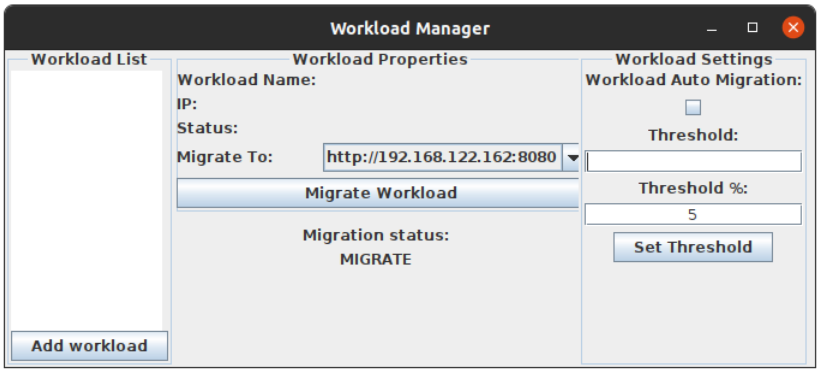


Figure 8: Initial UI

To add workloads to the system, the "Add Workload" button located under the workload list has to be used to add each workload. When the button is pressed a second window will appear which allows the operator to write the JSON specifications of said workload. The specifications of the workload need to be written in a specific format, which follows the workload JSON object described in listing-14. If the workload is a container, the VMProperties field is omitted, and if the workload is a VM, the Containerproperties field is omitted. When specifying a workload in JSON format, not all fields are necessary. AccessIP and AccessPort both are the only functionality of containers that expose a port, fx, a Python container running a flask server. NetworkSettings, as well as CheckpointID, are both fields which are used when guest communicate with each other during the transfer HTTP request, and therefore, there is no need to specify these fields either.

When the JSON specification fits the workload running on a specific guest, the "OK" button can be pressed, and a new window appears which enables the operator to specify which guest the workload is running on.

This is done for each workload running on the system, however, this will only be done once throughout the lifecycle of the software. When the workload information is specified through the UI, the workload specification is sent to the guest and stored there locally. When the software starts up again, each guest is prompted to send which workloads are

running on their system, which will then return the newly added workloads.

With every workload added to the system, the system is ready to be evaluated. Each workload can now be migrated by selecting the workload in the workload list, selecting the destination guest of the migration, and pressing the Migrate Workload button.

7.2 Evaluation

With the system set up and the workloads initialized, the system is ready to be evaluated. As mentioned in section-7.1, a list of actions to perform the evaluation has been made. The first two steps; Initiate workloads on a guest VM, and Workloads are added to the system, are already done and described in section-7.1.9. The next step of the evaluation is to perform the migrations from the source to the destination.

The process of migration through the UI is simple. To migrate a workload, the workload is selected in the left pane, then the destination IP and Port is selected in the drop-down box, and at last the migrate button is pressed to perform the migration.

The migration process will be duplicated on both computers described in the experimental setup in section-7.1.6, where the results will be evaluated through a comparison.

To evaluate the migration duration, each of the underlying implementation methods was logged into four categories.

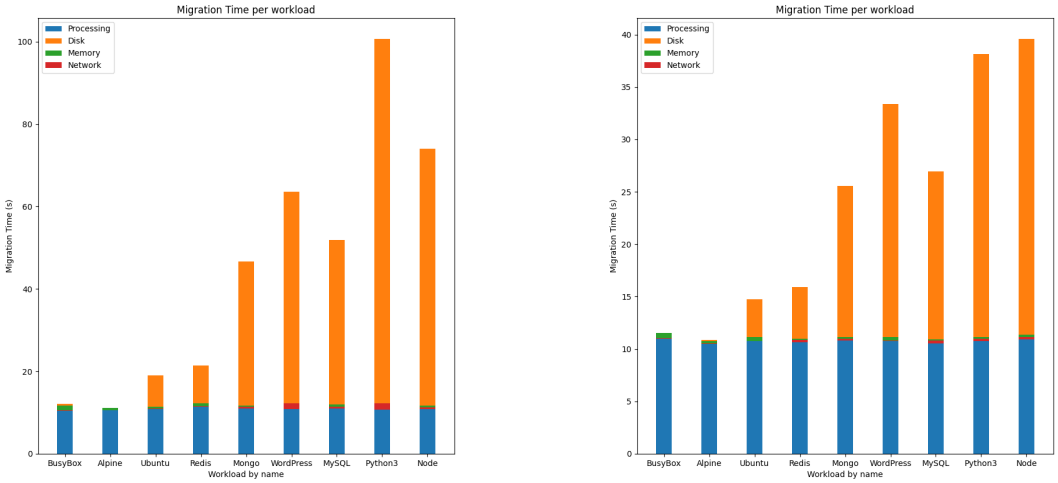
- Processing
- Disk
- Memory
- Network

With each workload divided into categorized durations of tasks, the migration process can be evaluated.

7.2.1 Container Migration Evaluation

The data sets is listed in full in appendix-A.1 and appendix-A.2. To start with a comparison of the different workloads and their migration duration can be done. Each workload has the aforementioned four migration duration; processing, disk, memory, and network. As seen in figure-9, the workload duration is primarily processing and disk tasks. One

observation within the data is that the processing seems to be relatively constant both between images and setups. Both have a duration of about 10 seconds. This anomaly will be discussed further in the Discussion section-8.

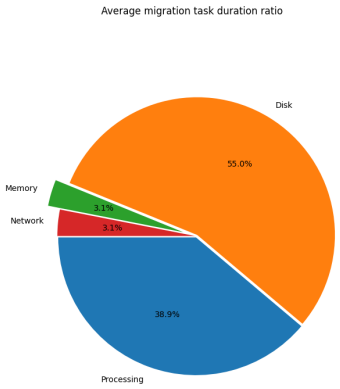


(a) Laptop #1

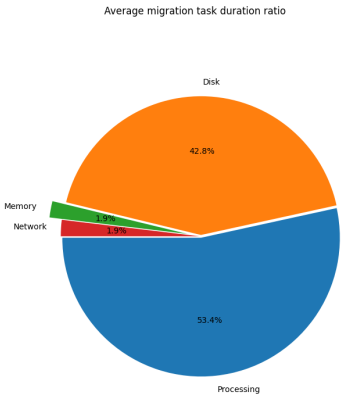
(b) Laptop #2

Figure 9: Migration duration based on workload. Each migration duration is separated into categories, and sorted based on image size.

To evaluate the duration of each category and analyze the impacted core elements on a machine, figure-10 was created. This shows that indeed the majority is both processing and disk tasks, with both Network and Memory being relatively similar. The network task is expected to be of a low value, due to the only networking calls within the system being insubstantial HTTP requests sent between destination and host. However, the memory task which saves the containers memory state to disk, is either asynchronous with the system, due to Dockers implementation of the checkpoint/restore functionality, or that the size of the memory transfer is just smaller than expected, and therefore results in a faster memory migration duration.



(a) Laptop #1

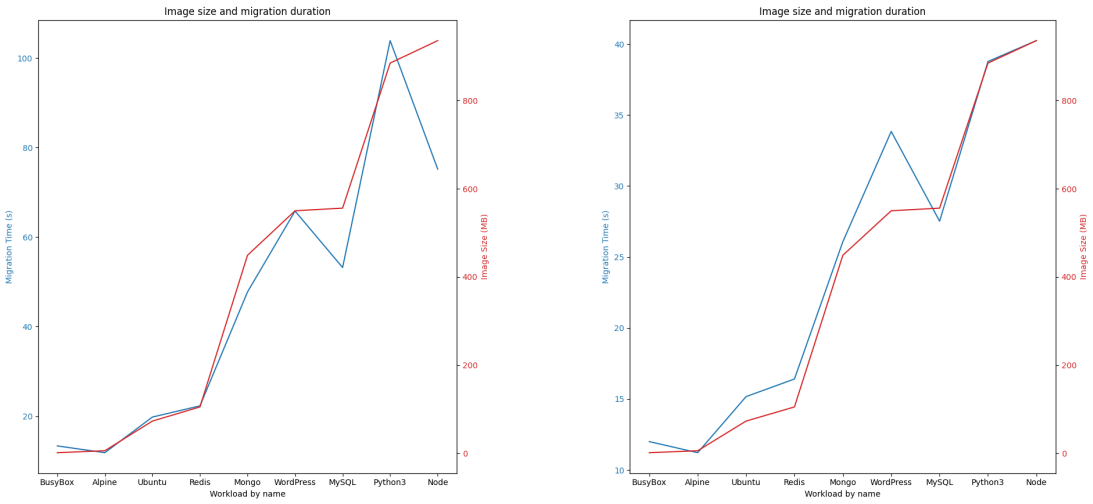


(b) Laptop #2

Figure 10: Pie chart of migration duration's. The chart provides an average of each tasks duration across all migrations.

When looking at the data in appendix-A.1 and appendix-A.2 we can evaluate if the disk migration duration corresponds with the actual read/write differences between the two systems.

Another observation of the data is that the migration duration seems proportional to the image size. As images become larger, the duration of the migration increases. In figure-11 the image sizes and durations are plotted to illustrate this trend. If the image sizes and the migration durations are indeed correlated it is potentially possible to predict the individual migration duration based on the image size. As mentioned before the dominant attributes of the system seems to be memory and processing, but due to the constant nature of the processing, the disk might be the attribute of the system which describes the migration duration.

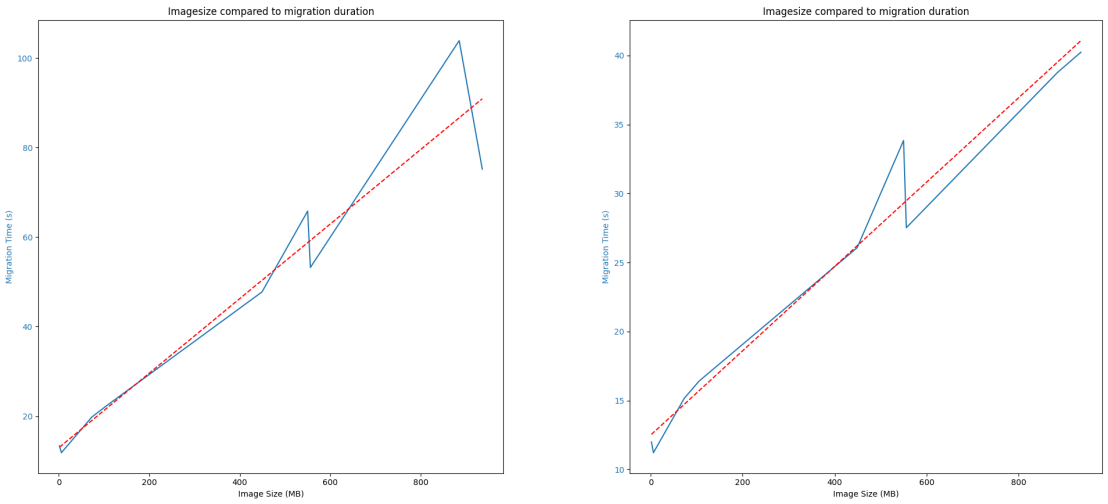


(a) Laptop #1

(b) Laptop #2

Figure 11: Image size compared to migration duration based on individual images.

As seen in figure-12 the correlation between image size and migration duration seems to be linear, with a couple of outliers. We can observe in figure-11 that the outliers within this graph are MySQL and Node for Laptop Setup #1, and WordPress for Laptop Setup #2. It is unclear why exactly these images do not follow the linearity of the rest of the comparisons, however, the deviation is not significant. The trendline produced in both subfigures in figure-12 both show that there is a trend between image size and migration duration and with the only substantial deviation being in Laptop Setup #1.



(a) Laptop #1

(b) Laptop #2

Figure 12: Image size plotted against migration durations to determine any correlation between these two data points.

As mentioned in the implementation, the migration process of a VM is relatively simple. The only method which is used to perform the migration is a single method that starts the migration process. This makes collecting durations of different tasks into categories complicated, due to there only being a single method to collect time on. Of course, the other steps of the process like, adding and removing the workload from the JSON array and sending HTTP requests from source to destination and back, can be timed, however, a thorough segmentation of the duration can not be performed.

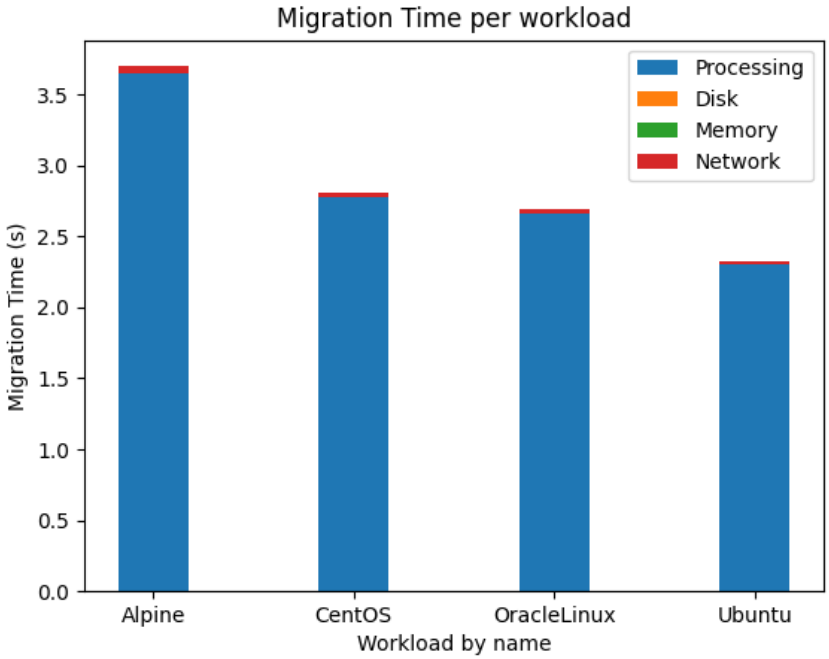


Figure 13: Migration duration based on workload. Each migration duration is separated into categories. The graph shows little to no correlation between workload size and migration duration.

As seen in figure-13, the VM workloads and their migration durations are constant, even though the operating systems have a wide size. This will be discussed further on why this is in the discussion in section-8. As expected the network tasks are relatively constant, all having the same duration as the container migration examples, varying in size with a small deviation.

7.2.2 Auto migration algorithm

For ease of operation and for DR to operate based on electricity pricing, the system can monitor current prices and attribute them to the migration of workloads. As indicated in figure-14 the system defines a threshold bound, from which it decides when to migrate a workload(s). The indicated threshold can manually set or by utilizing the implemented auto migration threshold feature, automatically define the threshold, based on past pricing events.

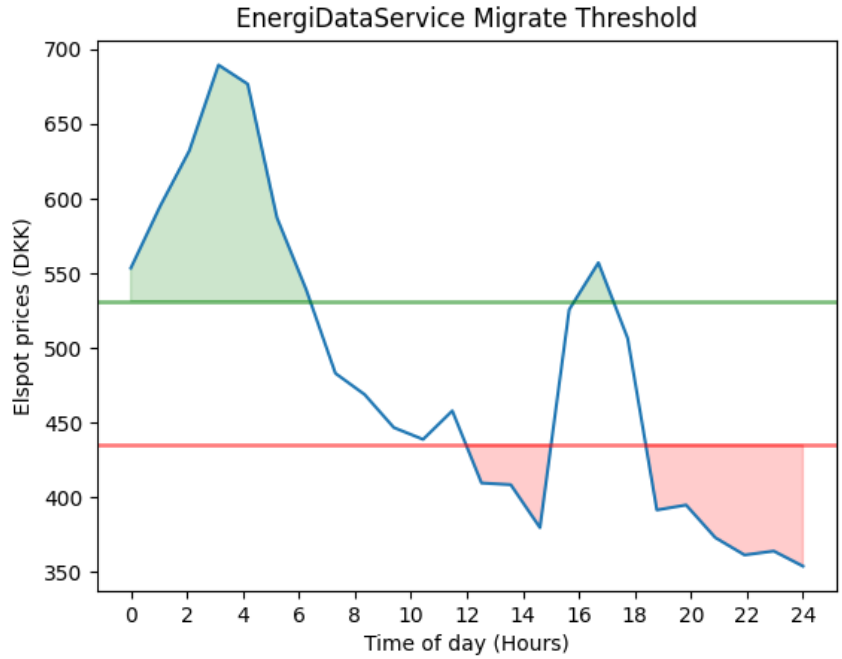


Figure 14: Auto migration threshold

The threshold of migration is based on the Elspot Prices by Nord Pool[39]. To calculate the upper and lower bound of the auto migration the following equation is used:

$$weeklyAverage \pm weeklyAverage * \frac{migrationThreshold}{100}$$

This provides a threshold of migration which is based on weekly activity.

Whenever the Elspot price exceeds the upper bound of the threshold, the system needs to consume less energy, and therefore the migration process is initiated. When the Elspot price dips below the lower bound of the threshold, the system needs to consume more energy, and can therefore undertake more workloads.

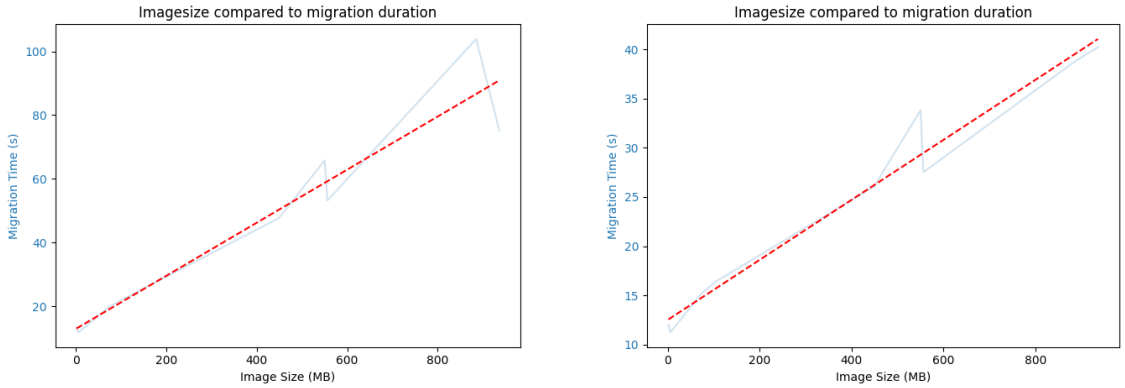
The system is currently able to migrate based upon the threshold, if the current pricing raises above, however, it is not possible to migrate the workloads back to the original machine.

8 Discussion

With the results and evaluation done, several key observations have to be discussed. Most of these topics were uncovered throughout the later stages of the development and sparked further interest in the topics at hand.

8.1 Container migration time predictions

Throughout the evaluation of the migration output data, a significant attribute of the migration was uncovered. The image size of containers was correlated with the total migration duration of the container. This does make sense due to the nature of file reading and file writing. The larger the file, the longer duration it takes to fully read the file, and vise versa for writing files. A container being migrated is essentially a large file transfer from one machine to another, and therefore the migration duration is proportional to the size of the files which comprise the container. However, as expected other tasks need to be executed in the process of container migration, which includes; starting and stopping the container, checkpointing and restoring the memory of the container, cleaning up supplementary files from the various controllers, as well as network calls.



(a) Laptop #1

(b) Laptop #2

Figure 15: Highlighted trendline of figure-12, to highlight the correlation between image size and migration duration

With the trendlines of the comparison between image size and combined migration time of each container highlighted in figure-15, it can be noted that they follow the trend

without much deviation. The formula of each trendline is:

- Laptop Setup #1 $y = 0.030530 * x + 12.506046$
- Laptop Setup #2 $y = 0.083262 * x + 12.968049$

The y-intercept of each linear equation almost entirely consists of the processing addition of 10 seconds. A further explanation of this constant will be discussed further in section-8.2. The remainder consists of the minimal duration of the remaining tasks, including memory, disk, and network.

As described before, the majority of the migration duration is reliant on disk performance. The speed of disk setup is described in table-8, and the main difference between the two setups was that Laptop #2 was energy-intensive between 3.5x and 3.7x faster than Laptop #1. When observing the linear formulas for the aforementioned trendlines of each system, the difference between the slopes of each line is only 2.6x difference. This is most likely because not all of the tasks which are categorized as disk are disk tasks, but also either memory or processing tasks. Therefore it is not possible to do a direct correlation and therefore conduct an accurate prediction of migration durations that fits both systems. However, with the knowledge of the trendline of migration duration based on the image size of each system, it is possible to predict the migration duration of a given workload on each setup respectively.

In practice, more evaluation of the system would be of need, which includes more separation and further categorization of tasks. A proposed solution to the prediction of migration durations on a given setup, would be to monitor a set of migrations and then create a prediction equation, similar to what was made in figure-15.

8.2 Processing durations

The most interesting of the outputs from the migration process is the processing duration. All of the tests which were conducted had a processing duration of just above 10 seconds. This seemed odd, due to this being the only variable that did not change significantly during large deviations in image size. By looking through the logs of the guest controllers to see where the constant durations were originating from, or if the processing tasks indeed were constant between the two setups. It was found that it was the clean-up process that was the offender. Within the clean-up process, the workload JSON file was removed from the JSON array, which should not have a duration of 10

seconds, and then the container was stopped. The former was the problem. It was then discovered that Docker has an internal timer when gracefully stopping a container, and the container exceeds 10 seconds to shut down, Docker kills the container. This is known as a grace-timer. If all of the container shutdowns were exceeding the grace-timer, and Docker was killing each container exactly after the grace-timer, the constant processing timer could be explained. However, this grace-timer can not just be excluded from the migration duration altogether. Containers being stopped, compared to killed, are stopped gracefully. This means that the container is informed about the shutdown and can finish up running tasks before the shutdown occurs. This is crucial to ensure data integrity. When running applications where data integrity is a priority the grace timer should be further increased to ensure that processes stop gracefully and are not killed.

However, an argument can be made to exclude the grace-timer all together to increase migration durations. Because the crucial information about the container is saved, i.e. the image and memory state, the graceful shutdown is not a priority. Because the complete state of the system is saved, before handling the shutdown, no amount of data corruption or data loss would be of impact to the system. However, this could be of impact if the migration is unsuccessful, and the container has to be restarted on the source guest.

8.3 Adding Workload types to the system

The addition of nodes within the system is a relatively simple implementation, however, only two workloads were prioritized in the solution, containers, and VMs. The individual implementation ideas for each workload about to be discussed will be described in further detail in section-9 Further Work.

8.3.1 Kubernetes

The addition of Kubernetes in the system would be of some benefit. Kubernetes is a technology used to containerize large applications and can be very attractive due to its decentralized nature. However, because Docker containers were a priority, the addition of another container-like technology would seem redundant. Even though Docker, as implemented in this project, does not support the decentralized structure of Kubernetes, Docker Swarm does.

8.3.2 Docker Swarm

Docker Swarm, like Kubernetes, was not a priority of the development. However, because Docker Swarm can be managed through the same SDK as the normal Docker implementation, no further tool dependencies would be needed.

8.3.3 Hypervisors and VMs

The reason that Libvirt was used as the virtualization API of choice, was its interoperability with a lot of hypervisors. When testing and experimenting with VMs, the accessibility of a hypervisor like QEMU or KVM on Linux systems, together with Libvirt, was more time-efficient than reserving another machine to run fx a Xen hypervisor. With the setup used throughout the experiments, VMs could be created and managed on the same machine, as those were running the developed controller software, only needing one machine effectively for each setup.

8.4 Applications in a data center

The applications of a demand response system in a data center vary from use-case to use-case. However, a few proposed setups and uses of a system like this will be presented and discussed in the following subsections.

8.4.1 Prioritization of workloads

As described in section-8.1 it is now possible to predict migration durations of workloads. This, in itself, presents the ability to start prioritizing workloads in the demand response system. With a part of the workloads in a data center, fx, being workloads consisting of large files, these workloads can get a priority meaning that they will be migrated at a later stage of the migration process. Because these workloads take a longer time to migrate, their potential migration downtime will be larger, and therefore to optimize downtime across the data center, these workloads will be migrated last in the migration process.

Other parameters of each workload could also aid in this prioritization. If fx it is possible to estimate the energy consumption contributed by a workload, you can factor in this together with the workload size to provide an even more accurate prioritization, allowing the data center to migrate the energy intensive tasks first, which would other-

wise contribute more to energy consumption than its peers. This prioritization could be added to each workloads specification JSON, and then implemented within the Energy-DataFetcher. The problem with small energy efficient workloads, might be subject to over-migration, hence making hosting of such workloads less attractive to the customer, if the downtime of a workload is a critical attribute of such workload.

This allows for another priority to integrate into the system. An SLA based attribute which tells the system which workloads are downtime critical, and if so how critical they are. This is very workload-specific, and has to be agreed upon in fx the SLA.

8.4.2 Migration within the data center

Another application of the proposed demand response system within a data center context is the concept of migrating workloads internally within the data center. When running a workload in a data center, there is no standardized priority of where each workload is distributed upon startup. For example, a data center with a capacity of 60% might have workloads spread out across the data center. Figure-16 illustrates this setup.

Figure 16: Data center workload spread. Each rack representing a typical data center load.

With a setup like this, all servers are running with the environmental system supporting all of the servers with conditioned air. As described before 30% of a data centers power consumption goes to cooling of the servers[5]. This includes cooling and conditioning of the servers, to allow them to not overheat. However, these environmental control systems have the ability to cool servers, no matter if they are at 60% capacity, or if they are at 100% capacity.

Therefore the desired state of a data center, where 60% of its capacity is occupied with workloads, where energy efficiency is a concern, workloads have to be compressed into adjacent server racks. The remaining servers can therefore be shut off, allowing for even more power savings, due to the large amount of power each server consumes while idling[5]. Figure-17 illustrates this setup.

Figure 17: Data center workload spread after workload migration

This setup would effectively need another stage in the migration process. Instead of having a normal operation stage, and a migration stage, an intermediate stage where workloads are compressed on the data centers could be added. Another stage where power consumption needs to be increased could be added as well. This stage would start migrating workloads to servers that have no workloads running on them, starting the servers, and therefore increasing power consumption.

Although servers and their hardware, as well as data centers in general, do not have a set amount of power consumed, an evaluation of this process can be made through estimation. As described in the article by J. Li, Z. Bao, and Z. Li.[5], power consumption of a data center is estimated to consist of IT infrastructure (servers and storage) which contribute 56%, cooling systems contribute 30%, power conditioning contributes 8%, networking equipment contribute 5% and lighting and security systems contribute 1%.

To get an estimate of the power consumption of a data center, a 30-server setup described by J. Li, Z. Bao, and Z. Li.[5], consumed 78,23 kWh. With this knowledge an estimation can be made about the power consumption of the setup illustrated in

figure-16, as seen in table-11.

Name	**Power Consumption (%)**	**Power Consumption (kWh)**
IT Infrastructure	56	43.8
Cooling	30	23.5
Power Conditioning	8	6.3
Networking	5	3.9
Lighting and Security	1	0.7
Total	100	78,23

Table 11: Data center electrical power consumption

With this in mind, another factor can also be added to the server power consumption. J. Li, Z. Bao, and Z. Li.[5] also describe the idle power consumption of a server, versus the power consumption while the server is under 100% load. With the experiments conducted in the aforementioned article idle servers consumed between 43% and 53% the energy of a server with 100% load. Another note is the observation that CPU utilization is approximately linear to the power consumption.

With this knowledge, a complete estimation of each element in the mock-up data center can be made. By shutting off cooling for 2 of the racks, and eliminating any workloads on these servers, preventing them from going idle, a sizeable energy cost can be decreased from the data center. The 23.5 kWh allocated to cooling in table-11 can be decreased to 15.6 kWh, and the 43.8 kWh allocated to IT Infrastructure can be decreased to 29.2, by eliminating a third of the consumption in these categories. This is due to two out of six servers being completely offline, consuming no power through the IT infrastructure and cooling. With an updated table-12 describe the updated setup.

The resulting system consumes 71.2% of the original setup, by simply reorganizing the workloads running on the data center.

The act of compressing workloads to a single part of the data center would be an effective compromise between downtime and energy consumption savings. However, this approach does not only provide benefits to the data center in question. The servers which are more frequently used for a large number of workloads will have a shorter life span due to over-usage. Workloads will also experience more downtime, due to the

Name	Power Consumption (%)	Power Consumption (kWh)
IT Infrastructure	52	29.2
Cooling	28	15.6
Power Conditioning	11	6.3
Networking	7	3.9
Lighting and Security	1.2	0.7
Total	100	55.7

Table 12: Data center electrical power consumption after migration.

additional step added, which would not have been there without this deployment.

8.5 Experimental Setup Evaluation

The experimental setup used in section-7.1.6 was used to emulate a data center in a minimal form. The act of emulating a data center's computing power, is not a simple task, due to the sheer performance difference between a data center and consumer hardware. However, the experimental setup used throughout the experimentation provided sufficient performance to evaluate the solution. Because the two laptop setups were relatively similar in all aspects but the storage speeds, this attribute could be directly correlated. This made it possible to directly identify how much the storage speeds had an impact on the migration durations, and therefore determine migration durations based on that parameter. In addition to this solution would be to add even more setups, which were identical in disk performance, but differed in CPU performance. This would allow the migration duration to be even further described. Another improvement is to experiment with the general performance of the system. By using cloud server hosting, the potentials of scaling performance to a data center level becomes a reality, by literally using the resources of a data center. However, without hardware-level access to the components, as well as potential user permission degradation, something like a shared storage solution would be problematic to create with relatively fast speeds. However, as this has already been evaluated with the existing setup, the combination of the two aforementioned approaches could provide benefits to the evaluation.

8.5.1 Evaluation of VMs

The migrations of VMs were not implemented as in-depth as the Docker Container migration process. This is due to the Libvirt-go API method of migration being a single method call to execute. Libvirt does provide a subset of methods, which makes it possible to control each step of the migration process, as well as customizing the migration technique, i.e. post- or pre-copy, used by the migration method. This would provide further insight into the task categorization of the migration process by Libvirt.

Another aspect briefly mentioned in the evaluation in section-7.2, was that the migration durations of all VM workloads were almost identical, although the workload sizes were different. Although no clear documentation of this could be found, as of writing this thesis, it is assumed that the migration method done by Libvirt runs asynchronously with the controller software. This would explain the anomaly of the migration durations being close to identical without much deviation. However, this assumption voids the need to perform the migration evaluation of the VMs, due to the task categorization being negated completely. This is a downside of Libvirt, which will need to be investigated further, and will be further described briefly in section-9.

9 Future Work

Throughout this section, elements that would benefit the developed solution, but either have not been prioritized or have been identified during the development process, are discussed and addressed.

9.1 New types of workloads

A future build of the system must introduce new workloads to broaden the potential of the system. Future workloads include:

- Kubernetes
- Docker Swarm
- Hypervisors
  - VMWare
  - Xen

All workloads are operational in the industry and therefore serve as an important addition. The support for each workload will be clarified further below. With further additions of workloads to the system, the use-cases of a DR system like the one described throughout this thesis would potentially become more attractive to the enterprise data center market.

9.1.1 Kubernetes and Docker Swarm

Kubernetes and Docker Swarm are both technologies that distribute tasks across worker nodes, which all contribute to a larger processing goal. To migrate a Docker Swarm or a Kubernetes Cluster to another server is therefore relatively simple. To migrate the workload, the workers have to be migrated and therefore move the processing from one server, or a set of servers, to another server. Implementing this behavior, however, requires additions to the already developed solution. For each workload specified as a Docker Swarm or a Kubernetes Cluster, each worker has to be specified as well. This is mainly the JSON description of the workloads object which would be altered, with accompanying migration methods that would have to be implemented.

```
{
    "Identifier": string,
    "AccessIP": string,
    "AccessPort": string,
    "AutoMigrate": boolean,
    "Available": boolean,
    "SharedDir": string,
    "Type": "Container" | "VM" | "Swarm":
    "Containerproperties": {
        "ContainerID": string,
        "Image": string,
        "Checkpoint": boolean,
        "NetworkSettings": object,
        "CheckpointID": string
    }
    "VMProperties": {
        "DomainName": string,
        "ConnectionURI": string
    }
    "SwarmProperties": {
        "ManagerNodes": { [
            "ID": string,
            "HostName": string,
            "Status": string
  ] }
    }

}
```

Listing 23: JSON Structure Workload - Swarm Added, guestcontroller

The SwarmProperties field described in listing-23 would make it possible to add a JSON array containing all of the information of the Swarm manager nodes in the system, with their accompanying ID, hostname, and current status, which are all fields used to identify each node. This information could be generated automatically by using the Docker SDK for Golang.

To migrate a Swarm workload all of the nodes need to be drained, and then restarted on the destination guest. This is done by first identifying the workload, then accessing

the NodeSpec of the workload, and finally setting the Availablity tag of each worker to Drain. While this is occurring, the transferHandler of the destination guest would identify that a Swarm migration was happening, starting the specified manager nodes upon the destination guest. By doing this asynchronous would result in the least amount of downtime, due to workers being drained and created simultaneously.

Kubernetes follows a similar course of action. An API candidate could be the officially supported Kubernetes/client-go API which is developed by the Kubernetes team[40]. The functionality of the Kubernetes integration would be similar to the Docker Swarm implementation in that all workers need to be registered within the workload JSON, and accompanying migration methods would need to be implemented.

9.1.2 Hypervisor support

To support the integration of new workloads, an extended list of hypervisors should be supported. An extended driver for Libvirt (VMWare ESX Hypervisor driver) allows for the support of multiple VMWare hypervisors out of the box[41]. The driver allows management of VMWare ESXi and GSX with GSX being a server virtualization program with support for both Linux and windows and ESXi being a hypervisor for the VMWare vSphere virtualization platform. The Libvirt hypervisor driver would serve as a good alternative to VMWares own solution vSphere, as it only supports VMWare products.

Libvirt comes with Xen compatibility, much like the compatibility of QEMU used throughout the evaluation. Xen however, is a bare-metal hypervisor like vSphere, and therefore requires another system to be deployed on. The hypervisor itself can be managed through Libvirt, just like the QEMU example and as vSphere would be.

The advantage of using a universal virtualization API like Libvirt is its broad compatibility with hypervisors. Proprietary functions will of course, more often than not, not be available through Libvert, however, because only migration is a concern, and that the migration process is facilitated by Libvirt, this is not of any concern. When a VM is already running on a hypervisor, Libvirt can simply connect to that individual hypervisor and treat it as a specific domain. If a VM is compatible with another domain, the migration can be executed.

9.2 Specific build of guestcontroller

With all of the new workloads potentially being added to the system, compatibility issues can occur. For example, when using CRIU for Docker migrations, certain setups are not compatible, and therefore break the entire DR system. When additional workloads are added to the DR system, more of these potential compatibility issues will become present. A solution to this is using build tags when building the guest controller in Golang. Build tags are used in Golang to exclude or include parts of the implementation for the final build. This can be used to, fx, create premium features and only include them in the premium build.

The use-case for this feature is that specific builds of the guest controller can be created, only enabling the workloads that are needed for the individual use-case. If a data center does not host Docker containers, there is no need to have Docker and CRIU as a prerequisite for migrating VMs.

This constraint was what initially picked the operating system of the evaluation, Debian 11. CRIU did not work on newer versions of Ubuntu, while Libvirt was not compatible with the older versions of Ubuntu and Debian. Therefore to get the correct compatibility of Libvirt, while also being able to install CRIU and all of its dependencies, Debian 11 was one of the only candidates. This problem can be negated completely, due to the complete exclusion of these tools through the use of Golang build tags.

9.3 System improvements

The following section will dive into system improvements and future goal as how to accomplish set goals.

9.3.1 Transparency of actions

To ensure transparency of actions and enabling the operator to follow the order of execution, a visible Logger would accomplish set goal. By utilising the JDK Logger and thereafter implementing the java.util.Handler it is possible to log every action deemed necessary for the operator, by defining each logging action by the right level. The Logger has several levels, thus the *INFO* level would suffice and ensure only operational methods are visible for the user. This in turn also ensures a log of all previous actions is available and therefore fulfills both requirements.

9.3.2 Add guests through interface

Currently, the solution only allows for the addition of new guests through the Guests.json. The current approach is not intuitive, therefore an update to the GUI to support adding new guest through an interface for declaring guests would be a priority update.

9.3.3 Graphical representation of Energy Data

The system automatically retrieves energy data, but as there is no way of seeing the collected prices, a GUI update which features a live data graph, allows for administrators of the system to monitor and independently decide when to migrate.

9.3.4 Further adjustment of Energy Data

As the current migration method is based upon a calculated threshold, the addition of other parameters used for migration expand the usability of the system. The introduction of grid capacity and grid demand/load enable operators with extended migration flexibility. The introduction of grid capacity will refer to the systems ability to deliver the dynamic energy consumption of the consumers [42].

9.3.5 Grid capacity

This data allows the operators to decide upon migration if the current capacity is unstable or unable to meet the requirements of the consumers.

9.3.6 Grid demand/load

By implementing electricity demand/load the system will be able to represent the rate at which electricity is consumed, and based on that input, the system should be able to produce a clever decision on when to migrate.

9.3.7 Different inputs of Energy Data

For a fully flexible DR system, the addition of multiple support factors for energy data would allow the operators to modify the specific requirements and customize a solution for the individual customer, as to when and how the DR system enables migration. The implementation of support for grid consumption would allow the DR system to not only

be able to migrate based on certain pricing but migrate based on the current utilization levels of the grid.

9.3.8 Workload naming scheme

As a data center can host multitudes of workloads, where multiple host could run identical services it is important for the system to be able to handle similar services. To ensure services will not be identical and/or to similar to the point of indistinguishable, unique modifiers will be integrated into each workload. Each identifier will serve as a tag to distinguish between workloads and thereby resulting in easier management of workloads within the system.

9.4 Domain specific options

A further extension of the migration options include the implementation of a geographical workload migration. Such migration should migrate workloads based upon grid capacity, workload importance and SLA specific obligations if any. To enable such migration several requirements must be fulfilled besides the already stated. Information regarding grid capacity and load from the desired location and an agreement with a data center within that destination, at which the migrated workloads will be transferred. The destined data center would be required to have an instance of the GuestController running for the system to be able to migrate between, moreover the current NFS solution would need to be modified to handle external access.

9.4.1 Containers

If the current NFS solution is not viable for the targeted data center, alternatives must be persued. An alternative for container migration would entail migration of container volumes and an implementation of a custom Docker Registry, to where each image will be pushed and fetched. At migration the host would pause the container and compress the volume, afterwards the container would continue on as the volume is being transfered to the targeted host, where it will be decompressed. The image would then be fetched by the target host, be compiled and executed with the migrated volume. Hereafter, the original host would be terminated and the migration will be complete.

10 Conclusion

Within this final section of the thesis, the research questions asked in section-1.4 will be concluded upon.

The developed solution provides a system that is able to automatically migrate workloads, based on an energy pricing parameter, meaning that the developed solution is a DR system, however, the core functionality, as well as the subject of the first research question *RQ1*, is migration.

–***RQ1:*** *How can workloads be migrated to facilitate a DR system in a data center context?*

As ***RQ1*** states, the topic of workload migration in a data center context, was the core focus of this project. Though abstractions like virtualization and containerization, workloads were facilitated and, through SDKs, APIs, and tools, migrated. Docker was the choice for container migration and facilitation, and Libvirt was the virtualization API used to orchestrate hypervisors to perform migrations.

–***RQ2:*** *Which attributes are important to a data center participating in such a DR system?*

The aspect of criticality and data integrity is what drives data centers, and provides the core use-cases for the data centers. By using tools like CRIU to checkpoint and restore the memory state of containers, additional state preservation was achieved, without a significant downtime increase of each workload.

–***RQ3:*** *How can a DR system in a data center context be designed to support multiple types of workloads?*

With the use of networking and shared storage between the guests, a simple three-phase setup was used to perform the migrations. With this framework, the only element which differs between workload types is their specification documents, and the migration methods, therefore, creating an ecosystem that enables simple implementation of additional workloads.

–***RQ4:*** *How can the state stay preserved while performing workload migration?*

As described in the answer to *RQ2* the state of workloads can be preserved through

tools like CRIU for containers. The technique to complete state preservation of a workload migration is very dependent on the workload itself. For VMs, most hypervisors, have a migration method for live migrations, i.e. migrations which preserved the complete current state of the workload.

Furthermore, with the research questions answered, throughout the development, an additional conclusion surfaced. During the migration results, a correlation between the size of the workload being migrated, and the migration duration was discovered. This, with more extensive testing, as well as further implementation, could provide an efficient way of calculating the migration duration of a given workload.

Data center DR contains a variety of aspects that needs to be further experimented with to provide a reliable commercial solution. However, the results of this thesis aim to provide a general understanding of what it takes to develop such a system.

Bibliography

[1] Ørsted. "CSR-rapport: Ørsted nærmer sig fuldstændig grøn omstilling: 99% vedvarende energiproduktion i 2025". In: (2019). URL: https://csr.dk/%C3%B8rsted-n%C3%A6rmer-sig-fuldst%C3%A6ndig-gr%C3%B8n-omstilling-99-vedvarende-energiproduktion-i-2025.

[2] Marianne Vang Ryde - DTU. "Sådan kan vedvarende energi gemmes". In: (2013). URL: https://www.dtu.dk/Nyheder/2013/11/DYNAMO_Saadan-kan-vedvarende-energi-gemmes?gclid=Cj0KCQiA3NX_BRDQARIsALA3fIKHHWOvgKB8_uL1633FtZTZEo9vBV9DsakUbKMds7w9i_2JfAcYaAmFjEALw_wcB.

[3] Klimarådet. "Store datacentre i Danmark". In: (2019). URL: https://klimaraadet.dk/da/analyser/store-datacentre-i-danmark.

[4] A. Wierman et al. "Opportunities and challenges for data center demand response". In: *International Green Computing Conference*. 2014, pp. 1–10. DOI: 10.1109/IGCC.2014.7039172.

[5] J. Li, Z. Bao, and Z. Li. "Modeling Demand Response Capability by Internet Data Centers Processing Batch Computing Jobs". In: *IEEE Transactions on Smart Grid* 6.2 (2015), pp. 737–747. DOI: 10.1109/TSG.2014.2363583.

[6] H. Wang et al. "Proactive Demand Response for Data Centers: A Win-Win Solution". In: *IEEE Transactions on Smart Grid* 7.3 (2016), pp. 1584–1596. DOI: 10.1109/TSG.2015.2501808.

[7] C. Tang and M. Dai. "Demand Response Control Strategies for On-campus Small Data Centers". In: *2012 9th International Conference on Ubiquitous Intelligence and Computing and 9th International Conference on Autonomic and Trusted Computing*. 2012, pp. 217–224. DOI: 10.1109/UIC-ATC.2012.97.

[8] James Hamilton. *Cooperative Expendable Micro-Slice Servers (CEMS): Low Cost, Low Power Servers for Internet-Scale Services*.

[9] CISCO. "What is a data center?" In: 2021. URL: https://www.cisco.com/c/en/us/solutions/data-center-virtualization/what-is-a-data-center.html.

[10] IBM. "Data Centers". In: 2021. URL: https://www.ibm.com/cloud/learn/data-centers.

[11] DataCenterKnowledge. "Google Data Center FAQ". In: 2021. URL: https://www.datacenterknowledge.com/archives/2017/03/16/google-data-center-faq.

[12] Red-Gate. "Storage-101 Data Center Storage Configurations". In: 2021. URL: https://www.red-gate.com/simple-talk/sql/database-administration/storage-101-data-center-storage-configurations/.

[13] RedHat. "What is a virtual machine (VM)?" In: 2021. URL: https://www.redhat.com/en/topics/virtualization/what-is-a-virtual-machine.

[14] IBM. "Hypervisor". In: 2021. URL: https://www.ibm.com/cloud/learn/hypervisors.

[15] SmartProfile.io. "VMWARE FURTHER EXPANDS MARKET SHARE FOR SERVER VIRTUALIZATION". In: 2018. URL: https://www.smartprofile.io/analytics-papers/vmware-further-expands-market-share-server-virtualization/.

[16] Docker. "What is a Container? A standardized unit of software". In: 2021. URL: https://www.docker.com/resources/what-container.

[17] Coding Digests. "Kubernetes Basics". In: URL: https://hazelement.github.io/kubernetes-basics.html.

[18] Matthew Palmer. "Kubernetes Networking Guide for Beginners". In: 2020. URL: https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-networking-guide-beginners.html.

[19] Docker. "docker export". In: 2021. URL: https://docs.docker.com/engine/reference/commandline/export/.

[20] CRIU. "Welcome Page". In: 2021. URL: https://criu.org/Main_Page.

[21] CRIU. "Usage Scenarios". In: 2021. URL: https://criu.org/Usage_scenarios.

[22] Google. "Migrating Node Pool". In: 2021. URL: https://cloud.google.com/kubernetes-engine/docs/tutorials/migrating-node-pool.

[23] Christopher Clark et al. "Live migration of virtual machines". In: May 2005.

[24] Sarat Chandra Pasumarthy. "Live Migration of Virtual Machines in the Cloud : An Investigation by Measurements". MA thesis. Blekinge Institute of Technology, Department of Communication Systems, p. 65.

[25] Next-Kraftwerke. "What is load management?" In: 2020. URL: https://www.next-kraftwerke.com/knowledge/load-management.

[26] Next-Kraftwerke. "What does Peak shaving mean?" In: 2020. URL: https://www.next-kraftwerke.com/knowledge/what-is-peak-shaving.

[27] Linni Jian, Yanchong Zheng, and Ziyun Shao. "High efficient valley-filling strategy for centralized coordinated charging of large-scale electric vehicles". In: *Applied Energy* 186 (2017), pp. 46–55. ISSN: 0306-2619. DOI: https://doi.org/10.1016/j.apenergy.2016.10.117. URL: https://www.sciencedirect.com/science/article/pii/S0306261916315677.

[28] ProductPlan. "MoSCoW Prioritization". In: 2020. URL: https://www.productplan.com/glossary/moscow-prioritization/.

[29] KVM. "Management Tools". In: 2021. URL: https://www.linux-kvm.org/page/Management_Tools.

[30] Libvirt. "Libvirt - Virtualization API". In: 2021. URL: https://libvirt.org/.

[31] VMWare. "VMware vSphere Standard Pricing". In: 2021. URL: https://store-us.vmware.com/vmware-vsphere-standard-288068100.html.

[32] VMWare. "vMotion Process under the hood". In: 2021. URL: https://blogs.vmware.com/vsphere/2019/07/the-vmotion-process-under-the-hood.html.

[33] Libvirt. "Migration". In: 2021. URL: https://libvirt.org/migration.html.

[34] Red Hat. "Red Hat continues to lead the Linux server market". In: 2021. URL: https://www.redhat.com/fr/blog/red-hat-continues-lead-linux-server-market.

[35] Enterprise Storage Forum. "SSD Speed comparison". In: 2019. URL: https://www.enterprisestorageforum.com/hardware/pcie-ssd-vs-sata/#:~:text=Size.,into%5C%20the%5C%20motherboard%5C%20slot%5C%20interface..

[36] Docker. "Docker Engine Install Debian". In: 2021. URL: https://docs.docker.com/engine/install/debian/.

[37] GitHub - CRIU. "CRIU Checkpoint/Restore". In: 2021. URL: https://github.com/checkpoint-restore/criu.

[38] Linuxize. "How to install and configure an nfs server on Ubuntu 20.04". In: 2021. URL: https://linuxize.com/post/how-to-install-and-configure-an-nfs-server-on-ubuntu-20-04/.

[39] Energi Data Service. "Elspot Prices API". In: 2021. URL: https://www.energidataservice.dk/tso-electricity/elspotprices.

[40] Github. "Kubernetes Client/Go API". In: 2021. URL: https://github.com/kubernetes/client-go.

[41] Libvirt. "VMWare ESX hypervisor". In: 2021. URL: https://libvirt.org/drvesx.html.

[42] Electric power research institute. "The Integrated Grid: Capacity and Energy in the Integrated Grid". In: 2016. URL: https://www.ftc.gov/system/files/documents/public_comments/2016/06/00151-128393.pdf.

A Appendix

A.1 Data set setup #1

Name	Imagesize (mb)	Processing	Disk	Memory	Network	Combined
BusyBox	1.23	10.4508	1.5912	1.2374	0.0865	13.3661
Alpine	5.61	10.4903	0.6364	0.6598	0.0660	11.8527
Ubuntu	72.9	10.8092	8.2341	0.6605	0.1055	19.8095
Redis	105	11.4517	9.9771	0.7366	0.1421	22.3077
Mongo	449	10.9998	35.6863	0.6479	0.4115	47.7456
WordPress	550	10.7754	52.7723	0.8502	1.4192	65.8173
MySQL	556	11.0265	40.8654	0.9068	0.4133	53.2121
Python3	885	10.7268	90.0020	1.5878	1.5663	103.8830
Node	936	10.8670	63.2103	0.7541	0.3606	75.1921

A.2 Data set setup #2

Name	Imagesize (mb)	Processing	Disk	Memory	Network	Combined
BusyBox	1.23	10.9956	0.4401	0.5208	0.0435	12
Alpine	5.61	10.4833	0.3793	0.2941	0.0636	11.2203
Ubuntu	72.9	10.7396	4.0139	0.378	0.0373	15.1688
Redis	105	10.6571	5.2293	0.3357	0.1876	16.4097
Mongo	449	10.8071	14.7647	0.3229	0.18	26.0747
WordPress	550	10.7657	22.6207	0.396	0.0636	33.846
MySQL	556	10.5563	16.3774	0.3638	0.2272	27.5247
Python3	885	10.7617	27.3838	0.4033	0.2264	38.7752
Node	936	10.9293	28.6863	0.4103	0.2181	40.244

A.3 Class Diagram - GuestController

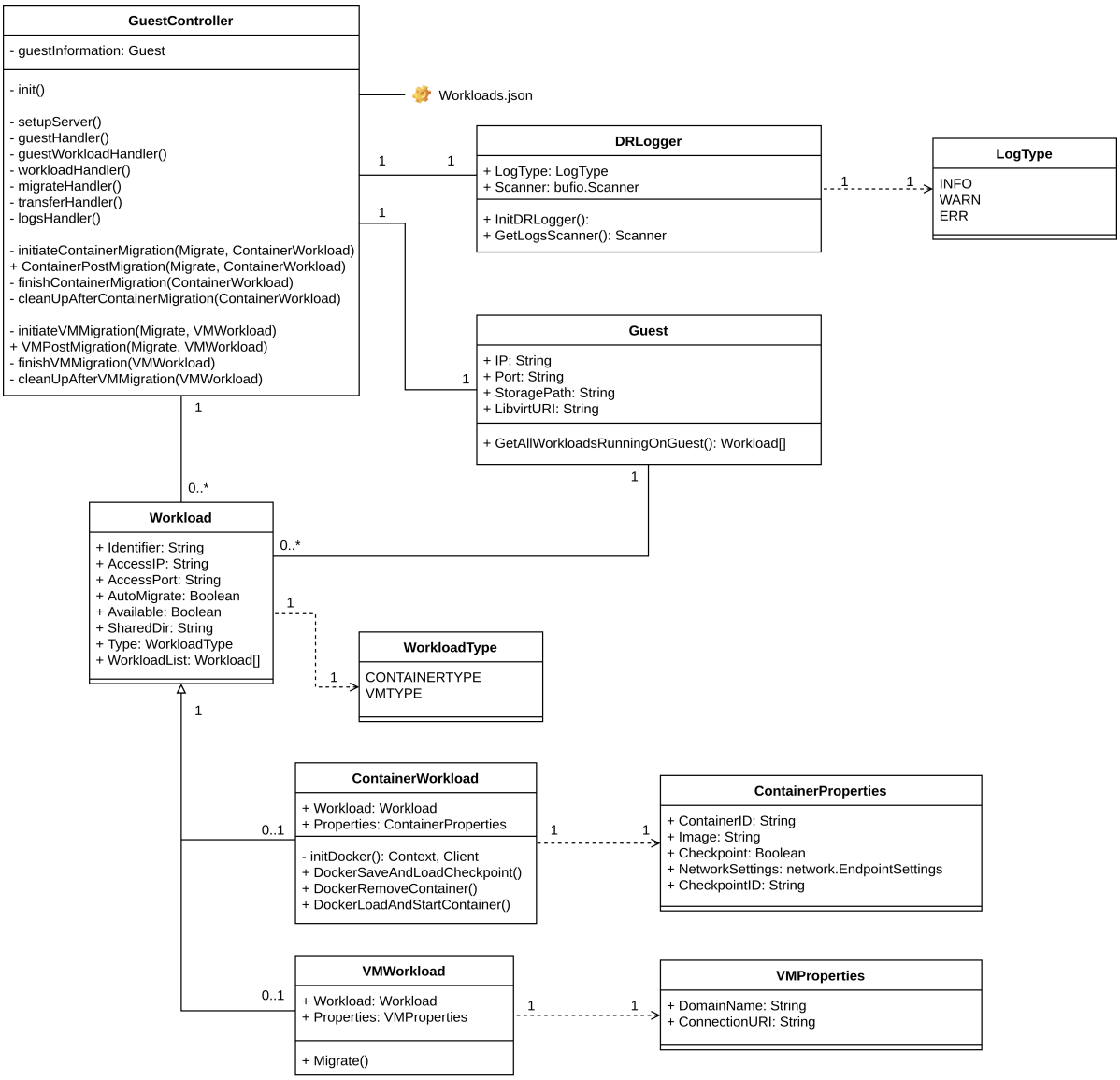


Figure 18: Class Diagram of GuestController - Golang

A.4 Class Diagram - Controller

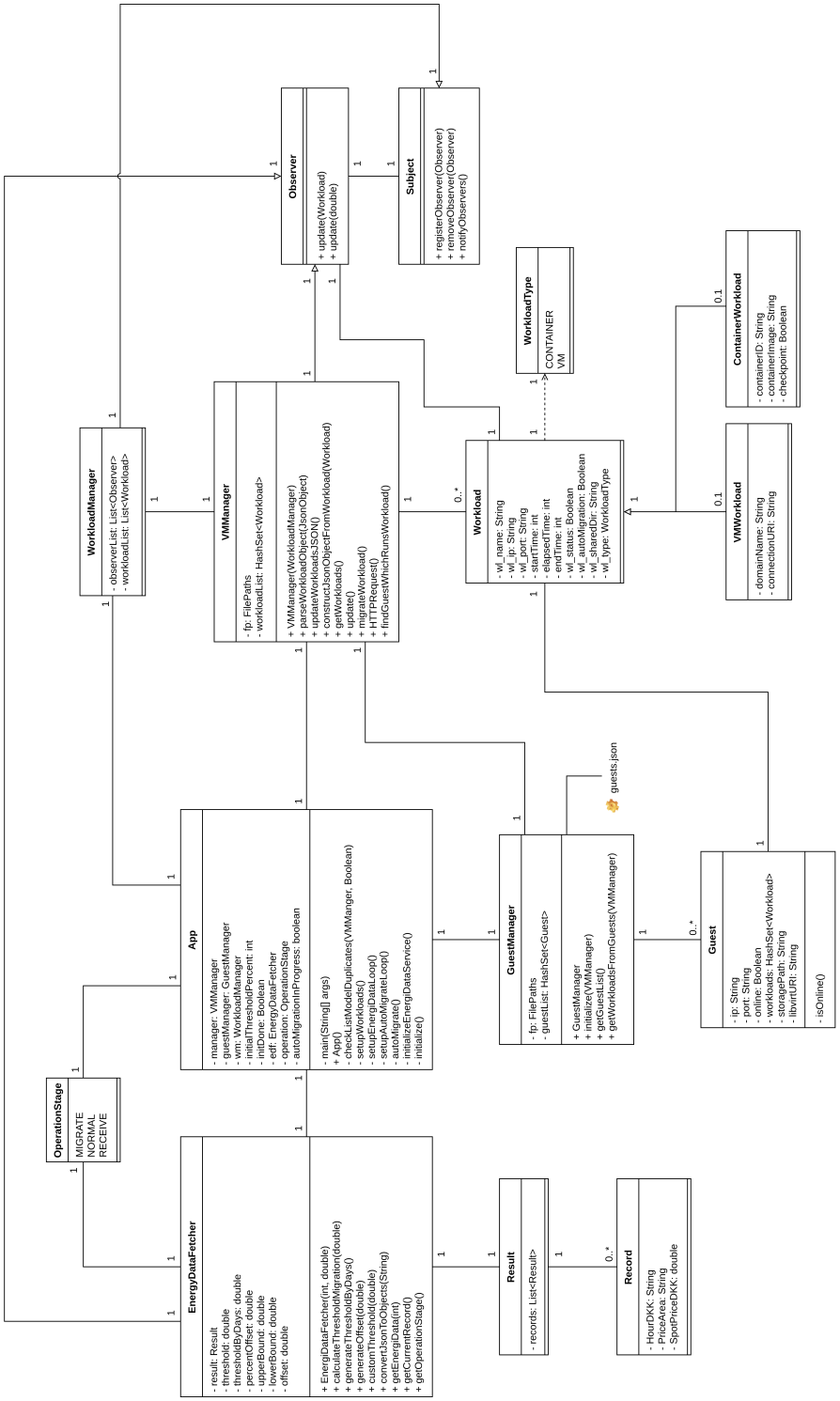


Figure 19: Class Diagram of Controller - Java