MASTER'S THESIS 2023

An Analysis of Desktop-as-a-Service Implementations

Hexu Huang

ISSN 1650-2884
LU-CS-EX: 2023-10

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-10

An Analysis of Desktop-as-a-Service Implementations

Hexu Huang

An Analysis of Desktop-as-a-Service Implementations

(Regarding Resource Consumption and High Availability)

Hexu Huang
dic14hhu@student.lu.se

April 25, 2023

Master's thesis work carried out at
Yark Network

Supervisors: Flavius Gruian, flavius.gruian@cs.lth.se
Long Qian, long@yarknet.com

Examiner: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Abstract

In recent years, with the development of server hardware and network performance, there has been an increasing acceptance of virtualization technology in the market. Among them, desktop virtualization has also become increasingly popular among enterprise and institutional customers, giving rise to many commercial products and vendors offering desktop-as-a-service (DaaS), with Yark Network being one of them. However, the virtual machine (VM) technology typically used in these products has certain performance bottlenecks and cannot meet the growing needs of customers. This project attempts to introduce container technology into the DaaS product to solve the problems of virtual machine technology.

We first built a minimal prototype of DaaS using virtual machine technology and container technology respectively and repeatedly, then collect and evaluate their performance metrics. After that, to explore the high availability of both technologies, we built minimal clusters for both implementations and simulated the downtime of a node in the cluster and observed the behavior of the cluster to see if it could recover itself.

The results of the experiments show that the container-based DaaS outperforms the conventional VM-based DaaS in terms of responsiveness and combined resource usage as the number of virtual desktop instances increases. In the high availability test, both technologies can achieve the expected self-recovery function, with the recovery speed of container technology being a bit faster than that of VM technology.

This project concludes that container-based DaaS can replace current VM-based DaaS in specific scenarios like pure Linux desktop environments and automated testing of desktop software. However, this technology currently has some limitations and can not satisfy some demands, such as no support for Microsoft Windows, inability to transfer sound, and poor graphical user experience.

Keywords: DaaS, Virtual Desktop, High Availability, Container, VM, Kubernetes

Contents

Chapter 1
Introduction

The popularity of virtualization has been growing over the past few years [23]. There are many adoptions of virtualization technology, one of which is desktop virtualization. Desktop as a Service (DaaS) is a service based on desktop virtualization. Many organizations and companies use DaaS to provide virtual desktop services. Users may reach internal systems from anywhere with their own devices or devices that are not as powerful as their workstations in the office. The only requirement is a device connected to the Internet with a lightweight client application of the DaaS installed. In the meantime, the company now needs fewer physical computers on-site, like tower personal computers (PCs) or laptops, which helps reduce the overall hardware cost and the maintenance efforts from the IT support department.

We must consider various factors to deliver a DaaS with a good user experience. Users might have diverse focuses on the requirements of DaaS, such as response time for an application, connection convenience, or graphics quality. The DaaS provided by Yark Network allows a user to connect to a remote desktop via a given link in a web browser. Users can start or shut down a remote desktop by themselves when logged into the DaaS. The company has been using Virtual Machine (VM) as the virtualization technology. In other words, each desktop instance that a user connects to serves as a VM. However, booting up a desktop instance in a VM may take minutes to initialize the working environment. As a VM runs a complete operating system inside, shutting down a desktop instance might sometimes take longer than one minute. The server that hosts the VMs also quickly becomes overloaded if many users are connected to their desktop instances in a short time. Thus Yark Network would like to turn to another virtualization technology - Container technology. In this report, we will implement DaaS prototypes based on both VM and Container technologies. Then we compare the two virtualization technologies regarding time consumption when starting/shutting down desktop instances and their resource usage.

Apart from responsiveness, another factor affecting user experience is availability. For a user to be able to connect to a desktop at any time, a DaaS must be highly available. That means we aim to reduce the probability of work interruptions and data loss. In the meantime, we may want to shorten the time the DaaS takes to recover from failures. To avoid

human intervention and make self-healing faster, we would like to introduce cluster systems and observe how they improve the availability of a DaaS based on different virtualization technologies.

1.1 Research Questions

This report aims to answer the following questions:

- **What is the impact on the response time and resource usage of Container-based DaaS compared to the VM-based DaaS?** This question will compare the response time of starting and shutting down an instance, disk read/write rate, memory usage, load average, and disk space usage between VM-based and Container-based DaaS.
- **What is the impact on the availability of DaaS when the VM-based and Container-based prototypes are extended to clusters?** This question aims to observe whether a remote desktop instance will automatically recover after the cluster member it was running on goes down.

1.2 Contribution

The contributions of this master thesis are:

- to provide a prototyped implementation of DaaS powered by containers instead of VMs
- to give a better understanding of the different response times and resource consumption patterns of Container-based DaaS compared to VM-based DaaS
- to illustrate how cluster systems ensure the high availability of desktop instances in DaaS

1.3 Approach

This section describes how we organize and carry out the work. We first looked at the existing VM-based DaaS by Yark Network to understand how it works. The second step was to look into the literature to learn more about Container technology and Container orchestration systems and how to implement DaaS using them. We also checked if Container-based DaaS exists on the market but found that most products are still VM-based. Furthermore, most of the existing DaaS products are commercial, meaning they are proprietary and closed-source. Customizing or integrating those products into the current systems, such as the user authentication service, could be complicated. Therefore we started to design and implement the prototype built with Containers. When choosing the concrete container and orchestration technology, Yark Network decided to introduce Docker and Kubernetes to benefit from their active communities to lower the risk and reduce development efforts.

The company has an existing VM-based DaaS in production. However, that system is highly integrated and consists of many components that are unrelated to the focus of this report, such as a user authentication service, a VM scheduling service and a distributed filesystem. In the meantime, the unrelated components also consume system resources and may impact the metrics we are going to collect. To make the comparison to the new Container-based prototype simpler, we realized that we had to simplify the current VM-based DaaS to make the analysis equivalent and efficient. Therefore we created a minimal VM-based prototype.

After we finished the analysis of the metrics data, we extended our VM-based prototype to a VM cluster. Similarly, a Kubernetes cluster for the Container-based prototype was established. Then we observe whether and how high availability is achieved for a desktop instance when the cluster member node where it lived on went down.

1.4 Overview

This report consists of seven chapters. Chapter 1 is the introduction chapter. It describes the research questions and gives an overview of the project. Chapter 2 and Chapter 3 introduce the relevant work and background to help understand how this project carries out. We then describe the design and the implementations of experiments in Chapter 4 and present their results in Chapter 5. Chapter 6 gathers the discussions around the experimentation results. Limitations are also mentioned in this chapter. Chapter 7 is the conclusion of the project, where we also answer all the research questions and describe future work.

Chapter 2
Related Work

Different operating systems or hypervisors may have an impact on the final performance of desktop instances. A hypervisor is a software layer that enables multiple virtual machines to share the same physical hardware. In this section, we would like to present previous work related to remote desktops, including performance evaluation with a different OS – Microsoft Windows, and different hypervisors – XenServer and Hyper-V. We also present research showing how DaaS is built on another open-source virtualization platform – OpenStack, and how a DaaS solution can be profitable as a product. These works have inspired us to investigate metrics that might be interesting for this project and highlighted the potential usefulness of the DaaS System we are analyzing.

2.1 Performance Evaluation

Similar research questions regarding Microsoft Windows have answers in the paper "Performance evaluation of virtual desktop operating systems in Virtual Desktop Infrastructure." Nakhai et al. used Microsoft Remote Desktop Virtualization Host (RDVH) to implement a Microsoft Virtual Desktop Infrastructure (VDI) in 2017 [37]. They implemented two Windows versions, Windows 8.1 Enterprise (32-bit) and Windows 10 Enterprise (32-bit), with Hyper-V as the hypervisor. The implementations showed the difference in performance in RAM utilization, CPU response time, and application response time based on different workloads. Their results show that according to their testing software Login VSI, Windows 8.1 Enterprise (32-bit) has less RAM utilization, and lower CPU and application response time than Windows 10 Enterprise (32-bit).

Different hypervisors may lead to performance deviations in desktop virtualization. Đorđević et al. dug into the disk read/write performance of two hypervisors: XenServer and Microsoft Hyper-V. They used HD TUNE PRO in their work "Comparing Hypervisor Virtualization Performance with the Example of Citrix Hypervisor (XenServer) and Microsoft Hyper-V"[52] for evaluation. Their results showed that if the Guest OS is Windows,

Hyper-V had a significantly higher performance than XenServer on disk read/write, especially for large data transfers.

These studies have motivated this project to investigate which metrics as interesting parameters for evaluating a desktop service in a virtualized environment.

2.2 DaaS with OpenStack

OpenStack is an open-source Virtual Infrastructure Manager (VIM), a software tool that enables the management and monitoring of virtual machines. Celesti et al. built a DaaS prototype with OpenStack in the paper "Improving desktop as a Service in OpenStack" [5]. Apart from performance evaluation, they included a few different approaches to support audio redirection for DaaS. What they measured was the application response time of remote video updates and remote audio playback. The experiment showed that the joint effort of the Guacamole proxy and the RDP protocol produces the best performance. This study highlights the option for delivering DaaS with VIM. Additionally, it also underscores the importance of considering usability in future DaaS implementations.

2.3 A Profitable Hybrid DaaS Solution

Google first introduced the word Cloud Computing in 2006. Since then, a few big players, including Amazon, Google, Microsoft, and IBM, have entered the market and brought Cloud Computing to the role of the foundation of many Pay-As-You-Go (PAYG) services. A study by Dhall and Tan [12] highlights the potential profitability of DaaS as a product and its successful implementations in the market, and it also provides meaningful insight into the future of DaaS. The findings of this study provide impetus for further inquiry into the efficacy of Desktop as a Service (DaaS) as a viable and profitable product. The study found that DaaS has been a well-received product among Small Medium Business (SMB) customers across industry verticals with compliant regulations. SMBs have been looking forward to moving away from in-house IT to the IT-as-a-service model because the latter provides agile scalability with the PAYG model. Also, DaaS is a successful offer for government customers like the German Ministry of Justice [11]. Many DaaS vendors exist in the market, including Microsoft, Amazon, Citrix, VMWare, Google, HPE, etc. Managed Service Providers then bring the services from those vendors to their customers with various customized requirements. In the meantime, the underlying solutions from these vendors are all based on Virtual Machine technology.

Chapter 3
Background

In this chapter, we introduce the background related to the report. Firstly, we present what DaaS is and what values DaaS can bring. Secondly, we explain how VM and Container technologies contribute to the implementation of DaaS. After that, we discuss what High Availability (HA) is and how we achieve HA in DaaS based on the two technologies.

3.1 Desktop as a Service

A typical scenario of using computers in a company is to distribute tower PCs or laptops to employees who need access to the company's internal systems for work [13]. These personal computers for work are generally dedicated to one person only. The company is responsible for maintaining them, and such work includes installing the operating system before the handover, replacing defective hardware, and taking the machines to its users in person. When an employee leaves the company, the pre-owned PC or laptop must be re-initialized, and the computer needs to be physically collected, cleaned, and stored safely. Much work deserves a dedicated person to a team of several professional system administrators.

Those machines are usually high in price while poor in portability, especially the ones with high performance. Since they are powerful yet expensive, sharing them with other employees would be more cost-effective. Moreover, it is common for some companies to send employees with computers to factories or their customers' locations to work on-site for a certain period. Therefore, the traditional approach mentioned is either incapable or inconvenient.

Desktop as a Service (DaaS) delivers a virtual desktop experience to its end users at a lower maintenance and hardware cost. The virtual desktop is a virtual representation of the traditional desktop that contains a collection of applications for work installed on the desktop. In such a way, the company's IT department can focus on maintaining virtual desktop instances instead of personal computers, for example, monitoring the usage of the virtual desktop instances to take appropriate actions to optimize the performance, keeping the ap-

plications running inside up-to-date, and so on. The machines that host the virtual desktop instances are high-performance servers with optimized hardware that are better at multitasking. Those servers run multiple desktop instances simultaneously to share computing and storage resources. Users can access the same virtual desktop from different devices, from a smartphone/tablet to a laptop, either on-site or remotely [47][7].

3.2 Protocols

We need a protocol for communication between users and the virtual desktop instances. There are a variety of candidate remote desktop protocols depending on client/server platforms. Virtual Network Computing (VNC) is the protocol we use in this thesis for our prototypes.

VNC is a protocol developed by The Olivetti & Oracle Research Laboratory [42]. The protocol shares computer desktops over the internet [44] based on the Remote Frame Buffer protocol (RFB), a protocol for remote access to graphical user interfaces. VNC serves as a Client-Server Model. A VNC client interacts with the keyboard and mouse of a local computer, and the RFB protocol transmits the local events, such as keystrokes, mouse moves and clicks, to the other end – a VNC server, and fetches what shows on the server's screen back to the client.

In some cases, it is impossible or undesirable to install a VNC client software additionally depending on clients' local regulations. However, a remote desktop instance can still be accessible with the help of noVNC [38]. noVNC is an open-source project created by Joel Martin. Apart from speaking the same RFB protocol in the background to a VNC server as a standard VNC client, it exposes an HTTP interface and wraps the raw RFB traffic inside a WebSocket connection. noVNC also contains several compatibility components for cross-browser support. Therefore, any device with a modern web browser can access the virtual desktop instance remotely. noVNC has been popular during the past few years. Many cloud-computing projects like OpenStack and OpenNebula employ noVNC to provide web consoles to their users. Proxmox VE also lists noVNC as one of the options to access its VMs on its WebUI.

3.3 Virtual Machine Technology

DaaS can be implemented in different solutions based on various technologies. We will cover two of them in this report, and one of the two is Virtual Machine (VM) technology.

One of the characteristics of the Virtual Desktop Infrastructure (VDI) solution is that the operating system (OS) lives within a Virtual Machine (VM) on a server. Instead of directly living on the underlying hardware, they are referred to as Guest OS and separated by a virtualization layer. Figure 3.1 shows the difference between a physical machine and a VM.

The virtualization layer illustrated here is a hypervisor. A hypervisor is one of the two main components of constructing virtualization. It works as the virtualization layer between the hardware and VMs. With a hypervisor, the OS and the applications running on it can be abstracted from the hardware in VMs. Therefore, installing more than one OS on a machine becomes possible. Multiple VMs can live on the same server, and their OSes can differ from

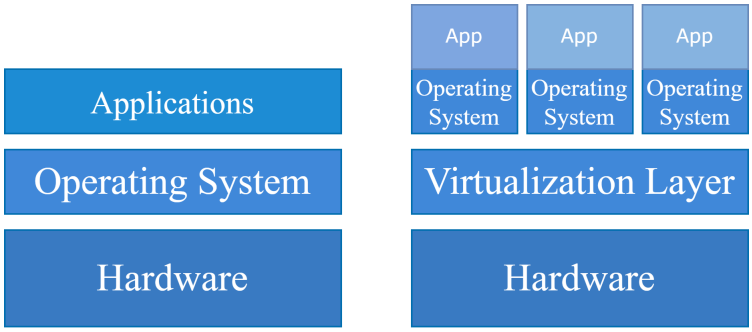


Figure 3.1: The comparison in architecture between a physical machine and a virtualization host machine.

each other and the hypervisor.

The other component is the Connection Broker, which does the communication jobs such as managing the lifecycle of VMs, user authentication, and the communication between the hypervisor and higher-level systems closer to the end-users.

3.3.1 Virtualized Infrastructure Manager

To make the experience of managing VMs more pleasant and effective, many vendors offer Virtualized Infrastructure Manager (VIM) built upon hypervisors and connection brokers to enable virtualization servers with a user-friendly interface. An administrator can create, manage, and destroy VMs on a unified platform without knowing the details of the underlying virtualization platform.

The VIM platform that this thesis uses is Proxmox Virtual Environment (Proxmox VE, or PVE)[18]. It is an open-source server virtualization management platform that uses QEMU-KVM as its hypervisor. The platform also offers a homemade connection broker. Users can utilize command-line (CLI) tools to interact with the services. Proxmox VE also provides a Web-based Graphical UI (Web UI, GUI) for users who prefer a visual interface among the user tools. Both the CLI and the Web UI are built upon the same connection broker.

QEMU-KVM

Both QEMU [41] and KVM [33] are hypervisors. However, they have different characteristics.

QEMU can do virtualization independently without the help of KVM. It can even emulate platforms different from the host's architecture with binary-code translation, such as virtualizing an ARM machine on a server powered by an x86 CPU. In the meantime, it can emulate hardware devices like hard disks, network cards, and many others. However, its performance is limited since QEMU is fully software-based and runs in the user space.

KVM is a kernel module that runs in the kernel space. It solves the performance issues of QEMU with its implementation of hardware virtualization (Intel VT-x and AMD-V) that eliminates the requirement of binary-code translation. KVM does not virtualize hardware devices as QEMU does.

QEMU-KVM [19] is a combined virtualization solution. QEMU emulates hardware devices without binary code translation anymore, and KVM virtualizes CPU and memory. KVM provides a device interface (/dev/kvm) that allows other applications to interact with

the virtualized hardware. QEMU makes use of the interface to collaborate with KVM. In such a way, QEMU-KVM takes care of the communication between the VM and the underlying KVM layer to provide a fully functional virtualized environment.

3.4 Container Technology

VDI is a solution for virtual desktop instances with hardware-level virtualization. Alternatively, virtualization can be implemented on the operating system level as well, with the help of Container technology. Each container contains a fully functional virtual desktop instance in this implementation.

A Container is a set of one or more processes isolated from the rest of the OS and other containers for running a specific workload. All the files required by running them are included in a single Container image. Unlike VM, a Container shares the kernel of the OS in which it runs instead of running its separate kernel. Figure 3.2 shows how container deployment differs from virtual deployment [15][8].

3.4.1 Container Characteristics

The main characteristic is that container-based deployments share the same OS kernel, while the VM deployments have separate OS kernels within each VM. In this way, a Container image only requires the supportive files, such as executable binaries and the dependent libraries, for the processes to run, which makes it simpler and more lightweight than a VM image. Containers are stateless. A running Container will not write anything to the image it runs from in its lifecycle, while VM usually will. Container images do not consist of environment-aware configurations. That means the same image is portable and consistent among all the running environments, including but not limited to local development, testing, and production. System administrators only need to maintain one single image per workload type. When running a new Container from its image, specific configurations are passed in via environment variables or in a mounted volume with configuration files or secrets. Furthermore, new images can be derived from the same existing image programmatically, meaning creating a batch of similar images for slightly different purposes becomes convenient and reproducible [15][8].

3.4.2 Container Runtime

Similar to VIM, Containers need an umbrella in the OS to provide the running environment and manage their lifecycles. Since a Container does not run its own separated OS kernel, the runtime also has to take care of the virtualized representation of many system resources, such as filesystem, devices, mounts, network, and the communication between containers and the host system. To ensure the interoperability of Container technologies, Open Container Initiative (OCI) [25] defined standardized specifications around Container Image, Distribution, and Runtime.

Docker Engine, which is often simply referred to as Docker, is one of the major contributors to OCI. It can be installed on all the mainstream Linux distributions like Debian. In this thesis, we use Docker as the Container Runtime. Docker provides a server daemon that

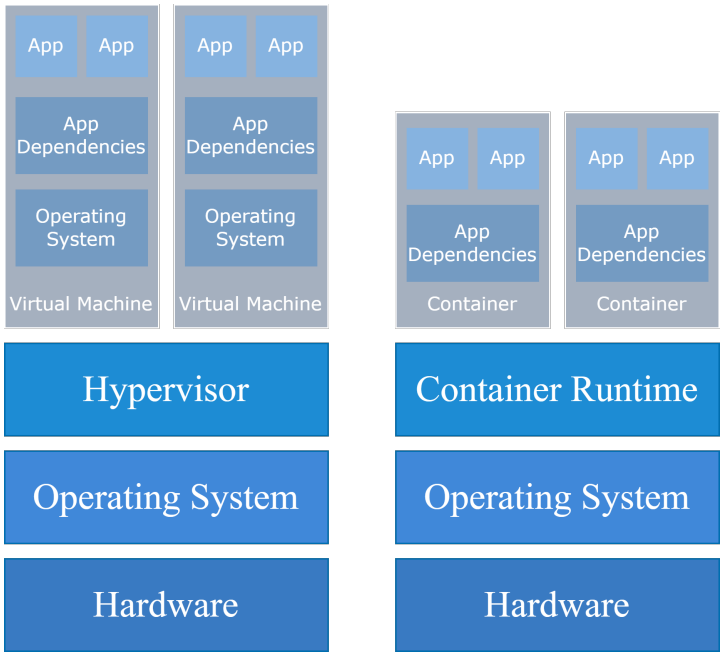


Figure 3.2: The comparison between the architectures of VM (left) and Container-based (right) virtualizations.

runs like a VIM and a command-line client to communicate with the daemon. In addition to starting and stopping containers, the client also has a bunch of subcommands to satisfy tasks like building new images and managing the container network.

The storage of Docker is powered by an overlay filesystem (OverlayFS) [4], which consists of stacked layers of files and directories. In an OverlayFS, there is always a lower directory and an upper directory, while the user sees a merged directory combined with the two. If a file exists in both the lower and the upper directories, the upper wins. In practice, an OverlayFS can have more than two layers. For instance in Docker, a running container has an overlay for its runtime storage on top of the Docker image, while the image itself can be built in multi-layer overlays [24]. With OverlayFS, different containers starting from the same image can share the image as the lower layer without an additional copy, and similarly, different images may save disk space by sharing their layers in common.

3.5 High Availability and Clusters

Building a perfect program that always executes expectedly without any problem is almost impossible. Various factors, including hardware issues, network partitioning, and external changes, could lead to downtimes in a running system. High availability has to be guaranteed for production services like DaaS. That is, the system should work continuously without significant interruptions when there is a power outage, network partition or disk corruption. When such issues happen, the cluster should be able to self-heal without human intervention [16]. We may utilize cluster solutions to avoid a single point of failure (SPOF) for systems like DaaS to achieve this goal.

3.5.1 Proxmox VE Cluster

Many companies provide High Availability for their VM-related products, such as VMWare vSphere, Citrix XenServer, and Microsoft Hyper-V. These products are all deployed as clusters to achieve HA. In the meantime, Proxmox VE, the VIM we use in this report, can also scale to a clustered deployment with a built-in tool called Proxmox VM Cluster Manager (*pvecm*).

Like other VM HA products, a PVE cluster requires a group of physical servers and a shared storage pool. Each server in the cluster is called a *node* and born equal. All the cluster nodes can act as master candidates, VM hosts, and storage pool members. A particular filesystem *pmxcfs*, powered by *corosync*, is employed to distribute the cluster configuration to all the nodes and ensure eventual consistency.

At least three nodes are needed to join a PVE cluster to achieve a reliable quorum. When a quorum is met, cluster operations can be executed on any node, and they will take effect the same way through *pmxcfs*, no matter whether the operations come from the Web User-Interface (WebUI) or the Command Line (CLI).

The shared storage pool shipped with PVE is Ceph [6]. Ceph is an open-source software-defined storage platform that delivers object, block, and file storage in a single, unified system. Ceph replicates data for two or more copies and distributes them in chunks (Placement Groups, PG) across the nodes to make it fault-tolerant. In a PVE cluster, Ceph's block storage (Rados Block Device, RBD) is used to store VM disk images. When one of the PVE nodes is unavailable, other nodes can still access the disk images of the VMs on the failed node, making it possible to start the affected VMs elsewhere. A basic structure of a PVE cluster is given in the figure 3.3.

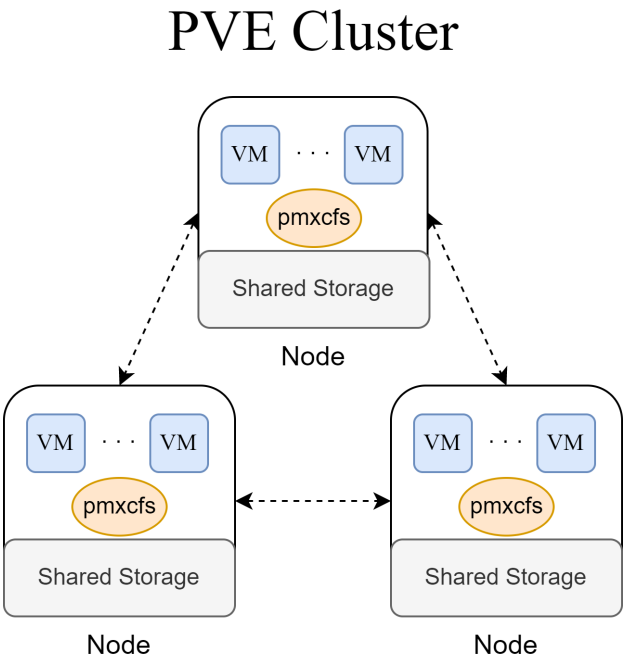


Figure 3.3: The architecture of a PVE Cluster

3.5.2 Kubernetes

In this thesis, an open-source container orchestration system, Kubernetes, will be used to manage our virtual desktop instances running in containers. Kubernetes deploys as clusters [29]. Each Kubernetes cluster mainly contains two roles - Control Plane Nodes (formerly known as Master Nodes) and Worker Nodes. Since Container is OS-level virtualization, the machines in a Kubernetes cluster do not have to be physical. Figure 3.4 shows the architecture of a Kubernetes cluster. The Control Plane Nodes maintain the metadata of the cluster and does the scheduling of workloads, while the Worker Nodes are responsible for running the workloads.

Kubernetes does not build application-level services on its own, and Containers are not run by Kubernetes directly. There are three components in every node - Kubelet, Kube-proxy, and last but not least, Container runtime, which is the service that manages the lifecycle of the actual containers directly. Kubernetes supports any container runtime that implements the Container Runtime Interface (CRI) [28], for example, Docker.

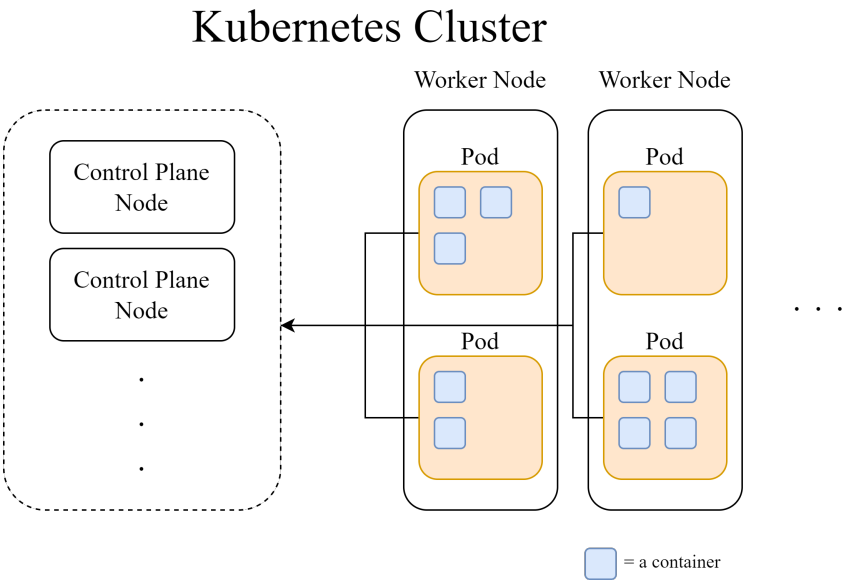


Figure 3.4: The architecture of a Kubernetes Cluster

3.6 Summary

Desktop as a Service (DaaS) is a cloud-based service that enables remote access to virtualized desktop environments. This service allows users to access their applications, files, and settings from any device and location. Virtual Network Computing (VNC) is one of the protocols that can be used to establish communication between the client device and the remote virtualized desktop. Virtualization technologies for DaaS can be further divided into virtual machine (VM) technology and container technology. VM technology employs hardware virtualization to create a fully virtualized environment for each desktop instance, while container technology utilizes a shared operating system kernel and isolates desktop instances within containers. One of the widely adopted container technology is Docker. Furthermore,

there are also commercially available Virtual Infrastructure Manager (VIM) platforms, such as Proxmox VE (PVE), that can be utilized to manage VM resources. PVE also offers Proxmox VE Cluster as its cluster solution, and Kubernetes can be utilized as a cluster solution for Container, which is used to achieve high availability of DaaS.

Chapter 4
Implementation

This chapter presents what prototypes are needed and how they were designed and implemented for experimentation. We first describe the design, including the tools and platforms used to build prototypes. Then the implementation for each prototype is elaborated in detail in the next sections.

4.1 Design

The prototypes consist of two types: essential and cluster-based. For each type, there is a respective solution for VM and Container. The essential type is to compare the response times and resource usage. The cluster-based type, extended from the essential one, is to observe the high availability behavior.

Our essential prototypes used Proxmox VE for the VM technology. For the Container case, we turned to Docker. Both prototypes were installed on the same physical machine to avoid any difference in influencing factors. The hardware specifications of this machine are in Appendix A.

The cluster-based prototypes were built on top of the essential ones. That means Proxmox VE would scale out to be a Proxmox VE Cluster, while Docker containers would be onboard to Kubernetes.

The desktop instances had the same software packages installed in all the prototypes. For simplicity and to minimize the side-effect to the measurements, we only included the following components that build up our virtual remote desktop:

- **Desktop environment Xfce4** [50], a desktop environment for Unix and Unix-like platforms. We chose it because it is lightweight while with minimal functions as a desktop environment, such as GUI, a menu bar, and a resource manager.
- **VNC-Server**, a service to provide a VNC system through which users can connect to and remotely control the remote desktop instance.

- **noVNC - HTML5 VNC client**, a service to speak to the local VNC-Server, in the RFB protocol and then translate to the HTTP protocol for end-users to access the desktop instance via a Web browser.
- **Chromium and Mozilla Firefox**, desktop applications to demonstrate the virtual desktop is working.

4.1.1 Tool Selection

Proxmox VE is complete and open-source as a VIM product by *Proxmox Server Solutions GmbH*. In addition to its diversified API and CLI resources, developers can modify its source code to adapt to the requirements (be aware that Proxmox VE's source code is under *AGPL*). As a commercial product, all feature sets, including the Cluster feature for high availability, are available at no additional license cost. At the same time, paid licenses lead to a better-maintained software repository and higher-level support services. Without buying a license, it is yet possible to seek help and support in the Proxmox Support Forum.

No commercial company is backing Kubernetes, but *Cloud Native Computing Foundation (CNCF)*, whose parent organization is *Linux Foundation*. The source code of Kubernetes is licensed under a more relaxed license *Apache License 2.0*. There is no centralized support forum for Kubernetes, but Kubernetes is so popular that many learning resources and online communities exist, plus it has quite complete official documentation [31].

4.2 The Essential Prototypes

This section describes how the essential prototypes were implemented.

4.2.1 VM implementation

The most popular service package of the company's existing VM-based DaaS has a single virtual CPU core, 1 GiB of RAM and 10 GiB of disk space, so in our simplified prototype, we also take this package set and assign such resources to each VM. An ISO file provided by Ubuntu [45] was used to install the OS of the first created VM. This ISO file is a minimal CD image that only contains the core software packages of a Ubuntu OS. We injected the image into the first VM's virtual CD drive on the freshly built Proxmox VE server, and then we installed Ubuntu on the VM with it.

An internal IP address was assigned to this VM so the VM could be connected from the Proxmox VE server. The software packages mentioned in the Design section were installed in the VM, which built a template VM image. With this template image, we can derive new VMs by copying the image file and modifying their internal IP addresses to avoid conflicts, so that a testing bed of identical VMs with functional networking is prepared.

4.2.2 Container implementation

We installed Docker Engine [14] on the same machine without any obstacles since Proxmox VE is based on Debian. When the docker daemon was up and running, the command-line tool

"docker" was used to interact with the Docker Engine to manage our docker image and desktop instances in containers. An image maintained by ConSol [10] was used for the Container implementation. This image contains the same components as the VM implementation listed in the Design section.

4.3 Cluster-based Prototypes

This section contains the cluster-based prototypes implemented on top of the essential ones.

4.3.1 Proxmox VE Cluster setup

We set up a three-node minimal Proxmox VE (PVE) Cluster [21] on the WebUI. Each node in a PVE Cluster has a vote, and votes are used to elect a master node among them when a cluster is established or when the number of nodes alive changes in a cluster, for example, when a new node joins or the connection to one of the nodes is lost.

The first step was to install Proxmox VE on all three physical machines [20]. During the installation, we added the IP addresses of all three nodes to their */etc/hosts* files to help them find each other by host names. After that, we created a cluster in the *Datacenter - Cluster* section on the WebUI of one node and generated a *Join Information*, which is an encoded string containing the metadata of the cluster. The metadata consists of the IP address, the fingerprint of the first node in the cluster, the cluster name, and the config version. Then the two remaining nodes were joined to the cluster with that *Join Information*, on the same web page.

4.3.2 Kubernetes setup

Our Kubernetes cluster consisted of two control plane nodes and two worker nodes, each of which had one virtual CPU, 1 GiB RAM, and a 10 GiB virtual disk, just like the VM essential prototype. The cluster had four nodes because two of them were Control Plane nodes, while we needed two more worker nodes to demonstrate the high availability feature.

We used Kubespray [32], a toolset based on Ansible [3], to initialize the Kubernetes cluster. Firstly, the *kubespray* project was downloaded to the local computer. Then we duplicated the sample inventory and modified the *ansible_host* IP addresses with ours in the *kubespray* folder. One last step was to adjust the groups *kube_control_plane* to include *node1* and *node2* and *kube_node* to include *node3* and *node4* as worker nodes. Then we ran *ansible-playbook* against the *cluster* playbook to provision a new Kubernetes cluster.

When the Kubernetes cluster was ready, we could start to use a command-line tool *kubectl* to interact with the cluster. To manage the workloads on Kubernetes, in our case, desktop instances, we composed a YAML (.yaml, .yml) definition file [51] to describe the desired states of the components and then used *kubectl* to apply them to the cluster.

The YAML definition of each desktop instance had three parts:

- **Deployment** defined what container image to use, which ports to listen on, and how many containers per desktop instance should be alive. If the number of living contain-

ers is lower than defined, Kubernetes will schedule a new one on the other nodes to satisfy the number again.

- **Service** linked one of the container's ports to a named service port. Among all the open ports in the desktop instance, only the noVNC port was specified here because it was the port to provide the actual desktop service.
- **Ingress** exposed the named port mentioned above to the public. Otherwise, the noVNC Service would be only accessible inside the cluster with an internal Cluster-IP address.

Chapter 5
Experiment

This chapter presents the experimentation. Firstly, we discuss what metrics we would like to collect. Next, we introduce how we collected the metrics. Then we describe how we performed the experiments. At the end of this chapter, we show the results of the experiments.

5.1 Metrics to Collect

There are four metrics that we were going to collect:

- **Response Time**, the time elapsed from the start of a desktop instance to the moment when it is ready to accept incoming connections from end-users and the time elapsed from sending the stop/shutdown command to a running desktop instance until it fully stops.
- **Disk Read/Write Rate**, the throughput of disk reads/writes per second measured on the host machine.
- **Memory Usage**, the amount of main memory used on the host machine in total.
- **Load Average**, the *loadavg* data fetched from the */proc/loadavg* file [34] provided by the Linux kernel on the host machine. The first three numbers in this file are load figures marking a running Linux machine's CPU and IO pressure, averaged over one, five and fifteen minutes. They are also known as short-term, mid-term, and long-term load averages.

These metrics can be categorized into two types: non-infrastructural and infrastructural. Infrastructural metrics are related to the server's physical hardware and network that can be collected by monitoring systems. The first one, Response Time, is the only non-infrastructural metric here because it does not reflect the running status of the physical machine, but rather how the desktop instances perform as user applications. The remaining

three are infrastructural metrics that are the raw numbers to show how the physical machine performs. These two types of metrics offer two aspects for evaluating the prototypes, and we collected them in different approaches. Additionally, we will also take a look at the disk space usage of the two virtualization technologies.

5.2 Methods to Collect Metrics

Collecting infrastructure metrics is the focus of many monitoring systems. However, non-infrastructural metrics have to be collected according to their characteristics.

5.2.1 Non-Infrastructural Metric

The scripts we used to start/shut down desktop instances on the physical machine were also responsible for collecting the only non-infrastructural metric – Response Time. We measured it directly in the scripts that conducted the experiments and generated a log file for each round of the experiments.

The log files were in CSV format and consisted of four columns: type (docker or VM), event (start or stop), ID (container or VM ID), and time elapsed for the event from start to finish. The CSV format could be used by *gnuplot* as data sources directly for visualization and analysis later.

5.2.2 Infrastructural Metrics

The remaining metrics, Load Average, Memory Usage, and Disk Read/Write Rate, belong to infrastructure metrics. The tool we used for collecting those metrics was *collectd* [9].

collectd runs as a daemon to collect infrastructural and application performance metrics periodically and then store them in a local database or pipe them to another service. In our setup, we ran *collectd* on the same physical machine where we performed the experiments. Then we configured *collectd* to send the metrics data it collected to a dedicated machine only for storing monitoring data, which also ran *collectd* to save the values. The metrics data were written into Round-Robin Database (RRD) files [39] on that machine. We did not write RRD files locally because the constant writes would interfere with the disk-related metrics we collected. When we finished all the experiments, we extracted the RRD files to get raw numbers to feed them into CSV files so that *gnuplot* could use them for plotting.

The columns of the CSV files extracted from RRD were as follows:

- **Disk Read/Write Rate**: seconds since the beginning of the round, UNIX timestamp, read bytes per second, written bytes per second
- **Memory Usage**: seconds since the beginning of the round, UNIX timestamp, used memory in bytes
- **Load Average**: seconds since the beginning of the round, UNIX timestamp, short-term, mid-term, and long-term load average values

The first two columns of those three metrics were identical, as shown above. The first column's values would be used for the X-axis. The second column, UNIX timestamp, made it easier to cross-validate the raw values in RRDs if we realized the plots looked in a different shape or pattern that is unlike others. The remaining columns were for the Y-axis.

5.3 Experiments

In this section, we will explain how the experiments are performed to collect metric values and observe the HA behaviors of the two cluster solutions.

5.3.1 Metric Collection

To collect sufficient metric data in an automated manner, we carried out a pressure test by running the experiments ten times. Each time, we had ten rounds of tests starting with one desktop instance and adding one concurrent instance in the next round. We implemented the logic above with shell scripts for both VM-based and Container-based prototypes.

In each round, we tried to repeat the starting and shutdown actions ten times to avoid possible edge cases and to make sure the results were qualified to match a stable pattern. We also scheduled five-minute breaks between the rounds and between the starting and stopping actions in the same round to avoid the impacts from the previous run or action. One reason was that the Load Average's mid-term metric was calculated based on the loads in the past five minutes. The interval between starting and shutting down the desktop instances also ensured that the desktop services had sufficient time to be completely ready. Except for the breaks above, we also dropped the system cache forcefully between the starting and stopping actions. This helped to exclude the possibility that an earlier test round affects a later round as the disk images cached in memory could skip further disk reads.

5.3.2 HA Observation

To test the HA of the VM prototype, we employed the PVE Cluster we set up in the previous chapter. We reused the same VM disk image created in the non-cluster VM experiment. For automatic VM migration to work, we had to create shared storage among the nodes. Without the shared storage, if one node was down, the source of the VM disk image would be unavailable immediately, and the VM migration was impossible due to the lack of disk image. We chose the built-in Ceph storage. Each node contributed one dedicated drive as *OSD* to the Ceph cluster. For VM disk images, we created a pool called "VM" with *min_size=2* to ensure each disk image was stored for at least two copies on two different nodes, so that they could be accessible when one node went down.

Kubernetes was used to test the HA of the Container prototype. One thing to mention is that when preparing the YAML file to deploy our desktop instance to Kubernetes, we had to add a particular block of *tolerations* [30] to the deployment configuration. The *tolerations* is a set of conditions that the Kubernetes scheduler can tolerate for a certain time before it decides to take an action defined in the corresponding *effect* field.

Kubernetes adds two tolerations automatically for nodes – *node.kubernetes.io/unreachable* and *node.kubernetes.io/not-ready* with *effect: NoExecute* and *tolerationSeconds: 300*. That means

if a node becomes unreachable or not ready to schedule workloads on, Kubernetes will still wait 300 seconds before the pod on that node is evicted and migrated to another node. We changed *tolerationSeconds* to one second in Listing 5.1 to accelerate the failover process.

```
tolerations:
- key: "node.kubernetes.io/not-ready"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 1
- key: "node.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 1
```

Listing 5.1: The configuration for tolerations in the HA experiment.

5.4 Results

This section presents the results of the experiments. Due to the large amounts of results, we will only introduce significant ones here. The results for every single case in detail are listed in Appendix B. The metrics of one, five, seven, and ten appear to be more representative and will get further described in the following subsections.

5.4.1 Response Time

Since all experiments were repeated ten times, we used the median and standard deviation values as the final result in this report. A detailed table with response times from each round can be found in Appendix B. Table 5.1 shows that it takes significantly longer for the VM to start and shut down regardless of the number of desktop instances. Response Time gets longer as the number of concurrent desktop instances increases, especially the starting action of the VMs. We also noticed from the standard deviation that the Response Time of the VM implementation is less stable than that of the Docker implementation.

5.4.2 Resource Consumption

This section summarizes the results of Disk Read/Write Rate, Memory Usage, and Load Average in the form of figures. Each figure contains all the results from the ten repetitions of the experiments to observe if the results fall apart.

Disk Read/Write Rate

Figures 5.1 to 5.4 illustrate how Disk Read/Write Rate performs in the first round of the experiments when starting or stopping ten instances concurrently, for both Docker- and VM-based prototypes. Docker's disk read/write rates can reach higher peaks than VM. For example, the high peak of Docker is outstanding when shutting down the desktop instances, as the value can reach almost 60 MB/s in figure 5.2. However, the duration of the peak is significantly shorter than the VM-based prototype, as it takes much less time for the Docker-based prototype to complete the stopping action.

Start(s)		
Desktop Instances	VM	Docker
1	16.23 ± 0.46	2.94 ± 0.24
2	20.59 ± 1.36	3.45 ± 0.18
3	24.20 ± 1.18	3.65 ± 0.18
4	28.80 ± 1.36	4.00 ± 0.12
5	33.96 ± 2.77	4.17 ± 0.51
6	38.91 ± 3.21	4.50 ± 0.39
7	43.15 ± 4.30	4.69 ± 0.48
8	48.55 ± 4.39	5.57 ± 0.53
9	52.44 ± 4.17	5.76 ± 0.58
10	58.58 ± 5.52	5.94 ± 0.70

Stop(s)		
Desktop Instances	VM	Docker
1	4.77 ±0.41	1.97 ± 0.05
2	5.58 ± 0.53	2.14 ± 0.08
3	6.19 ± 0.19	2.20 ± 0.09
4	6.70 ± 0.20	2.33 ± 0.16
5	7.36 ± 0.22	2.40 ± 0.19
6	7.98 ± 0.51	2.50 ± 0.28
7	8.42 ± 0.32	2.60 ± 0.32
8	9.66 ± 1.34	2.79 ± 0.36
9	10.10 ± 0.64	2.97 ± 0.48
10	11.10 ± 0.54	2.95 ± 0.45

Table 5.1: Table of start and shutdown time in seconds for both prototypes. The median and standard deviation are given.

Furthermore, Figures 5.5 to 5.12 show the ten rounds of experiments in complete for starting/stopping one, five, seven and ten desktop instances simultaneously. In these figures, VM lasts longer on the X-axis because this prototype is slower, as indicated by their Response Time. The peaks in the figures show high Disk Read/Write Rates when the desktop instances in VM/Docker are starting or shutting down. The spaces between the peaks are the five-minute breaks in between. In the VM figures, the higher peaks occur when desktop instances are starting, and a lower peak follows each higher peak because the instances are shutting down. The same pattern persists in all ten repetitions. Nevertheless, this pattern is not as clear for the Docker figures. Disk Read/Write Rate behaves similarly in both actions.

Digging deeper, the read rates of the VM-based prototype become significantly higher as the number of instances increases, while the Docker-based prototype stays roughly the same. As for the write rates, the Docker-based prototype does not perform as stable as its read rates, but it is still much lower than the write rates of the VM-based prototype.

Memory Usage

Figures 5.13-5.16 show a thorough investigation of Memory Usage. Since we ran all the tests on the same machine, the memory usage is identical for both prototypes when idle. Both prototypes behave quite stably during all ten rounds. In the meantime, desktop instances in VMs consume much more memory than that in Containers. Even when ten instances are started simultaneously in containers, the value does not get higher than three GiB while VMs are already approaching eight GiB.

Load Average

The Load Average here refers to the Linux system Load Average [34]. The Load Average results during the experiments are shown in Figures 5.17-5.20. These figures illustrate that the peaks become higher as the number of desktop instances increases. One thing to mention here is that when plotting the metrics, we intentionally dropped the long-term load

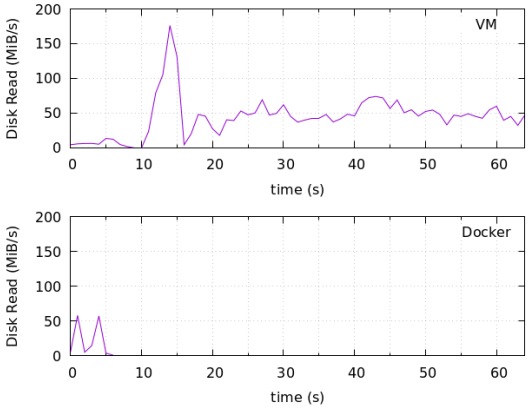


Figure 5.1: Disk Read Rate of starting ten desktop instances

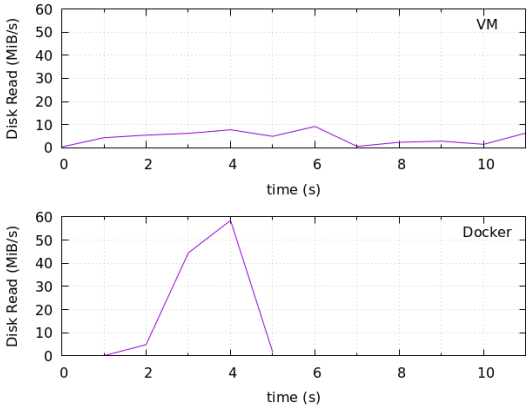


Figure 5.2: Disk Read Rate of stopping ten desktop instances

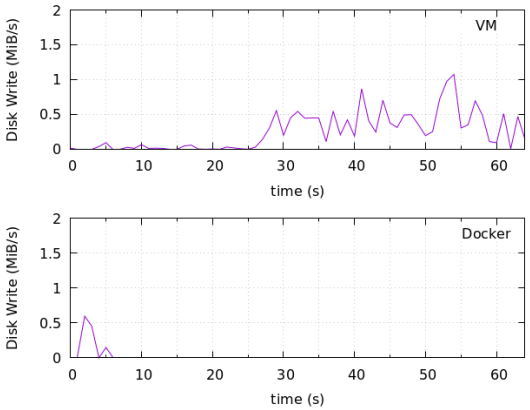


Figure 5.3: Disk Write Rate of starting ten desktop instances

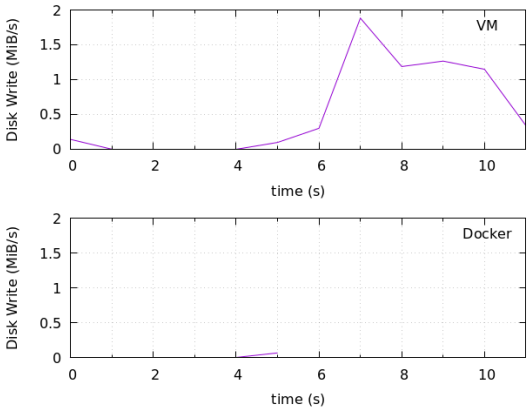


Figure 5.4: Disk Write Rate of stopping ten desktop instances

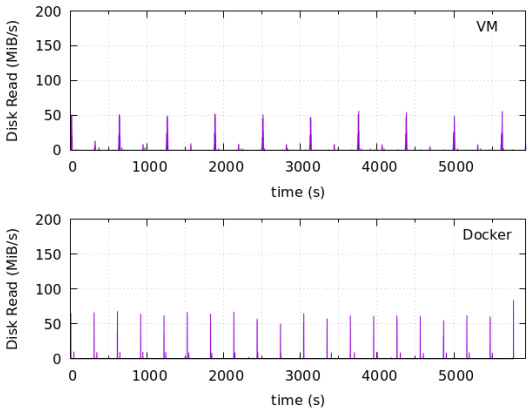


Figure 5.5: Disk Read Rate for one desktop instance starting and stopping ten rounds

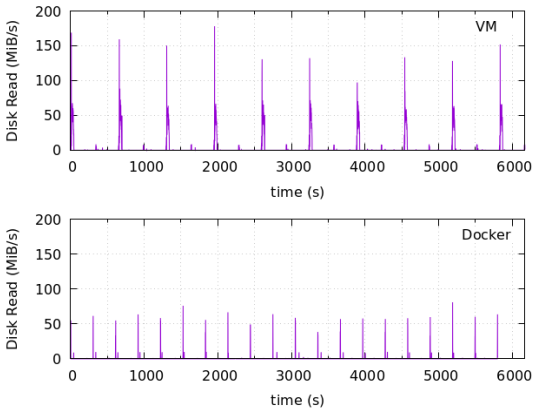


Figure 5.6: Disk Read Rate for five desktop instances starting and stopping ten rounds

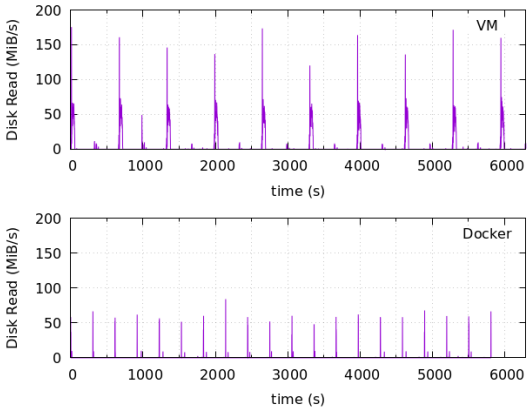


Figure 5.7: Disk Read Rate for seven desktop instances starting and stopping ten rounds

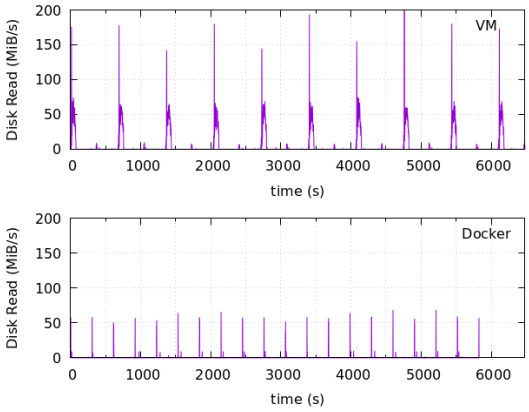


Figure 5.8: Disk Read Rate for ten desktop instances starting and stopping ten rounds

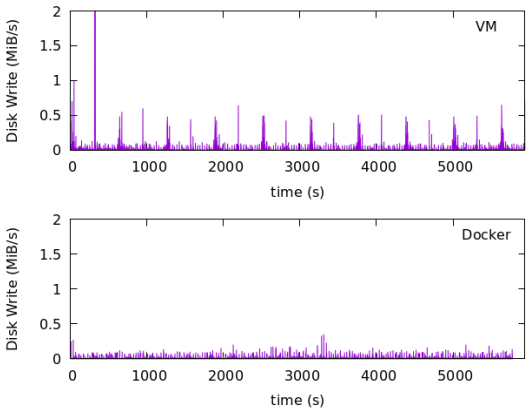


Figure 5.9: Disk Write Rate for one desktop instance starting and stopping ten rounds

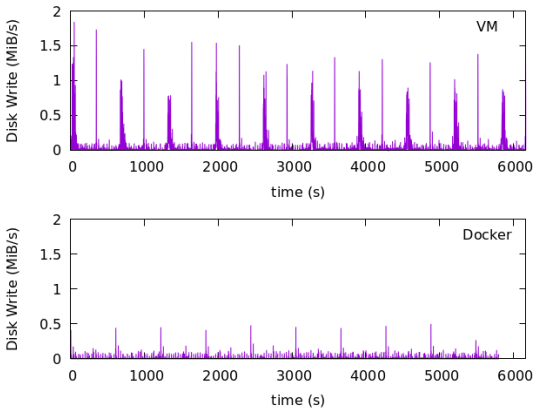


Figure 5.10: Disk Write Rate for five desktop instances starting and stopping ten rounds

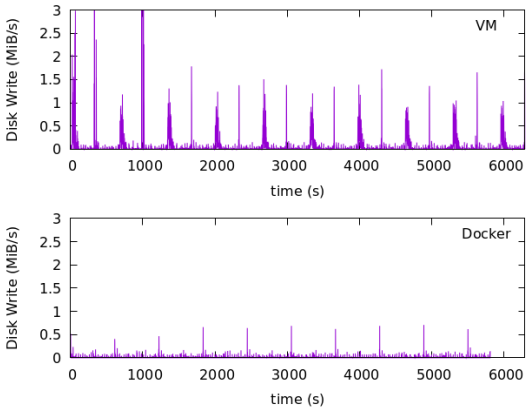


Figure 5.11: Disk Write Rate for seven desktop instances starting and stopping ten rounds

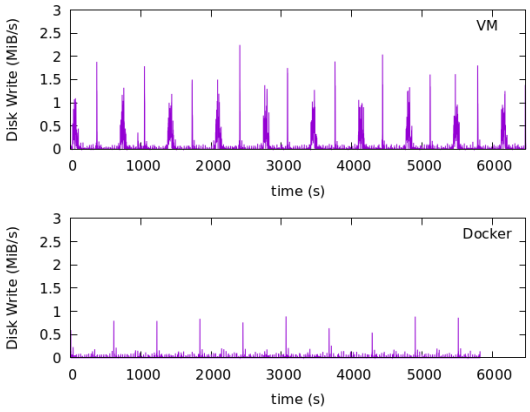


Figure 5.12: Disk Write Rate for ten desktop instances starting and stopping ten rounds

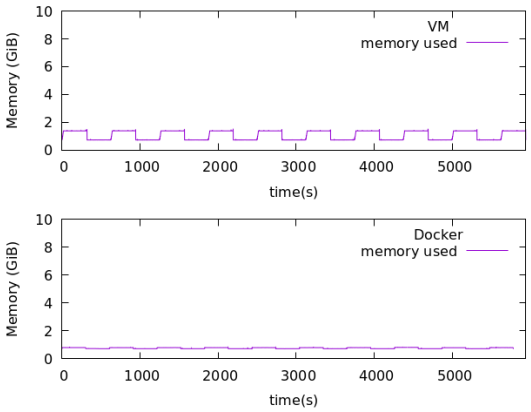


Figure 5.13: Memory Usage for one desktop instance starting and stopping ten rounds

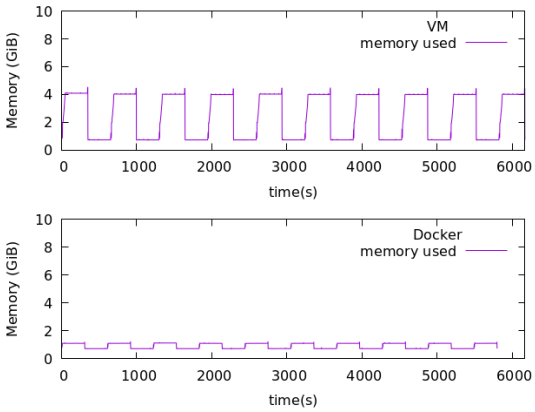


Figure 5.14: Memory Usage for five desktop instances starting and stopping ten rounds

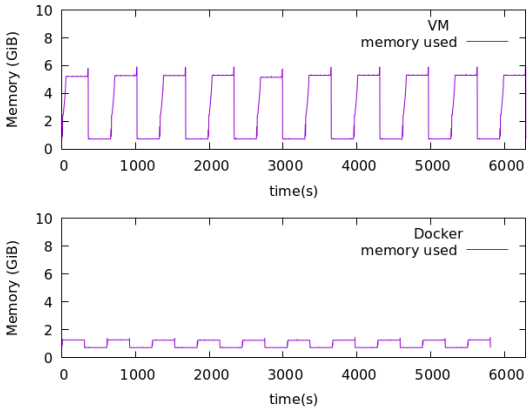


Figure 5.15: Memory Usage for seven desktop instances starting and stopping ten rounds

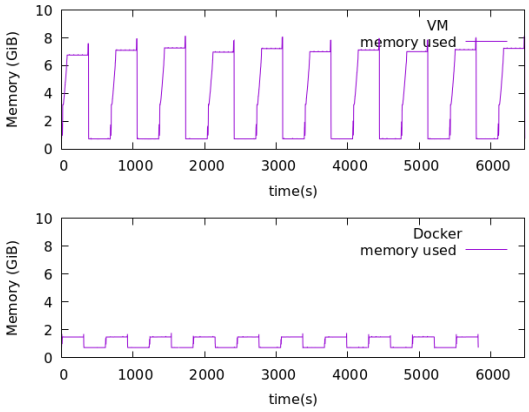


Figure 5.16: Memory Usage for ten desktop instances starting and stopping ten rounds

average because we had five-minute breaks between the start/shutdown actions and another five-minute interval before triggering every next round. The long-term fifteen-minute load average could not reflect the metric well.

Figure 5.17 shows that although the Docker prototype's loads are more stable, both Docker and VM prototypes share the exact peak value of around one. However, the situation changes dramatically when the number of instances reaches five, seven, and ten, as shown in Figures 5.18, 5.19 and 5.20. Due to much more intensive disk utilization, the Load Averages of the VMs go far beyond the containers, whose highest peak value is barely above three, and they can always produce outstanding peaks in every round including both starting and shutdowns. In contrast, for the container-based prototype, where disk utilization is more relaxed, increasing the number of desktop instances to five, seven, and ten may not even cause the startup actions in each round to spike noticeably, not to mention shutdowns, so the average load spikes exhibit some randomness.

Figure 5.21 shows how the load average can behave in a given round by zooming in on one of the rounds when launching ten desktop instances. A high peak occurs when the instances get started, and the Load Average slowly decreases over time. When shutting down the instances, a smaller peak appeared at second 375 for the VM prototype and at second 317 for the Docker prototype on the x-axis, and it too decreased over time. VM and Docker behave similarly on Load Average, while Docker shows significantly lower values.

Storage

In the previous chapter, VM- and Container-based systems have shown very different architectural characteristics, leading to their difference in disk storage. Table 5.2 shows the disk storage of each prototype used for various numbers of instances. While the number of instances increases, the image size and the growth of the Docker prototype are significantly lower compared to the VM prototype, as only the upper part in the overlay of the whole disk usage increases slightly.

	Docker		VM
Desktop instances	Image (GiB)	Overlay (MiB)	Image (GiB)
1	1.182	0.249	4.3
2	1.182	0.459	8.6
3	1.182	0.622	12.9
4	1.182	0.921	17.2
5	1.182	1.180	21.5
6	1.182	1.404	25.8
7	1.182	1.66	30.1
8	1.182	1.871	34.4
9	1.182	2.129	38.7
10	1.182	2.377	43.0

Table 5.2: This is a table of disk space usage of Docker and VM. Only the unit of the overlay is MiB, while all the rest is GiB.

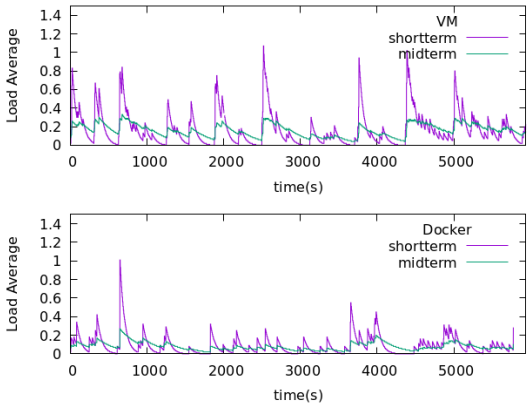


Figure 5.17: Load Average for one desktop instance starting and stopping ten rounds

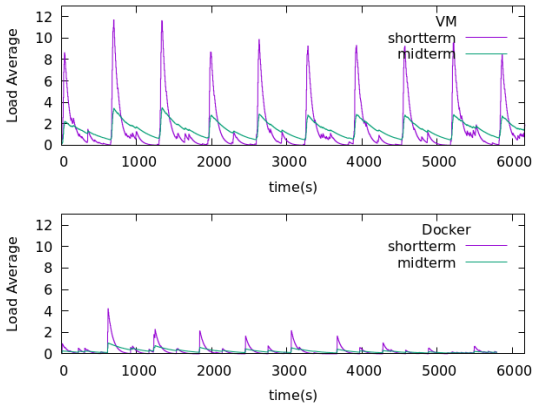


Figure 5.18: Load Average for five desktop instances starting and stopping ten rounds

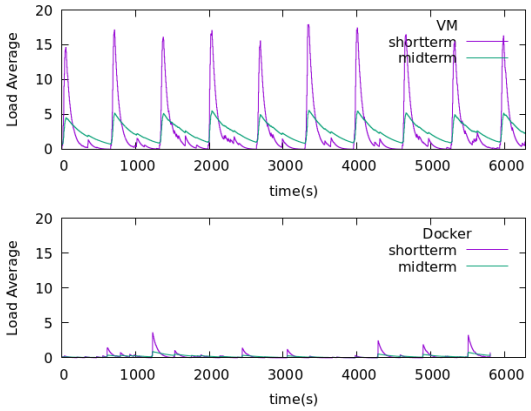


Figure 5.19: Load Average for seven desktop instances starting and stopping ten rounds

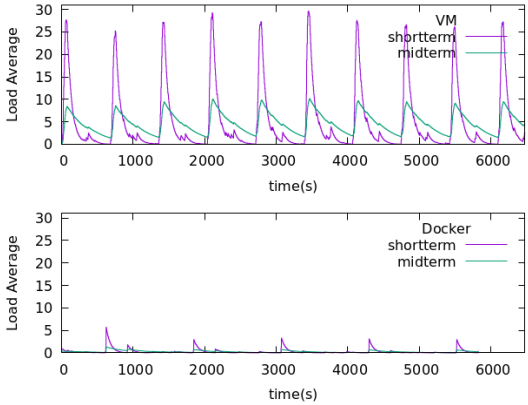


Figure 5.20: Load Average for ten desktop instances starting and stopping ten rounds

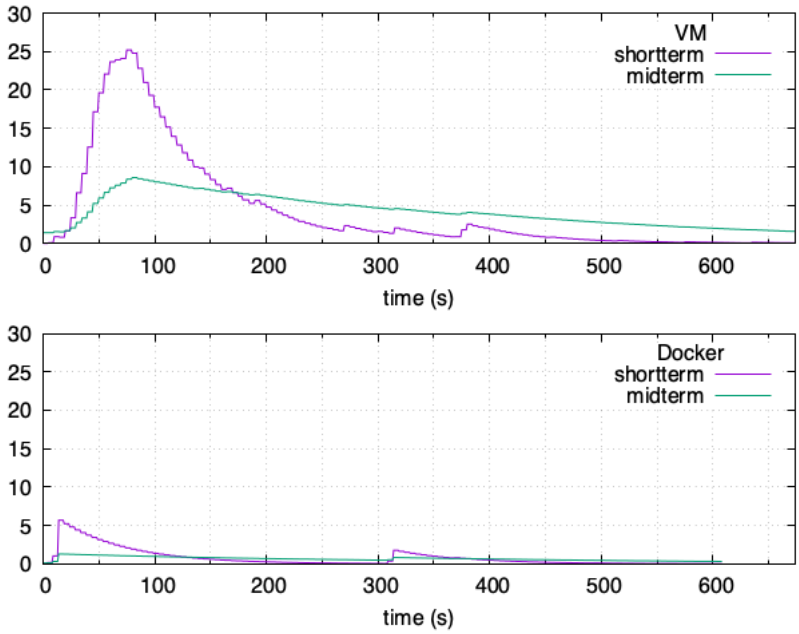


Figure 5.21: Load Average for VM and Docker in one round of starting and stopping ten desktop instances.

5.4.3 High Availability Test

In this section, we do not collect any new metrics. Instead, we only describe the high availability behaviors of the extended implementations of VM and Container in clusters on a PVE Cluster and a Kubernetes cluster.

PVE cluster

In the PVE Cluster, when the node that our test desktop instance was running on went down, the desktop instance became inaccessible immediately. When it was in use, connected via noVNC, the running applications would no longer respond to user operations like mouse clicks or keyboard inputs.

The cluster soon detected the node was down and automatically tried to start the unavailable desktop instance in VM to another one that was still up.

From the perspective of a user, while a missing VM was starting on another node, a loading spinner soon showed up on the noVNC web page and eventually raised a "connection timeout" error if the user tried to reconnect or refresh the noVNC page during this time. When the network stack was up in the newly started VM while the noVNC service was not ready, the user would see a "connection refused" error instead. Soon after the error, another refresh of the page brought back the desktop screen in the web browser.

In our experiment, we carried out the HA experiments ten times to collect the time consumption of the failovers. It took 134.50±4.35 seconds for our PVE Cluster to detect the node failure and bring the desktop instance until its noVNC service back online on another node alive. This value represents the median and standard deviation.

Kubernetes

We also carried out a failover test in the Kubernetes cluster. As mentioned in the previous chapter, we applied the YAML file to deploy a desktop instance to the cluster and could see that the instance was living on one of the worker nodes – *node4*:

```
$ kubectl get pods -o wide
NAME                       READY   STATUS    RESTARTS   AGE   IP             NODE
desktop-6bdf96c5b9-qblfl   1/1     Running   0          17s   10.233.105.7   node4
```

Listing 5.2: The status of running pods when there was one living desktop.

Then we shut down *node4*, and *kubectl get nodes* showed that the node was in the *NotReady* state:

```
$ kubectl get nodes
NAME    STATUS     ROLES                  AGE   VERSION
node1   Ready      control-plane,master   9d    v1.21.3
node2   Ready      control-plane,master   9d    v1.21.3
node3   Ready      <none>                 9d    v1.21.3
node4   NotReady   <none>                 9d    v1.21.3
```

Listing 5.3: The status of all nodes when one node was shut down.

In the meantime, the Control Plane detected the number of living instances of the deployment *desktop* (0) was below the desired value (1) and started to spin it up on the remaining node *node3*, meaning a failover was done.

```
$ kubectl get pods -o wide
NAME                       READY   STATUS              RESTARTS   AGE     IP             NODE
desktop-6bdf96c5b9-hbcfz   0/1     ContainerCreating   0          4s      <none>         node3
desktop-6bdf96c5b9-qblfl   1/1     Terminating         0          3m13s   10.233.105.7   node4
```

Listing 5.4: The status of the desktop pods after the node it was scheduled was shut down.

On the client-side, a user would see a different error compared to the PVE Cluster implementation. When *node4* went down, the user would see the loading spinner and a frozen desktop as well. Nevertheless, when the user refreshed the page in the web browser, a "503 Service Temporarily Unavailable" error appeared instead of the "connection refused" error we witnessed in the PVE Cluster implementation. The reason is that the noVNC service in our Kubernetes Cluster was behind Ingress, which would not be down due to a node failure because it had its own HA mechanism in Kubernetes. When the service behind Ingress became unavailable, the 503 error would always show up. When the desktop service recovered in a container running on *node3*, the 503 error would disappear.

Upon conducting ten repeated experiments, it was determined that the availability of the *desktop* instance was restored within a median time frame of 45.0 seconds, with a standard deviation of 4.0 seconds. The desktop screen was back in the web browser after another refresh on the web page, but the service was running on a different node.

Chapter 6
Discussion

This chapter discusses the results from the previous chapter, including infrastructural and non-infrastructural metrics. Then we compare PVE Cluster and Kubernetes for high availability purposes, generally first and then specifically for DaaS. We also mention some existing commercial DaaS products on the market in this part. In the end, we also elaborate on limitations and possible pitfalls that could potentially have been solved or done better.

6.1 Evaluation on Metrics

As collected in the previous chapter, we discuss the two types of metrics here, including non-infrastructural and infrastructural metrics. Response Time is the only non-infrastructural metric, while infrastructural metrics consist of Disk Read/Write Rates, Memory usage, Load Average, and Disk Space usage.

6.1.1 Response Time

Chapter 5 shows desktop instances powered by containers are significantly faster than those in VMs on starting up and shutting down. The feature of the Container technology that it can reuse the kernel of its host has strongly influenced this result, while each VM has to boot up its own kernel and OS. The difference in response times becomes more prominent as the number of instances increases. Despite this increase in response time, the standard deviation remains relatively small in comparison, indicating a stable service provided by both technologies.

After conducting failover tests in both prototypes, it was determined that Proxmox VE Cluster had a median recovery time of more than two minutes while Kubernetes had a median recovery time of less than one minute. However, according to our results, the median startup time of one desktop takes only 16 and 3 seconds for VM and Docker. In the case of Kubernetes, an adjustable variable, node-monitor-grace-period, allows a node to be unresponsive before a

failover begins. The default value of this variable is 40 seconds, resulting in a failover finished between 43 and 54 seconds. A detailed timetable of all ten rounds can be found in Appendix B. In the case of the Proxmox VE Cluster, we were unable to discover which variables were affecting the basis of the recovery time, which was always longer than two minutes. However, we found a statement on their Wiki page that the error detection and failover typically take about 2 minutes [21].

6.1.2 Infrastructural Metrics

The results are similar among Disk Read/Write Rate, Load Average and Disk Space usage. The Container prototype is more resource-efficient than the VM prototype. When hosting the same number of desktop instances, containers bring lower pressure on disk drives and CPUs than VMs. With the current physical machines, we can host more desktops if we switch the underlying technology from VM to Container. This advantage is especially significant regarding Disk Space usage. As shown in Table 5.2, the Container prototype naturally uses less disk space because all the desktop instances reuse the same image as the base. While the desktop instances are running, the additional usage is from their overlays. As for the VM prototype, its base image already consumes much more disk space. Although we could trim the unused components and packages to generate a slimmer base image for desktop instances, the image has to contain at least the files required to boot a Linux system, such as *initramfs* and the *kernel*. We know that COW (Copy-On-Write), supported by Qemu/KVM, and the disk image format *qcow2*, can achieve a similar result to the overlay filesystem Docker benefits from, but PVE does not support this feature yet. Nevertheless, even though PVE might add this feature someday, it still does not help much because of the size of Linux system files.

6.2 Evaluation on Cluster Solutions for High Availability

We also compared two cluster solutions extended from the VM- and Container-based prototypes in the previous chapter – PVE Cluster and Kubernetes. They are not the only choices for DaaS with High Availability. There are also existing options from both commercial companies and the open-source community.

6.2.1 PVE cluster vs. Kubernetes

In a PVE cluster, the minimum number of nodes is three to achieve a reliable quorum. One node in the cluster plays an equivalent role to another. A PVE cluster does not have a differentiation between the Control Plane and the Worker node as in Kubernetes. Also, in practice, control plane nodes will not run any workloads other than the core Kubernetes services that back the cluster. Furthermore, to ensure the control plane is not a single point of failure, one Kubernetes cluster should have at least two control plane nodes, and so do the worker nodes – that sums up to the minimum number of nodes in a Kubernetes cluster – four.

We are not saying four is worse than three. The difference comes from the system designs of the two solutions. It is usually not a concern in a production environment either because

it does not matter whether a large cluster already consisting of many more machines has one more node or one fewer node. Also, control plane nodes do not have to be as powerful as the worker nodes; they can even be cloud servers or VMs guarded by more robust solutions with high availability too.

Regarding the complexity of using Proxmox VE Cluster and Kubernetes for DaaS, PVE has a powerful WebUI that has implemented the most common maintenance tasks. One can create a cluster, add or remove a node, or quickly set up shared storage on the WebUI. Its API and CLI are also intuitive to use. In Kubernetes, these maintenance tasks have to carry out differently. Setting up and modifying the cluster requires *Kubespray* that involves knowledge about *Ansible*, and deploying desktop instances needs to compose a YAML file. There are many more technical concepts and architectural expertise to learn to drive a Kubernetes cluster smoothly. The learning curve, therefore, becomes steeper. On the other hand, Kubernetes is highly customizable and powerful for integrating with other systems and automation. Scaling out for Kubernetes is more cost-effective and better supports servers with diverse hardware specifications.

There are quite a few features of Kubernetes that suit DaaS well. Probes and Deployment contribute to high availability, and Ingress takes care of load balancing and TLS termination. As a result, we do not have to develop and maintain these critical supportive components in DaaS from scratch.

6.2.2 Existing Cluster Solutions for DaaS

In the early phase of our project, we comprehensively examined existing tools and solutions for DaaS together with engineers from Yark Network. One notable example is the Horizon platform by VMWare [46], which has been widely adopted by DaaS providers catering to both large and small enterprises. However, the existing products are proprietary and can be complex to integrate into the current system in Yark Network, which does not have Active Directory or Smart Card Authentication support [48]. Therefore, we opted to retain the extant VIM solution, Proxmox VE, in conjunction with its associated cluster solution, PVE Cluster. In addition, Horizon is still powered by VM technology, while Container is what we are heading for. OpenStack is another choice that is even more powerful and convenient if the organization needs other capabilities the OpenStack components [40] can provide, for example, the AWS S3-compatible Object Store, *Swift*. OpenStack also has a component to manage containers – *Zun*. However, Kubernetes has become more actively developed and widely adopted in the past few years, while Zun has not been recognized much. As of January 2023, the keyword "Kubernetes" has about 142 million results on Google, while "openstack zun" has 0.34 million only. On GitHub, the Kubernetes project [1] has 95.1k stars, while Zun [2] has 81 stars only.

6.3 Limitations

Due to the limited knowledge and resources, it is inevitable for this report to have remaining unsolved problems and possible pitfalls. We strived to sort them out or address them such as dropping the system cache to collect actual disk read rate, but there the following ones remain.

6.3.1 Occasional Blockage During Experimentation

In the early phase of the experimentation, we noticed that the scripts in section 5.3.1 we used to repeatedly start/stop desktop instances got stuck when starting many instances in VMs. To debug and verify our constantly optimized experimental scripts, we executed them a dozen or more times, meaning over two thousand rounds of starting/shutdowns in total, but the issue was observed only twice. One of the two was when we were starting six desktop instances, while the other case happened on seven. When such an issue happened, we had to terminate the script and kill the hanging VMs to continue. Regarding the root cause, we have not found it yet. However, it is necessary to point out that our experiments were in an uncommon situation where up to ten desktop instances started and stopped concurrently on a single node, bringing race-condition and high pressure to the server. This extreme scenario will not happen in a clustered DaaS in production. The company will not allow so many users to start or stop desktop instances in a very short moment without a queuing system or a scheduler. The capacity of the DaaS will also be scaled up to meet the increase in its number of users. This issue did not hit the Container prototype, though. Furthermore, if this were to occur in Kubernetes, the scheduler could automatically restart a stuck instance when the liveness and readiness probes [27] are defined.

6.3.2 Accuracy in Metric Collection

Regarding the accuracy of the metrics we collected, we have seen that in the VM case, the first several desktop instances have larger disk images than the remaining ones. The reason is that those VMs were started more times than the rest; therefore, more logs were written into their virtual disks. We should have considered this in the experiments and devised a better way to avoid it, for example, to always start with copying from the template image before each round. However, the mitigation can significantly impact the overall complexity of the experiment design, and each round may take much longer to finish.

In terms of Load Average, the peaks caused by starting and stopping desktop instances are not as apparent as other metrics, especially in the Container prototype. One thing to point out here is that in our experiments, since the interval between two rounds was five minutes, we ditched the Load Average of fifteen minutes. Unlike the other real-time metrics, the peaks in the results we collected were already average and flattened, therefore abrupt changes in a short time might be simply missing.

Some system services and operations of the operating system itself may also interfere with the accuracy of the final results when performing the experiments. For example, since PVE runs on top of Debian Linux, a cron job is there to update the apt cache periodically. This cron job could have brought additional disk reads/writes to the experiment. This type of influential factor may also explain the abnormal spikes in the Disk Read/Write Rate results. Let's take Figure B.1b as an example. The VM prototype had two spikes that went beyond the maximal value of the y-axis, even in the first two rounds. In the same figure, the Container prototype showed an abrupt spike in the middle, while it constantly had low and stable disk writes other than that. Similar issues happened to Figure B.1d and more.

We used to store the metric values on the same server where the experiments ran, which resulted in constant disk writes, so we changed *collectd* to send those values to a different machine. We managed to exclude as many influence facts as possible, but there were still

some issues we could not fully address or could not fully avoid. Supportive PVE daemons to make the platform fully functional were among them [22].

Additionally, we could have repeated the experiments that we did to collect the metrics for more times, because ten times might be insuffient to ensure the data accuracy as high as possible.

6.3.3 Unable to change the minimal failover time in PVE Cluster

As observed in the PVE Cluster HA experiment, the failover of a desktop instance always took more than two minutes. This is also confirmed by documentation [21] on the wiki page of Proxmox VE mentioned earlier in this chapter. We wanted to change this value to verify if it was the case or change it to 40 seconds to match the equivalent setting in Kubernetes. However, digging into both documentation and historical posts in the user community, we failed to discover where and how to change it. We also posted a question in the ProxMox Forum, but to date, no response has been received.

6.3.4 Lack of Metrics Collection in the Clustered Deployments

It would be interesting to see the response times and the infrastructural metrics in the two HA solutions in clusters. However, we did not conduct the data collection again as we did in the non-cluster comparison.

Chapter 7
Conclusion

This thesis compares DaaS prototypes implemented with VM and Container as the underlying technologies. The comparison parameters are Response Time and infrastructural metrics, including Disk Read/Write Rate, Memory Usage, Load Average, and Disk Space Usage. The VM prototype runs on a Virtual Infrastructure Manager (VIM) platform Proxmox VE (PVE). The Container prototype implemented with Docker runs on the same machine. One to ten desktop instances were started and stopped concurrently by scripts with five-minute intervals between every start or stop action. These experiments were repeated ten times for each number of instances to eliminate measurement errors. The timestamps of those actions were logged within the scripts to log files for Response Time calculation. At the same time, the infrastructural metrics were recorded by a monitoring system *collectd* in a format named Round Robin Database (RRD).

The high availability (HA) solutions based on clusters built with the two technologies were also tested and observed. PVE cluster for VM, and Kubernetes for Container, were employed to reveal how high availability of DaaS performs by the two technologies. In both clusters, one of the worker nodes was shut down to simulate a common node failure. Both cluster solutions did their work well to achieve HA. The desktop instances that ran on the failed node recovered automatically on another worker node that was still alive.

7.1 Answers to Research Questions

RQ1: What are the impacts on the response time and resource usage of container-based DaaS compared to the VM-based DaaS?

Compared to the VM-based prototype, we found that the desktop instances provided by the Container prototype responded faster on both starting and shutting down actions. The more instances we tested in parallel, the larger the deviation we observed in Response Time. The Container-based DaaS also beat the VM-based DaaS in resource usage. According to our monitoring data, the former had a lower Load Average, less Memory Usage, and much

less Disk Space Usage. Regarding Disk Read/Write Rate, although the Container-based prototype did not necessarily infer lower utilization of disk IO, the reads and writes on the disk drives were more stable and lasted shorter, as the starting and shutting down actions finished earlier than the VM-based prototype. As a result, the Container technology reduces the pressure on disk drives.

RQ2: What is the impact on the availability of DaaS when the VM-based and Container-based prototypes are extended to clusters?

Clustering brings high availability to both VM and Container technologies. On the one hand, PVE Cluster with shared storage may achieve similar high availability as Kubernetes. When a VM-based desktop instance became unreachable due to a node failure, it was soon detected and recovered automatically to another healthy node in the PVE cluster. However, Kubernetes is better at this. A failover could take approximately one-third of the PVE cluster's time – the latter took over one minute in our experiment. In addition, Kubernetes has more built-in components that could help in the DaaS scenario. For example, *Probes* better maintain the availability of desktop instances, and Ingress can take care of load balancing and TLS termination for the exposed noVNC service. Although Kubernetes has more features and advantages, it is more difficult to learn its concepts and maintain its setup, which might result in higher human resource costs in the end. PVE Cluster has a user-friendly Web UI and is more straightforward to interact with. What is more, Kubernetes cannot support Microsoft Windows instances yet, which could be a deal-breaker for some customers that rely on Windows-only software.

7.2 Future work

Due to the COVID-19 pandemic, many companies encourage employees to work from home. Some companies even allow employees to work from home for an indefinite time. It is an opportunity for us to explore the possibility of DaaS. However, as more and more companies start to adopt DaaS, a gap in the DaaS we have at hand is the absence of Microsoft Windows (MS Windows). MS Windows is a popular desktop operating system; though the number of Windows users in companies has reduced over the last decade, over 70% of the global market still holds Windows as the OS for their PC [43]. The current prototypes we implement in this report are not compatible with MS Windows yet. Even though Microsoft has been working on Windows Container, it is not available for desktop environments but primarily for server applications without a graphical user interface (GUI). Moreover, those Windows Containers have to run on the same Windows Server version except Windows Server 2022 and Windows 11 [36] because they run on top of the same kernel of the host system [35]. We can try to support Windows applications with solutions like Wine [49] in our DaaS for now as a workaround. Still, in the meantime, we should keep our eyes open to the updates of the Windows Container to keep up with its development. We can also provide Container images with alternative software running in Linux for particular applications wherever possible.

The user experience of DaaS is not only limited to startup/shutdown time but also related to the desktop's responsiveness and functionalities. Responsiveness here refers to whether user interactions and the video refresh are smooth. As mentioned by A. Celesti et al. in their work [5], RDP performs the best, while noVNC is the worst. As for functionalities, neither of our DaaS implementations supports all the common features a desktop user might want. For

example, VNC/noVNC has no native audio redirection, and our current version of noVNC does not even support a clipboard. Newer versions of noVNC support it, but it is still not as convenient as the native copy-and-paste experience. In the future, we might provide more connection protocol options such as RDP for end-users to deliver a better desktop experience.

The lower startup/shutdown time of Container-based DaaS can be helpful to desktop software development as well. In particular, end-to-end testing is a scenario where Response Time plays an important role. Imagine a new desktop software release that requires end-to-end testing on a matrix of desktop environments. Not only different operating systems (OS) like MS Windows and Linux but also different versions of the OSes are involved. MS Windows alone has a lot of versions, such as Windows 7, Windows 10, and the Windows Server family. Moreover, Linux has many distributions like Ubuntu, Debian, Fedora, CentOS, etc. End-to-end testing on such a matrix could take a long time when running on a VM-based DaaS because the startup/shutdown actions are already quite time-consuming. Also, thanks to the OverlayFS [4], deriving docker images from the same base image for similar testing environments can be done with less time, disk space, and human effort. In the meantime, maintaining many docker images for different OSes could also be challenging. We might want to figure out a solution to build the docker images and keep all of them up-to-date in an automated way.

The test machines we used in this report are all equipped with Hard Disk Drives (HDDs), while Solid State Drives (SSDs) have become increasingly popular over the past years. SSDs can be much faster than HDDs in both sequential and random access workloads [26], and SSD's price per TiB has been approaching HDDs. Floyer [17] estimated that in the year 2026, SSD would be cheaper than HDD. Since SSDs will still be more expensive than HDDs for the time being, another interim option is hybrid storage. Hybrid storage is built with HDDs for infrequently accessed data (Cold Files) and SSDs for frequently accessed data like VM/Container images (Hot Files). With a higher price-performance ratio, hybrid storage might be a favorable solution for now.

References

[1] kubernetes. https://github.com/kubernetes/kubernetes. (last visited 2023-01-26).

[2] zun. https://github.com/openstack/zun. (last visited 2023-01-26).

[3] Ansible. How ansible works. https://www.ansible.com/overview/how-ansible-works. (last visited 2022-11-02).

[4] N. Brown. Overlay filesystem. https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html. (last visited 2022-11-02).

[5] A. Celesti, D. Mulfari, M. Fazio, M. Villari, and A. Puliafito. Improving desktop as a service in openstack. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 281–288, 2016.

[6] Ceph. Ceph. the future of storage. https://ceph.io/en/discover/. (last visited 2020-10-15).

[7] Citrix. What is daas? https://www.citrix.com/solutions/vdi-and-daas/what-is-desktop-as-a-service-daas.html. (last visited 2023-04-07).

[8] Google Cloud. What are containers? https://cloud.google.com/learn/what-are-containers. (last visited 2023-04-07).

[9] collectd. collectd – the system statistics collection daemon. https://collectd.org/. (last visited 2022-11-15).

[10] ConSol. Docker container images with "headless" vnc session, January 2019. https://github.com/ConSol/docker-headless-vnc-container (last visited 2022-11-02).

[11] S. Dernbecher. Having the mind in the cloud: Organizational mindfulness and the successful use of desktop as a service. In *2014 47th Hawaii International Conference on System Sciences*, pages 2137–2146, 2014.

[12] M. Dhall and Q. Tan. A profitable hybrid desktop as a service solution. In *2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, pages 55–62, 2019.

[13] C. Dilmegani. Data as a service (daas): What, why, how, use cases & tools, April 2020. `https://research.aimultiple.com/data-as-a-service/`. (last visited 2023-04-07).

[14] Docker. *Install Docker Engine on Debian*. `https://docs.docker.com/engine/install/debian/`. (last visited 2022-10-02).

[15] Docker. Use containers to build, share and run your applications. `https://www.docker.com/resources/what-container/`. (last visited 2023-04-07).

[16] E. Dubrova. *Fault-Tolerant Design*. Springer New York, NY, Cheshire, CT, USA, 2013.

[17] D. Floyer. Qlc flash hamrs hdd, January 2011. `https://wikibon.com/qlc-flash-hamrs-hdd`. (last visited 2022-11-02).

[18] Proxmox Server Solutions Gmbh. Features. `https://proxmox.com/en/proxmox-ve/features`(last visited 2022-11-05).

[19] Proxmox Server Solutions Gmbh. Qemu/kvm virtual machines. `https://pve.proxmox.com/wiki/Qemu/KVM_Virtual_Machines`. (last visited 2022-11-02).

[20] Proxmox Server Solutions Gmbh. *Installation*, November 2021. `https://pve.proxmox.com/wiki/Installation`. (last visited 2022-11-02).

[21] Proxmox Server Solutions Gmbh. High availability, May 2022. `https://pve.proxmox.com/wiki/High_Availability`. (last visited 2022-11-02).

[22] Proxmox Server Solutions Gmbh. Proxmox ve administration guide, May 2022. `https://pve.proxmox.com/pve-docs/pve-admin-guide.html#_important_service_daemons`. (last visited 2022-11-02).

[23] A. A. Z. A. Ibrahim, D. Kliazovich, P. Bouvry, and A. Oleksiak. Virtual desktop infrastructures: Architecture, survey and green aspects proof of concept. In *2016 Seventh International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, 2016.

[24] Docker Inc. Use the overlayfs storage driver. `https://docs.docker.com/storage/storagedriver/overlayfs-driver/`. (last visited 2022-11-02).

[25] Open Container Initiative. Open container initiative. `https://opencontainers.org`. (last visited 2022-11-02).

[26] V. Kasavajhala. Solid state drive vs. hard disk drive price and performance study. White paper, Dell PowerVault Technical Marketing, May 2011. `https://i.dell.com/sites/csdocuments/Shared-Content_data-Sheets_Documents/en/ssd_vs_hdd_price_and_performance_study.pdf`. (last visited 2022-11-02).

[27] Kubernetes. *Configure Liveness, Readiness and Startup Probes*. https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/. (last visited 2022-11-02).

[28] Kubernetes. Container runtimes, September 2022. https://kubernetes.io/docs/setup/production-environment/container-runtimes/. (last visited 2022-11-05).

[29] Kubernetes. Kubernetes components, October 2022. https://kubernetes.io/docs/concepts/overview/components/. (last visited 2022-11-05).

[30] Kubernetes. Taints and tolerations, October 2022. https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/. (last visited 2023-01-31).

[31] Kubernetes. What is kubernetes?, October 2022. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. (last visited 2022-11-02).

[32] kubespray. *Deploy a Production Ready Kubernetes Cluster*. https://kubespray.io/#/ (last visited 2022-11-02).

[33] KVM. Kernel virtual machine. https://www.linux-kvm.org/page/Main_Page. (last visited 2022-11-02).

[34] Linux. *proc(5) - Linux manual page*, August 2021. https://man7.org/linux/man-pages/man5/proc.5.html. (last visited 2022-11-02).

[35] Microsoft. Containers vs. virtual machines, October 2021. https://learn.microsoft.com/en-gb/virtualization/windowscontainers/about/containers-vs-vm. (last visited 2022-11-02).

[36] Microsoft. Windows container version compatibility, September 2022. https://learn.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/version-compatibility?tabs=windows-server-2022%2Cwindows-11. (last visited 2022-11-02).

[37] P. H. Nakhai and N. B. Anuar. Performance evaluation of virtual desktop operating systems in virtual desktop infrastructure. In *2017 IEEE Conference on Application, Information and Network Security (AINS)*, pages 105–110, 2017.

[38] noVNC core team. novnc. https://novnc.com/info.html. (last visited 2020-10-15).

[39] T. Oetiker. About rrdtool, February 2017. https://oss.oetiker.ch/rrdtool/. (last visited 2022-11-07).

[40] OpenStack. Openstack services. https://www.openstack.org/software/project-navigator/openstack-components#openstack-services. (last visited 2022-11-02).

[41] QEMU. About qemu. https://www.qemu.org/docs/master/about/index.html. (last visited 2022-11-02).

[42] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[43] Statista. Global market share held by operating systems for desktop pcs, from january 2013 to june 2022, July 2022. https://www.statista.com/statistics/218089/global-market-share-of-windows-7/. (last visited 2020-10-02).

[44] Ubuntu. Vnc, October 2012. https://help.ubuntu.com/community/VNC?action=show&redirect=VNCOverSSH. (last visited 2022-11-08).

[45] Ubuntu. *Installation/MinimalCD*, February 2022. https://help.ubuntu.com/community/Installation/MinimalCD. (last visited 2022-11-02).

[46] VMware. Vmware horizon. https://www.vmware.com/products/horizon.html. (last visited 2023-01-26).

[47] VMware. What is daas (desktop as a service)? https://www.vmware.com/topics/glossary/content/desktop-as-a-service.html. (last visited 2023-04-07).

[48] VMware. *Choosing a User Authentication Method*, April 2020. https://docs.vmware.com/en/VMware-Horizon/2203/horizon-architecture-planning/GUID-8685B14D-868E-4CB5-BF13-D00B10B63E2A.html. (last visited 2022-11-02).

[49] Wine. What is wine. https://www.winehq.org. (last visited 2022-11-02).

[50] Xfce. Xfce 4.18 documentation. https://docs.xfce.org/start. (last visited 2022-11-13).

[51] YAML. Yaml: Yaml ain't markup language™. https://yaml.org. (last visited 2022-11-02).

[52] B. Đorđević, V. Timčenko, S. Savić, and N. Davidović. Comparing hypervisor virtualization performance with the example of citrix hypervisor (xenserver) and microsoft hyper-v. In *2020 19th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pages 1–6, 2020.

Appendices

Appendix A
Hardware Specification

This appendix presents the hardware specification of the physical machine on which prototypes implemented with Docker and VM are running.

- CPU: Intel E5-1620 v2 @ 3.70 GHz (4 cores, 8 threads)
- RAM: 64 GiB
- HDD: 2 TiB x4 (Software RAID10)
- NIC: Intel I350 Gigabit Ethernet @ 1Gbps

Appendix B
Experimental Results

This appendix contains the results of all the experiments that have been performed, including some of the figures already mentioned in Chapter 5. Firstly, four tables of response time are given, where values are in milliseconds. Then the infrastructural metrics and HA failover time are shown. The infrastructural results are presented in the following order:

- Disk Read Rate (Figure B.1)
- Disk Write Rate (Figure B.2)
- Memory Usage (Figure B.3)
- Load Average (Figure B.4)

Each subfigure represents the result of ten consecutive rounds. We plot the figures in this way to better visualize the stability of the two prototypes. In all figures, the x-axis is the time in seconds and the y-axis represents the resulting value. We would like to point out here that the maximal values in the y-axis may vary among figures for clarity.

B.1 Response time

	Number instances (Docker Start)									
Round	1	2	3	4	5	6	7	8	9	10
1	2908	3484	3696	3807	4074	5634	5306	5696	6149	5956
2	2983	3441	3706	4616	5006	4239	5687	5747	5909	6402
3	2931	3648	3648	4151	5141	5008	5518	5895	5977	6150
4	2945	3327	3375	3850	3933	4538	5620	5716	5902	6193
5	2917	3466	3524	4833	5706	4546	4722	5965	6022	6066
6	3673	3380	3645	4021	4769	4405	5584	5803	5810	6142
7	2937	3401	3968	7758	4271	4646	5438	5742	5899	7355
8	3084	3525	3682	3998	5112	4529	4714	5765	5958	5924
9	2798	3264	3594	4817	3937	4361	5364	5764	6526	6398
10	3104	3903	3974	3928	4284	4266	5322	6034	5716	6305

Table B.1: The time in milliseconds that a certain amount of Docker-based desktop instances took to boot up.

	Number instances (Docker Shutdown)									
Round	1	2	3	4	5	6	7	8	9	10
1	1977	2248	2328	2553	2543	2712	2932	2951	3302	3559
2	2068	2154	2203	2486	2566	2802	2911	3219	3504	3411
3	1910	2185	2337	2354	2627	2571	2973	3172	3418	3417
4	1925	2009	2245	2428	2550	2708	2759	3507	3268	3552
5	1973	2096	2390	2476	2563	2784	2880	3125	3438	3506
6	1915	2193	2318	2434	2424	3279	2826	3074	3248	3401
7	1957	2065	2177	2372	2622	2759	3365	3397	4180	3620
8	1984	2167	2261	2497	2543	2728	2876	3200	3343	3502
9	1950	2177	2300	2476	2631	2833	2873	3179	3497	3476
10	1975	2223	2240	2394	2592	2692	3087	3115	3314	3484

Table B.2: The time in milliseconds that a certain amount of Docker-based desktop instances took to shut down.

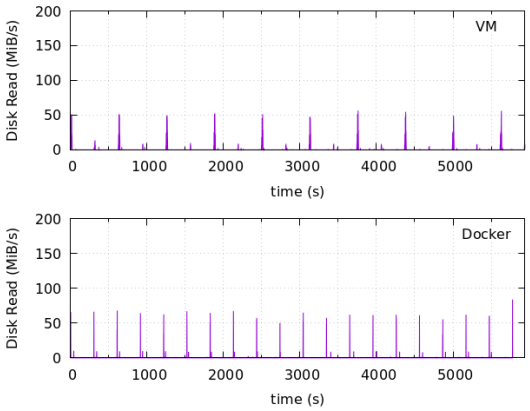
	Number instances (VM Start)									
Round	1	2	3	4	5	6	7	8	9	10
1	16108	25485	28054	30757	41561	45079	49238	53233	56660	64200
2	15808	20026	24325	29859	34461	42179	47567	60143	57179	63604
3	16916	20997	24299	30907	37962	41775	47598	51470	54850	64800
4	15882	21109	25248	29224	35367	41778	45399	52330	56588	63400
5	16158	22858	23175	31456	36163	42480	44587	54116	56660	63805
6	16309	21092	25652	28664	35455	40625	48171	52605	56051	62651
7	16683	20491	24819	30447	35420	42430	46573	52236	58685	61768
8	16150	22003	25181	28804	37846	42510	48436	53290	58551	62052
9	17025	20784	24317	30085	37221	41702	49006	50651	56949	62948
10	16962	20018	26142	29786	36363	41651	45701	51830	56721	63410

Table B.3: The time in milliseconds that a certain amount of VM-based desktop instances took to boot up.

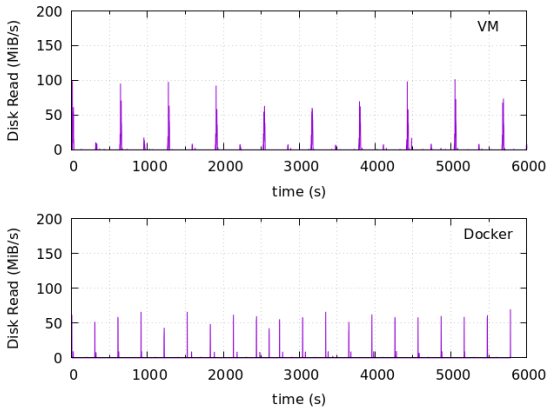
	Number instances (VM Shutdown)									
Round	1	2	3	4	5	6	7	8	9	10
1	5665	5811	6508	7089	7378	8703	9417	10000	12169	11786
2	4406	7260	6301	6964	7437	9583	8481	9389	10696	11480
3	5269	5632	6017	6485	7714	8124	8521	9309	10533	12055
4	4616	5320	6132	6693	7209	7958	8601	10114	10869	11493
5	4681	5543	6175	6509	7369	7825	8734	9714	10023	11090
6	4803	5458	6175	6694	7387	8074	8620	9987	10731	11594
7	4633	5388	6175	6745	7429	7738	8725	9867	10104	11702
8	5471	5658	6219	6868	7834	8193	8807	10326	10822	11411
9	4727	5681	6647	6858	7050	7975	8804	14156	10936	11405
10	4816	5566	6428	6552	7541	8080	8422	9820	10936	12666

Table B.4: The time in milliseconds that a certain amount of VM-based desktop instances took to shut down.

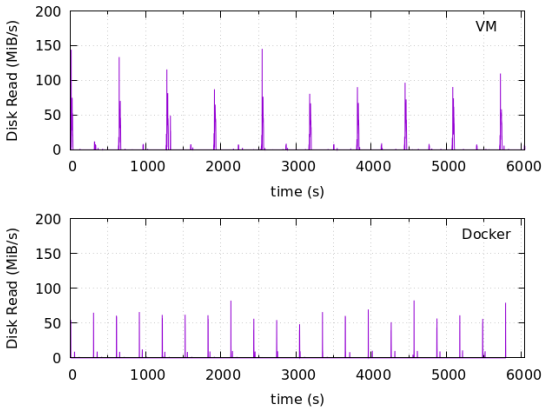
B.2 Disk Read/Write Rate



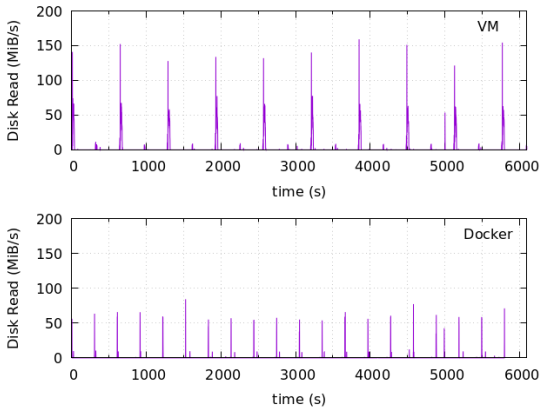
(a) Figure of one desktop instance



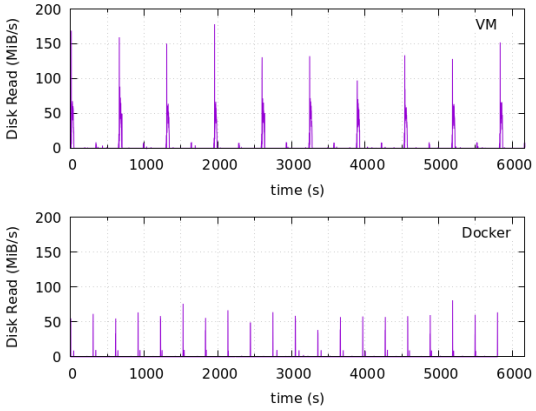
(b) Figure of two desktop instances



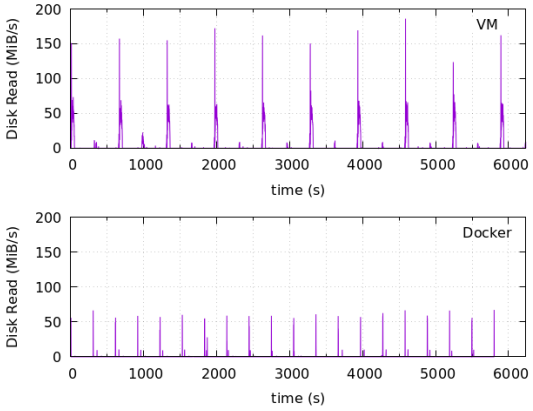
(c) Figure of three desktop instances



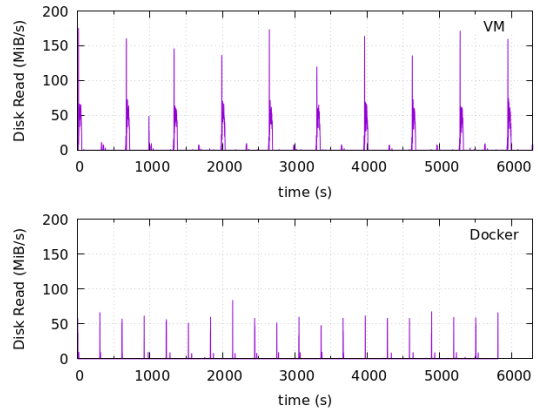
(d) Figure of four desktop instances



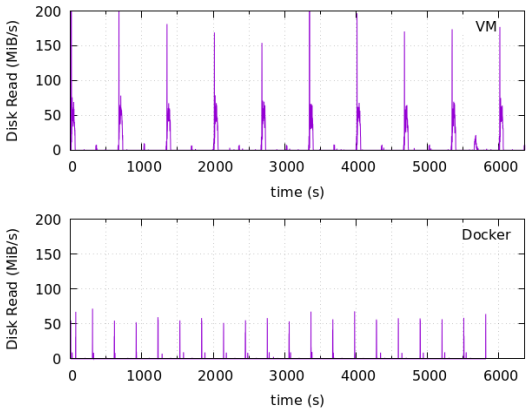
(e) Figure of five desktop instances



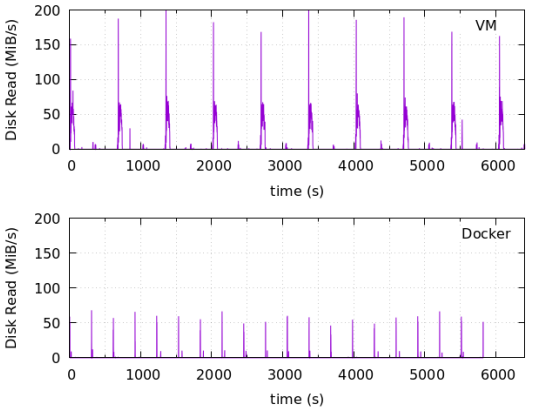
(f) Figure of six desktop instances



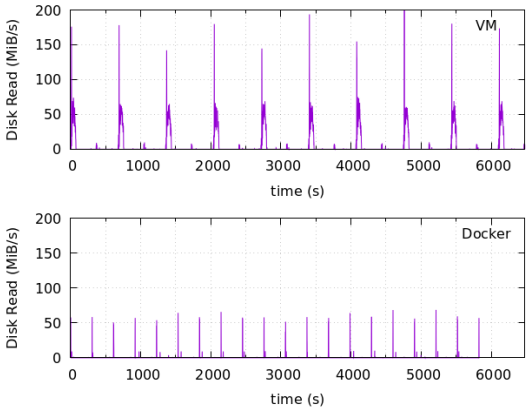
(g) Figure of seven desktop instances



(h) Figure of eight desktop instances

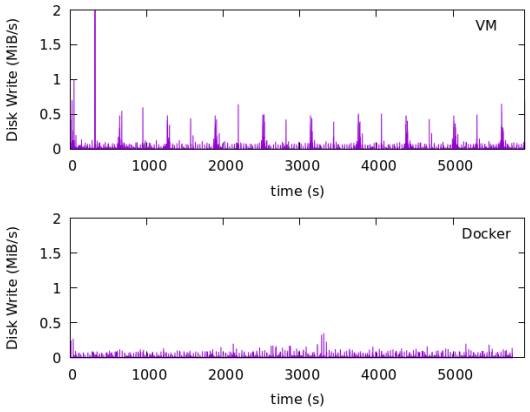


(i) Figure of nine desktop instances

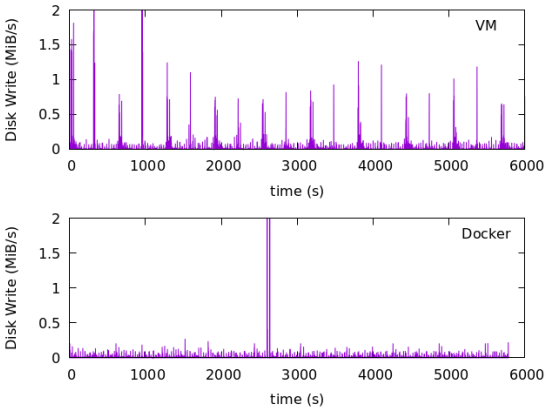


(j) Figure of ten desktop instances

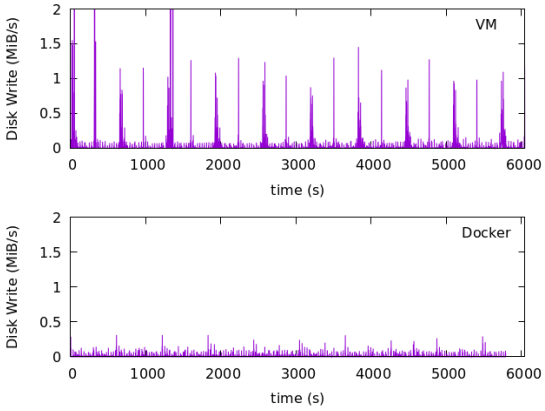
Figure B.1: Disk Read rate of ten consecutive samples in one plot



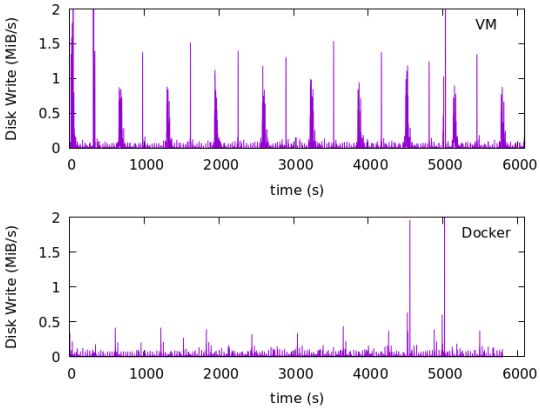
(a) Figure of one desktop instance



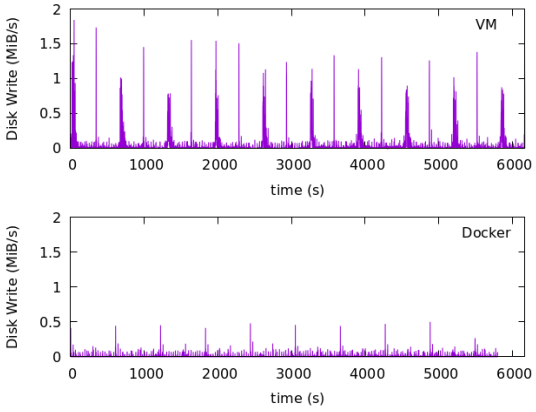
(b) Figure of two desktop instances



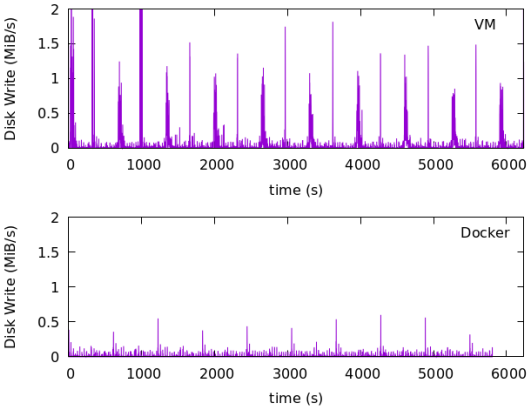
(c) Figure of three desktop instances



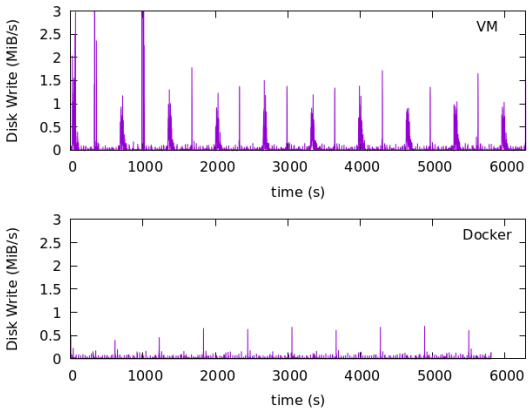
(d) Figure of four desktop instances



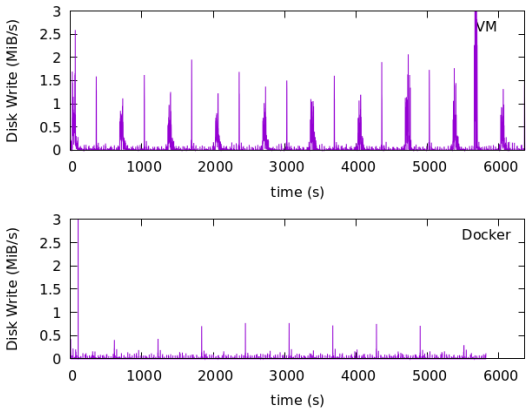
(e) Figure of five desktop instances



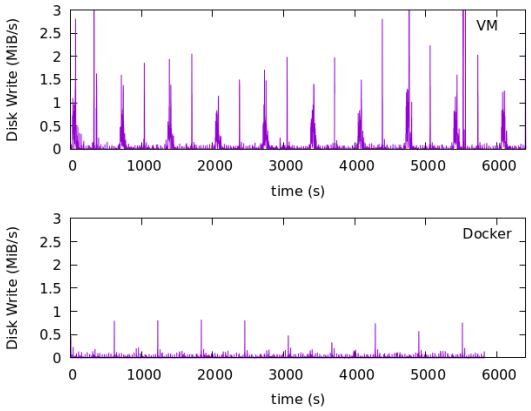
(f) Figure of six desktop instances



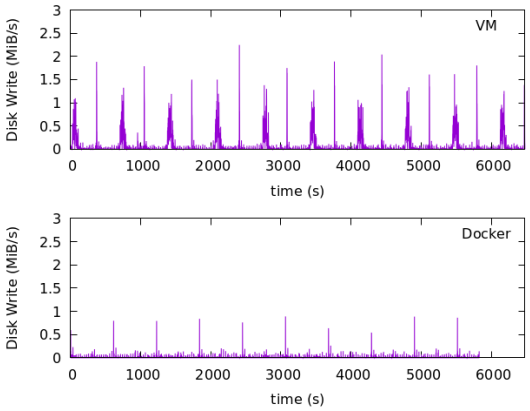
(g) Figure of seven desktop instances



(h) Figure of eight desktop instances



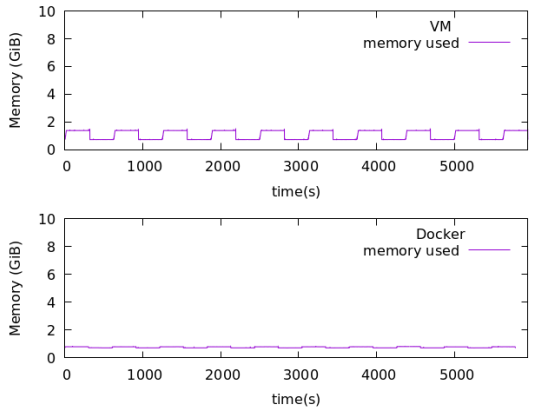
(i) Figure of nine desktop instances



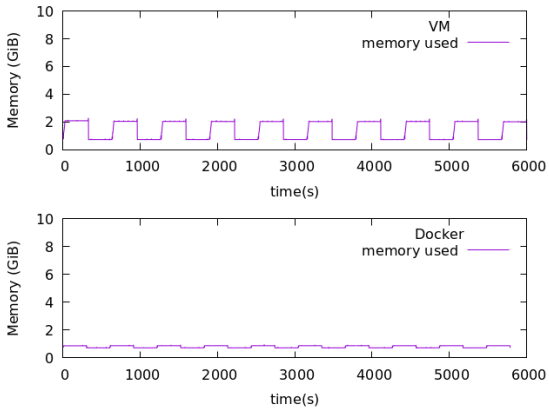
(j) Figure of ten desktop instances

Figure B.2: Disk write rate of ten consecutive samples in one plot

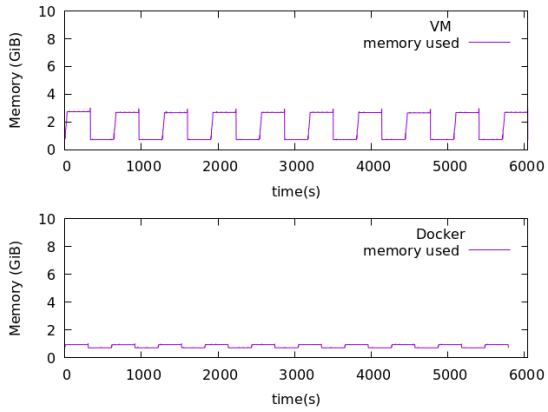
B.3 Memory Usage



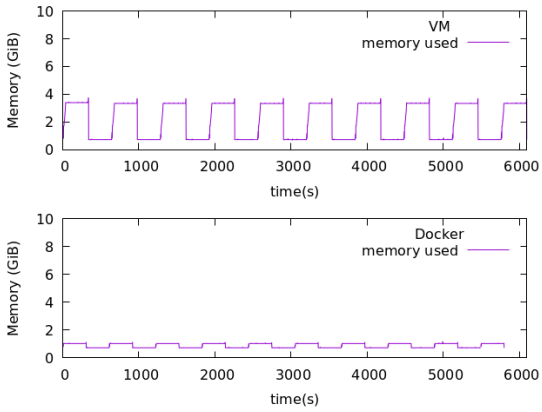
(a) Figure of one desktop instance



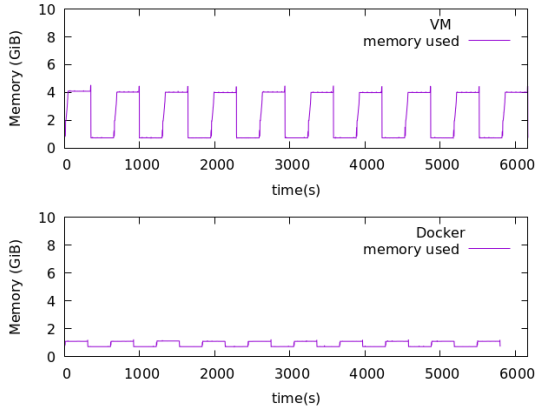
(b) Figure of two desktop instances



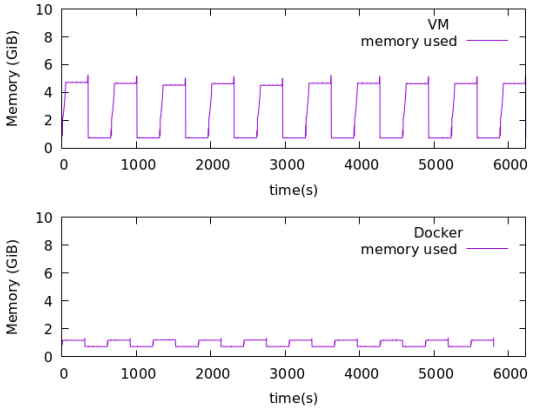
(c) Figure of three desktop instances



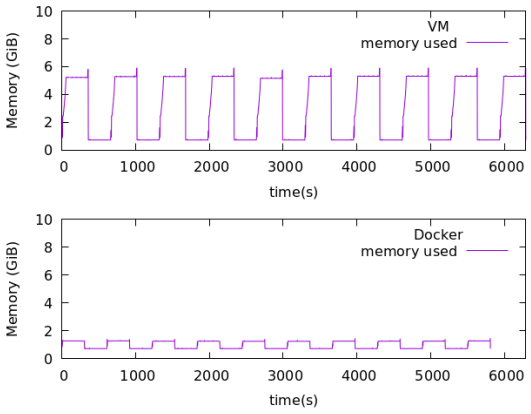
(d) Figure of four desktop instances



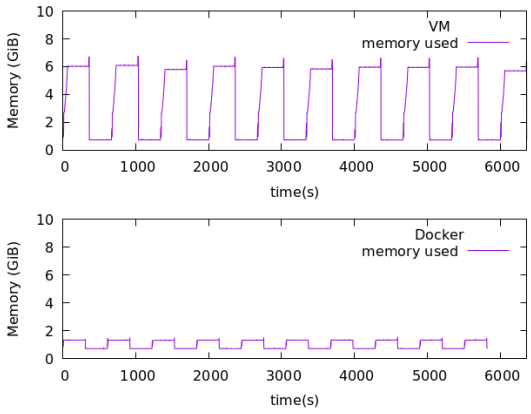
(e) Figure of five desktop instances



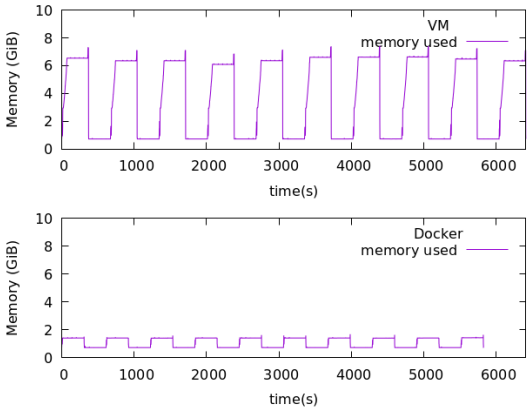
(f) Figure of six desktop instances



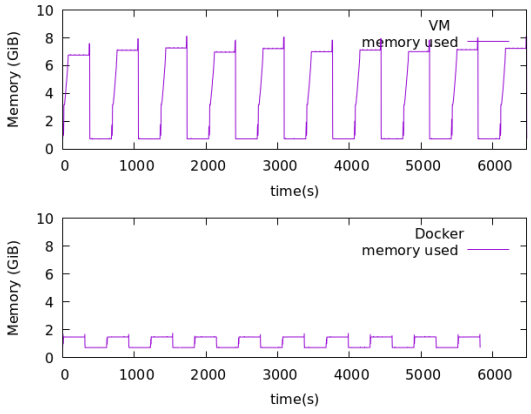
(g) Figure of seven desktop instances



(h) Figure of eight desktop instances



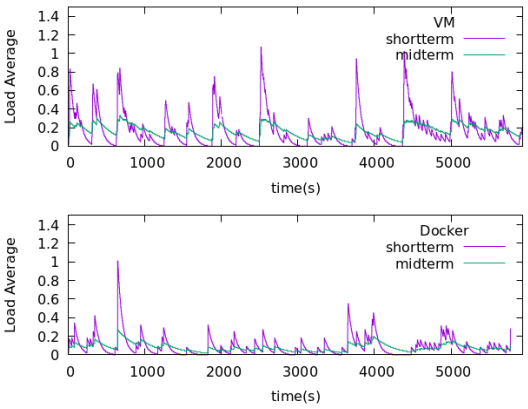
(i) Figure of nine desktop instances



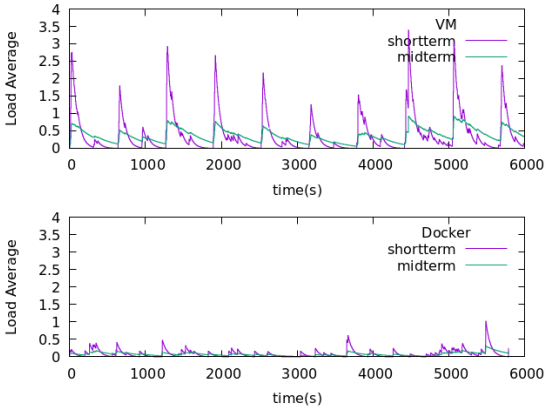
(j) Figure of ten desktop instances

Figure B.3: Memory of ten consecutive samples in one plot

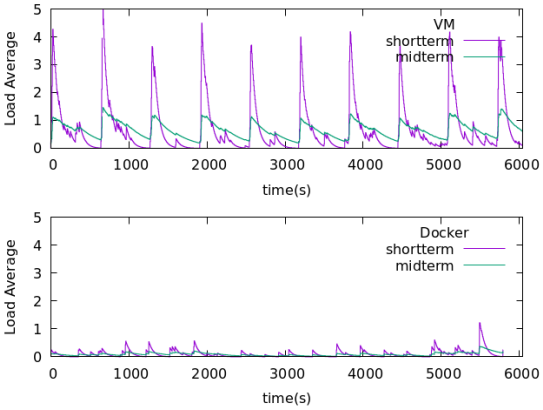
B.4 Load Average



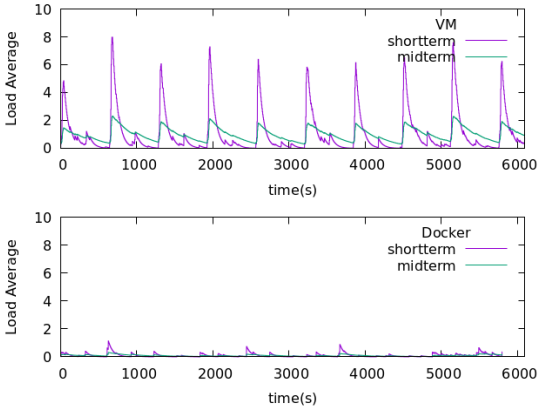
(a) Figure of one desktop instance



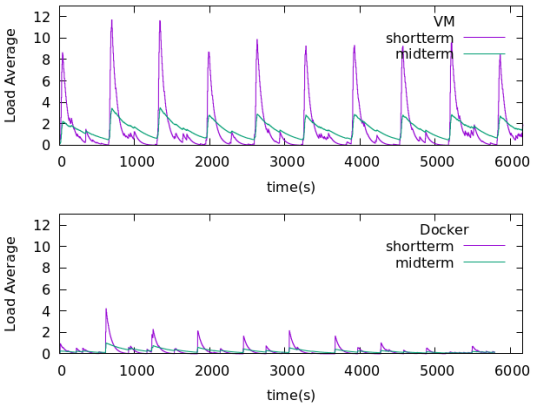
(b) Figure of two desktop instances



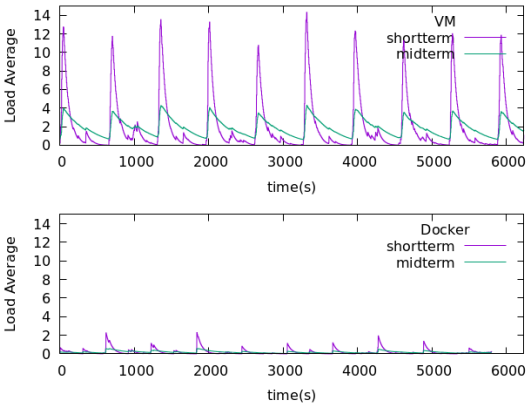
(c) Figure of three desktop instances



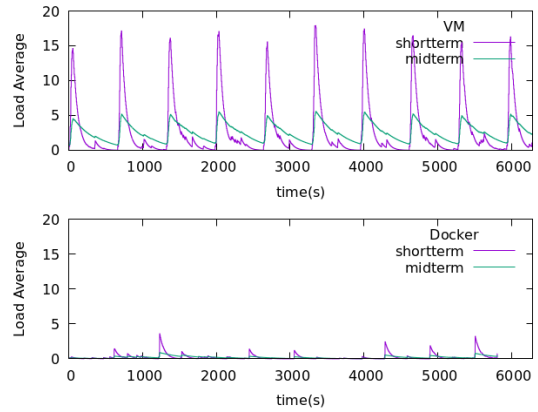
(d) Figure of four desktop instances



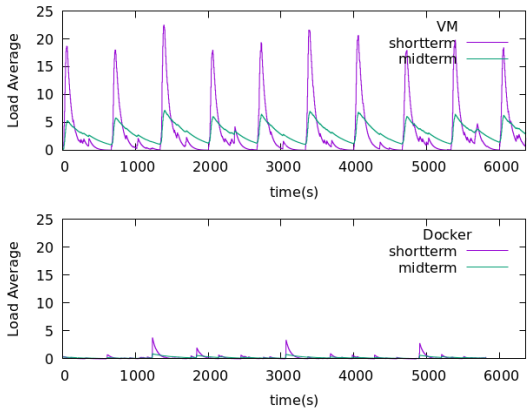
(e) Figure of five desktop instances



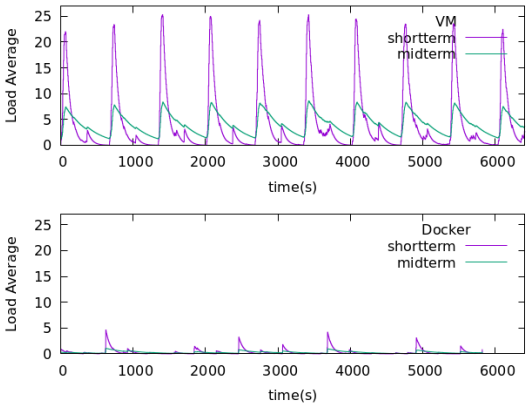
(f) Figure of six desktop instances



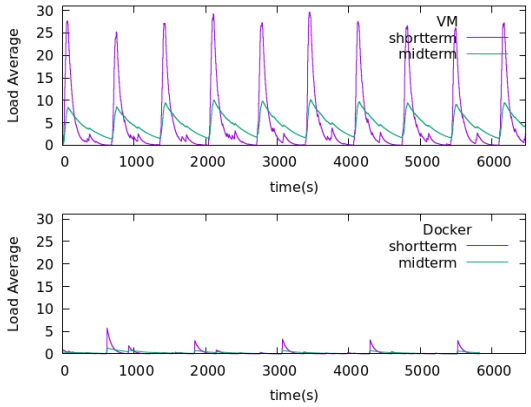
(g) Figure of seven desktop instances



(h) Figure of eight desktop instances



(i) Figure of nine desktop instances



(j) Figure of ten desktop instances

Figure B.4: Load average of ten consecutive samples in one plot

B.5 HA Failover Time

HA Failover Time (s)		
	Proxmox VE Cluster	Kubernetes
1	133	43
2	144	47
3	134	41
4	134	44
5	135	45
6	145	49
7	136	51
8	134	45
9	135	54
10	134	44

Table B.5: The time in seconds that the failover of a desktop instance takes in both prototypes.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA

EXAMENSARBETE An Analysis of Desktop-as-a-Service Implementations
STUDENT Hexu Huang
HANDLEDARE Flavius Gruian (LTH), Long Qian (Yark Network)
EXAMINATOR Jonas Skeppstedt (LTH)

Fast and Resource-Efficient Desktop From Anywhere

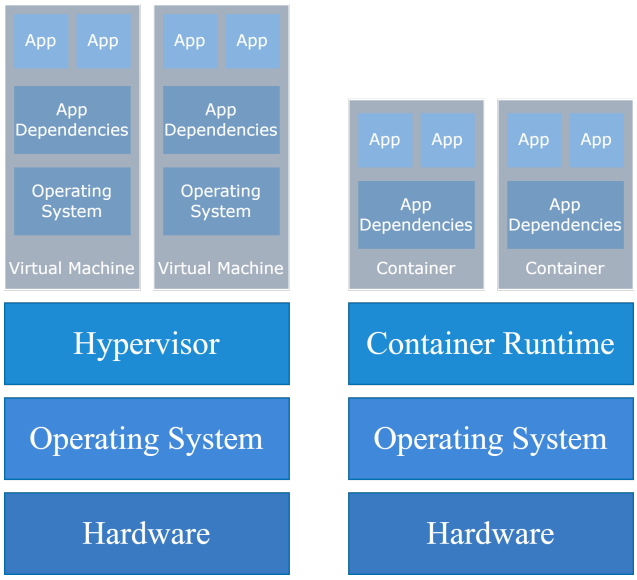
POPULÄRVETENSKAPLIG SAMMANFATTNING **Hexu Huang**

Virtualization has become increasingly popular these years, among which Virtual Desktop has been outstanding and widely accepted. This thesis explores the possibility of using Container for Desktop-as-a-Service (DaaS) by comparing the performance and high availability solutions between Container and Virtual Machine (VM) technologies.

Improved hardware performance and decreasing costs have led to the rapid spread of virtualization technologies, especially Virtual Desktops. DaaS provides a total solution for virtual desktops targeting enterprise and institution customers. Online virtual desktops can be useful when users need to work from home or onsite, but are not comfortable carrying or moving the high-performance workstations in the office. DaaS is also conducive to centralized configuration and management, saving maintenance costs on physical devices.

Most of the existing DaaS solutions on the market are based on VM technology. It has good versatility but oftentimes performance bottlenecks or are not easily customizable to quickly meet the growing demands of users. Those conventional VM solutions suffer from slow boot/shutdown and high resource usage because each virtual desktop instance must run in a virtual machine, and each VM essentially runs a full operating system (OS) including its own kernel.

DaaS powered by containers can overcome these problems. Container technology is not complete virtualization; each running container can share the kernel that the host is running and does not need to boot its own OS and kernel. Therefore it is excellent for both boot/shutdown time and the usage of resources like CPU, RAM, and disk.



For cluster high availability (HA) of DaaS, container technology can use Kubernetes, which has been popular and mature lately. It is free and open source, well documented, and has a good popular user base. Our experimental results show that a failover in the Container and Kubernetes implementation is also faster than the Proxmox VE Cluster based on the VM technology.