TÉCNICO LISBOA

Centroid

A Remote Desktop With Multi-OS Interoperability Support

André Pimenta Alves

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Nuno Miguel Carvalho dos Santos
Prof. Ricardo Jorge Fernandes Chaves

Examination Committee

Chairperson: Prof. Nuno Cavaco Gomes Horta
Supervisor: Prof. Nuno Miguel Carvalho dos Santos
Members of the Committee: Prof. João Nuno de Oliveira e Silva

November 2016

Acknowledgments

I want to thank my thesis supervisors for their enthusiasm and much needed guidance. My family for supporting and giving me all the conditions to succeed. My girlfriend for helping me grow as a person and inspiring me to do my best. And finally, my friends, especially my colleagues from my startup Peers, for helping me getting one step closer to my dreams.

To each and every one of you, thank you.

Abstract

With the advent of cloud computing, emerging services provide people access to their data from any device over the Internet. Following this trend, remote desktops provide virtualized operating systems, where people can run applications and files, that can be accessed anywhere and anytime over the Internet. Some of the existing remote desktop works provide a browser interface for conveniently accessing them. Others also provide access to multiple operating systems expanding the range of applications that users can run. However, remote desktop works still incur several problems. Their overhead produces long waiting times, they are not capable of efficiently streaming graphically demanding applications and, when applications run on different operating systems, they cannot communicate with each other. This reduces remote desktops usability making them a less effective alternative to using applications on the local device. Therefore, our goal is to analyse and develop a solution that overcomes these issues.

Considering this challenge, this thesis proposes a system capable of delivering remote desktops that support transparent communications between applications running on different operating systems, with a particular focus on scalability, remote display quality and costs. A functional prototype of the system named Centroid was produced consisting of a remote desktop service that delivers applications running on the cloud to a browser interface. Since they can run on different operating systems, specifically Windows and Linux, an interoperability component was also developed to enable communications between them. The experimental evaluation's results show that Centroid is scalable, application communication is efficient even between two different operating systems, the remote display performs twice as good needing only 2/3 the bitrate comparing to related works, and is able to reduce costs by over 40%.

Keywords

Remote Desktop, Cloud Computing, Multi-OS, Interoperability, Remote Display

Resumo

Com o advento da computação na nuvem, têm emergido serviços que fornecem às pessoas acesso aos seus dados a partir de qualquer dispositivo através da Internet. Seguindo esta tendência, serviços de ambiente de trabalho remoto fornecem sistemas operativos virtuais, onde se pode correr aplicações e ficheiros, que podem ser acedidos de qualquer lado e a qualquer momento através da Internet. Alguns trabalhos existentes nesta área, propõem sistemas que podem ser acedidos de uma forma conveniente a partir de um browser. Outros ainda oferecem acesso a múltiplos sistemas operativos expandido o espetro de aplicações que se pode usar. Contudo, estes trabalhos contêm vários problemas. Produzem tempos de espera elevados, não conseguem transmitir eficazmente aplicações que sejam graficamente intensivas e ainda, quando as aplicações correm em sistemas operativos diferentes, não é possível qualquer comunicação entre elas. Isto reduz a usabilidade destes serviços fazendo com que acabem por ser uma alternativa ineficaz para simplesmente usar aplicações no dispositivo local. Portanto, o nosso objetivo é analisar e desenvolver uma solução para estes problemas.

Tendo em conta este desafio, esta dissertação propõe um sistema capaz de fornecer ambientes de trabalho remoto com suporte a comunicação transparente entre aplicações que estejam a correr em sistemas operativos diferentes, com particular foco em escalabilidade, qualidade do display remoto e custos. Um protótipo funcional do sistema, chamado Centroid, foi produzido e consiste num serviço de amibiente de trabalho remoto que transmite aplicações, que correm na nuvem, para uma interface no browser. Dado que estas aplicações podem correr em sistemas operativos diferentes, mais especificamente Windows e Linux, um componente de interoperabilidade foi também desenvolvido para permitir comunicações entre estas. Os resultados da avaliação experimental mostram que o Centroid é escalável, a comunicação entre aplicações é eficiente mesmo entre dois sistemas operativos diferentes, o display remoto tem uma performance duas vezes melhor precisando só 2/3 da bitrate comparando com trabalhos relacionados, e ainda reduz os custos em mais de 40%.

Palavras Chave

Ambiente de Trabalho Remoto, Computação em Nuvem, Multiplos Sistemas Operativos, Interoperabilidade, Display Remoto

Contents

List of Figures

List of Tables

Acronyms

OS	Operating System
VM	Virtual Machine
VDI	Virtual Desktop Infrastructure
HQ	High-Quality
LQ	Low-Quality
FPS	Frames Per Second
IDE	Integrated Development Environment
AWS	Amazon Web Services
EC2	Elastic Compute Cloud
S3	Simple Storage Service

1

Introduction

In this work, our focus is to develop a system that provides a remote desktop service to users. In this context, a remote desktop consists of a virtualized operating system environment hosted in the cloud which can be accessed by users over the Internet through a browser interface. Our system allows for running unmodified applications built for conventional operating systems, such as Windows and Linux, on the cloud. The remote desktops operate by sending the screen images of the applications to the browser and receiving back the users' mouse and keyboard operations.

1.1 Motivation

Cloud computing has given people the ability to synchronize their computing environment over different devices. People can access their photos, videos and documents whenever they want over the Internet. Google Drive and Dropbox are examples of services that work on a range of operating systems that provide this type of service. Following this trend, there are remote desktops. Remote desktops allow people not only to access their files, but also their applications any time and anywhere over the Internet. Furthermore, they allow for businesses to provide, in a more efficient way, a ready-to-use computing environment to workers. And workers can use them from any location over the Internet. Essentially, they provide operating systems, running on virtual machines, hosted on a remote server that users can access from any device.

Existing works, such as W. Wei et al. [1], developed a system that provides remote desktops that are accessed through a browser interface. Given that browsers exist on a range of operating system by default, users can start using them in a more convenient way not having to install any additional software. However, this work provides remote desktops with only Windows operating system. Other works, such as G. Lai et al. [2] and M. Fuzi et al. [3], support remote desktops with multiple operating

systems. This extends the range of applications that users can run. However, running applications on different operating systems means they run on different virtual machines, thus separated from each other. A negative aspect of this situation is that these application no longer have access to the same files. This may be not convenient if users want to work, for example, on the same set of files with any application regardless of their operating system. To overcome this, Redar [4] implements a distributed file system that is mounted on every execution environment so that applications, even if separated from each other, can access a global user file system.

In spite of these features, existing works still incur several problems. Firstly, their overhead comparing to running applications locally is very noticeable, for example, users have to wait for a full operating system and virtual machine to load before they can start using an application. This can take up to 4 minutes. Furthermore, the remote displays, which are responsible for streaming applications' screen images to users, do not perform well for graphically demanding applications. This limits the types of applications that users can use, reducing remote desktops usability. Finally, some remote desktop services already provide access to different operating systems by making applications run on different virtual machines. However, the consequence of this is that these applications are no longer able to communicate with each other like they would if they were running on the same operating system. For example, if a user's favourite IDE application runs on Windows but the compiler he needs runs on Linux. The IDE is not able to call the compiler. Being able to run applications from different operating systems expands the range of applications, but in the end, users may have to choose only one operating system if they need applications to communicate.

1.2 Goals and Requirements

The objective of this work is to develop a system capable of delivering remote desktops that support transparent communications between applications running on different operating systems. We called this ability, interoperability. More specifically, the requirements for this work are:

R1: *Interaction between applications from different operating systems (interoperability):* Applications running on the same or separate virtual machines and on the same or different operating systems must be able to interact with each other transparently. This interaction happens when one application explicitly executes another one. The input and output of each application is then redirected between them, thus enabling communication.

R2: *Scalable:* Being a remote desktop that is going to be used by people to run their applications requires inherently a good user experience. For example, users should not wait a long time for an application to start comparing to what they would on a local computer. And, this performance of the system should not decrease when the number of users increases.

R3: *Remote display performance suitable for any type of application:* The remote display is how users access applications and use them. It should not discriminate applications even if they are very graphically dynamic. For this to happen, the performance of the remote display must be able to handle even the most demanding graphical applications.

R4: *Lower cost when compared to other related works' approaches:* Centroid must be designed and implemented taking into account costs. It should cost less than other related works solutions and also be competitive to buying a high-end computer.

1.3 Contributions

This thesis analyses, develops and evaluates a system capable of delivering remote desktops with a focus on scalability, costs, remote display quality and communications between applications from different operating systems. As such, this work makes the following contributions:

1. Proposes a system called Centroid capable of efficiently delivering and managing remote desktops, a new remote display capable of streaming the most demanding graphical applications and a novel way for applications to interact with each other even if running on separate machines or different operating systems.

2. Produces a functional prototype of Centroid consisting of a system that delivers applications via remote display to a browser interface. These applications run on containers distributed among available virtual machines and have access to the same user file system. Since they can run on different operating systems, specifically Windows and Linux, an interoperability component was developed to enable communications between them. We deployed our system on the Amazon Web Services (AWS) cloud.

3. Presents a comprehensive analysis of several related works in three different areas: remote desktop services and their infrastructures; remote displays; and communication between applications running separated from each other.

4. Provides an experimental evaluation of Centroid in regard to its scalability, remote display quality, application communication performance and costs. Results show that Centroid is scalable, application communication is efficient even between two different operating systems, the remote display performs twice as good needing only 2/3 the bitrate comparing to related works, and is able to reduce costs by over 40%.

5. Allows for an integration of a more complex distributed file system named Storekeeper that produced a paper, made by the authors of this thesis, that was published and accepted on SRDS 2016, the 2016 IEEE 35th Symposium on Reliable Distributed Systems.

1.4 Thesis Outline

The rest of this document is divided as follows. Section 2 describes concepts that require previous knowledge in order to implement and design the proposed system. Section 3 introduces the related work, covering existing solutions that are related to our work. Section 4 presents the full design of Centroid. Section 5 presents its implementation. Section 6 introduces Centroid's evaluation results. Section 7 presents the conclusions and future work.

2

Background

Designing and building the proposed system requires the knowledge of concepts ranging from virtualization (Section 2.1) to cloud computing (Section 2.2) to remote desktops (Section 2.3). With virtualization it is possible to abstract hardware resources to create virtual computing environments such as virtual machines and containers which are both used on Centroid. Furthermore, cloud computing makes use of virtualization to provide a scalable, highly available platform for developers to create and deploy complex systems. Cloud computing hosts all the Centroid components such as the virtual machines running operating systems, the distributed file system and the administrator platform. Moreover, the applications running on these operating systems are then delivered to end-users via remote display. The rest of this chapter describes these concepts.

2.1 Virtualization

Starting from a low level concept, virtualization allows running operating systems or applications on the same pool of resources isolated from each other. There are several types of virtualization, such as application virtualization and operating system virtualization. Learning about operating system virtualization is crucial to understand the mechanisms of cloud computing where the remote desktop will run. Operating system virtualization is a technology that introduces a software abstraction layer between the computer resources and the operating system and applications running on top of it [5]. This abstraction layer controls directly and partitions these hardware resources, creating virtual environments called virtual machines. The operating systems then run on these virtual machines and behave as they normally would on a real environment. As a result, it is possible to run multiple operating systems in parallel on the same hardware. Virtualization brings many benefits such as improved security, flexibility, scalability, adaptability and reduced costs as described in [5]. The biggest disadvantage of virtualization

is its overhead [5] because it generally requires the translation of instructions from virtual environment to the physical hardware. This section introduces two virtualization approaches: the hypervisor-based virtualization and the container-based virtualization.

2.1.1 Hypervisors

As mentioned above, virtualization introduces an abstraction of the computer resources. The layer between the operating systems and this hardware resources can be provided by adding a hypervisor. A hypervisor, also called virtual machine monitor (VMM), is a software layer that provides the abstraction of the hardware to the operating system by allowing multiple virtual machines, termed as guests, to run on a host machine [6]. There are two main types of hypervisors: type 1 hypervisors, for example Vmware ESX [7] or Xen [8], and type 2 hypervisors, for example VirtualBox [9] or Parallels [10]. The major difference between the two hypervisors is that the first runs directly on top of the underlying hardware, and the second runs as a normal application in an operating system, as shown in Figure 2.1.

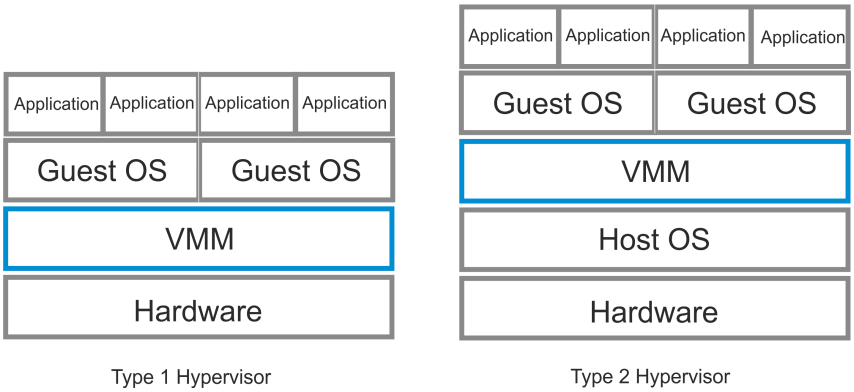

Figure 2.1: Architecture of type 1 and type 2 hypervisors.

Type 1 hypervisors, also known as ”bare metal”, provide the guest virtual machines with direct access to the hardware. The responsibility of the hypervisor is to schedule and allocate resources to the virtual machines because there is no operating system running below it [11].

Type 2 hypervisors run as an application in an operating system. This operating system is known as the ”Host OS” and the operating systems running on top of the hypervisor are called ”Guest OS” [11]. The Guest OS has access to the hardware through the Host OS and all calls to the hardware are intercepted by the Host OS which then performs them in behalf of the Guest OS, sending back the result [6]. Compared to a type 1 hypervisor, the type 2 hypervisor can be slower and more prone to security vulnerabilities [11]. Type 2 hypervisors run in a normal operating system which is a security hazard because if the host OS is compromised, allowing it to have full control of the guest operating systems. Even the fastest virtual machines with hypervisors type 2, suffer from a small overhead as

the machine instructions have to be transmitted from Guest OS to Host OS to hardware and the virtual machine image is not easy to migrate as it takes a large storage space and takes minutes to boot. A more lightweight approach that mitigates this issues are containers. Although they also virtualize an operating system, they use resources directly from the host, making them much smaller, faster and easier to migrate.

2.1.2 Containers

Container-based virtualization can be seen as a lightweight alternative to virtualization through hypervisors, especially for type 2 hypervisors. This is achieved by delivering a different kind of abstraction in both virtualization and isolation [12]. Container-based virtualization implements isolation of processes at the operating system level and each container runs on top of the same host operating system kernel, in essence, it does not need to create a completely new operating system for each container. In comparison, a type 2 hypervisor provides abstraction for full guest operating systems, one for each virtual machine [13]. This comparison is illustrated in figure 2.2.

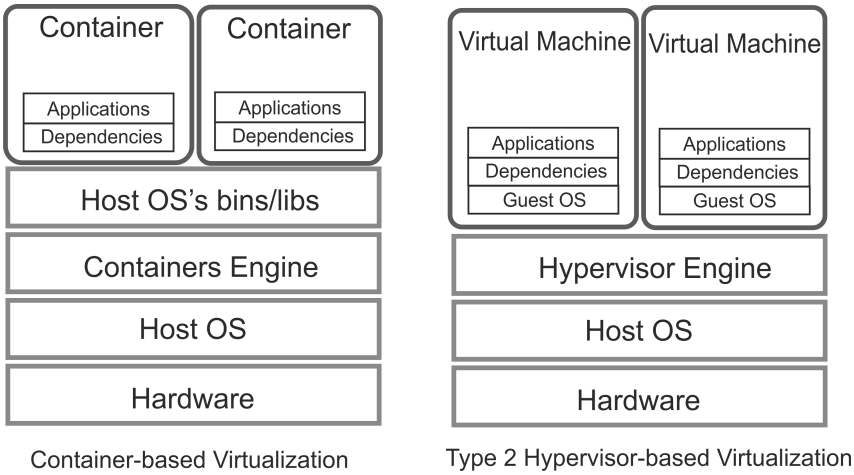

Figure 2.2: Container architecture compared to an hypervisor architecture.

Isolation is achieved by kernel namespaces. It allows processes to have different and private views of the operating system. Resources like filesystem, process IDs, inter-process communication (IPC) and network can all be isolated through namespaces. This approach makes each container look and execute as if a stand-alone operating system.

An example of a container technology provider is Docker [14] which is one of the most used by developers. Docker helps to build applications inside containers and share them among team members bringing down development time [15]. It was initially based on LXC (Linux Containers) and uses resource isolation features of the Linux kernel. However, this means that Docker containers only run on Linux distributions for now. A Docker for Windows is being developed and will be released with Windows

Server 2016.

The concept of a shared kernel makes the disk images of containers much smaller than virtual machines. Also, the fact that none of the instructions have to be translated from Guest OS to Host OS, makes the containers experience near that of native performance and faster start-time [16]. The disadvantages of sharing a single operating system kernel is weaker isolation [13] and the fact that containers have to be ran on top of the same operating system as their host, this means that Linux containers cannot run on Windows and vice-versa [12].

2.2 Cloud Computing

On a higher level, cloud computing is a platform built using virtualization technology. Cloud computing can, for example, be used to host the remote desktops where users can connect and use from anywhere and anytime taking advantage of the virtually infinite resources that the cloud provides.

Cloud computing is the delivery of services over the internet [17]. These services can be computing resources like CPU, application software and storage. The goal of this computing model is to make more efficient the use of distributed resources putting them together to achieve higher throughput and tackle large scale computation problems [18]. Some examples of cloud computing providers are Amazon Web Services (AWS) [19], Google Cloud Platform [20] and Microsoft Azure [21]. They leveraged their existing infrastructure, initially meant to be used for internal operations, by providing public access to their data centres adding new revenue stream [22].

Customers can buy the computing resources provided instead of having to set the infrastructure or data centre themselves, getting billed based on the resources consumed. This method of payment is called pay-as-you-go and gives costumers the opportunity to buy computing resources as if they were a utility [22]. The physical location and configuration of the underneath system are unknown to the customer, they only deploy their own software, which can be dynamically scaled to fulfil the customer needs. It allows for the most cost-effective development of scalable applications or systems on highly-available environments [23].

In addition to a reduction of costs, particularly relevant to small and medium companies, cloud computing offers other advantages [24] as detailed in Table 2.1:

The biggest concerns about cloud computing technology are security and privacy issues. For example, when an organization stores sensitive data that are not encrypted on the cloud there is always an inherent level of risk, both coming from the cloud provider that has now access to that data or from attacks to the cloud, which can lead to unlawful access to that data [23].

Based on the level of abstraction of the underlying infrastructure provided, cloud computing can be differentiated by mainly three service models. Going from the highest abstraction to the lowest

Uninterrupted Services:	Usually more dependable compared to the infrastructure installed on the organization because the staff and the architecture are specialized and prepared to handle system failures. Amazon EC2 Service Level Agreement specifies a service commitment stating a Monthly Uptime Percentage of at least 99.95% [25].
Easy Management:	The cloud provider is responsible for managing the infrastructure so IT teams have their job made easier. Also, from the perspective of a user, any application running on the cloud can be accessible from a web browser or device with internet connectivity.
Disaster Management	Cloud storage services allows organizations to back up crucial data. Additionally, the cloud provider itself usually have disaster recovery systems in place.
Green Computing	The efficiency of the cloud computing allows for lesser electronic waste and energy consumption, leading to environment preserving.

Table 2.1: Cloud Computing Advantages

abstraction respectively, Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). Online applications like Google Docs or web-based e-mail are included in the Software as a Service model. Developing and deploying user applications is the basis for Platform as a Service model. And processing power or storage are provided by an Infrastructure as a Service model. These models are illustrated on figure 2.3.

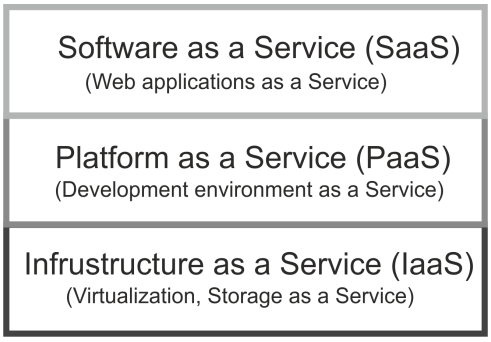

Figure 2.3: Cloud Xaas models.

2.2.1 Infrastructure as a Service

This type of service provides dedicated virtual machines with full control over the instance machine [17]. It is provided CPU, storage capacity, operating system, network and the user has total configuration power over the virtual machine. The exact way how the virtual machine accesses the computing resources is hidden from the user, only being the responsibility of the cloud provider. Developers can use these virtual machines to deploy their software, applications, servers or databases.

Amazon Elastic Compute Cloud (EC2) [26] is one of the most popular Infrastructure as a Service

providers, it allows the acquisition of virtual machines with a lot of configurable resources and several operating system.

2.2.2 Platform as a Service

This type of service provides an environment ready to develop and deploy applications [27]. This high-level software infrastructure comes with tools and programming languages that the developer can use to build a web application. Developers have no control of the underlying infrastructure or operating system, they only depend on the available programming languages and tools presented by the cloud provider. This strategy allows for a reduction in the development time offering readily available tools, services and an easy way to scale.

An example of a platform as a service is Google App Engine that offers a Python or a Java environment comprising the end-to-end life cycle of developing, testing, deploying and hosting of sophisticated web applications [18].

2.2.3 Software as a Service

This type of service delivers applications running on a cloud infrastructure to the user through the internet from a thin client interface such as a web browser. The application is hosted off site and the user does not need to worry about installation, updates or the underlying infrastructure as it is totally abstracted from the user [17]. Typically, this kind of software is called a web application.

Popular software services are the Google Apps: the web based email, calendar, Google Docs which allows access and sharing of documents, spreadsheets and presentations and also Google Drive that provides storage to the user allowing it to be accessed from anywhere and share data with other people over the internet.

2.3 Remote Desktops

With the development of cloud computing, attention is turning towards the emerging cloud services, which use remote centralized resources. Users are now able to access their personal computing environment over the internet with any device. One of the approaches to take advantage of these cloud services is the possibility of providing remote desktops [28].

A remote desktop environment is the practice of hosting a desktop operating system installed on a virtual machine that is located on the cloud or on a remote server [29]. It operates by receiving the mouse and keyboard information from the client and sending back the desktop screen image, this architecture is illustrated on figure 2.4. All the applications and operating system run on the cloud and do not use any

of the client's computer resources. On the client side, there is only a small piece of software installed that is just capable of connecting to the remote desktop and receiving its graphical data. This software is called a thin-client.

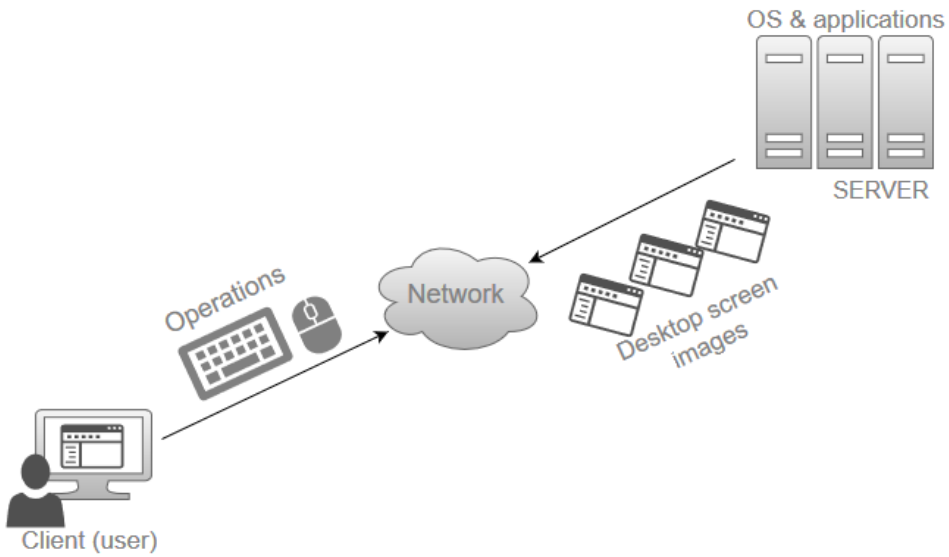

Figure 2.4: Remote desktop architecture.

A remote desktop service should provide remote operating systems that feel exactly like they are running on the local machine, meaning that the latency should not be noticeable to the user. A number of variables influences this latency like internet connection speed, which nowadays is a lesser problem [30], remote and local machine encoding/decoding capabilities, but the most prominent factor is the implementation of the remote display protocol.

The remote display protocol is the main differentiating factor between remote desktop providers. The remote display is responsible for the transmission of the graphical data [30]. Each protocol uses its own way to transmit the screen from the remote site to the client local device, but the two basic operations that are common to all of these protocols are:

1) Detection of the display updates by comparing the current frame buffer with the previous one at a fixed time interval.

2) Compression of the image of the updated area to be transferred to the client.

There are a number of remote display protocols such as PC over IP (pcoip) [31], Remote Graphics Software (RGS) [32], and the very popular VNC [33] and Remote Desktop Protocol (RDP) [34]. The last two protocols are the most popular are explained below:

VNC is a cross-platform application that can transfer the desktop of a server computer to a client. The network protocol used by VNC is called Remote Frame Buffer protocol which supports different kinds of image compression encoding such as hextile, zlib and tight [35]. Hextile algorithm divides the

screen into 16x16 pixels tiles and then compares them with the previous frame. If there are any changed pixels, the compression process iterates the tile to find an unicolored rectangle and then encodes the color, position and size of the rectangle. Hextile is known for its high speed, but does not perform so well with complex images. Zlib is a software implementation of the deflate algorithm. Deflate is a lossless compression algorithm that tries to find the matched data from a storage of previous information. This algorithm has a compression level parameter which controls the ratio between speed and compression level that can be set from 1 to 9 where 1 is the fastest speed but lowest compression level and 9 is the slowest speed but highest compression level. This compression level is usually set to a low number because a long compression time may result on undesirable latency. Tight encoding, introduced by Kaplinsky, incorporates JPEG which is a lossless compression algorithm that performs very well, in terms of compression level, when applied to natural images such as landscape photos but can have a negative effect on the system response because of its slow speed.

RDP (Remote Desktop Protocol) was created by Microsoft and is only available on Microsoft Windows. RDP captures rendering commands of Windows on the server and sends them over TCP to the client which works well for office applications or any application with fewer graphical primitives [35]. When applied to graphical demanding applications it performs poorly due to a large amount of rendering commands generated by the graphical application.

Current remote displays can stream an office-like application with acceptable quality, however, high-end applications, especially graphical demanding applications still present a major obstacle [35].

2.4 Summary

This chapter presents several concepts that serve as a knowledge support to build and design of the proposed system. These concepts include virtualization, cloud computing and remote desktops. Virtualization is a technology that introduces a virtual layer between hardware and software. The most common virtualization methods are deployed through hypervisors and containers. In regard to remote computation, cloud computing is an emerging technology that facilitates the deployment and management of software, web servers and web applications. A way to take advantage of cloud computing is by providing a computing environment where users can run and install applications. Basically, it functions as a local computer or desktop, but because it runs entirely on the cloud, it is called a remote desktop. These remote desktops work by sending desktop screen images to the user and receiving mouse and keyboard operations. The streaming of these desktop screen images is called remote display.

3

Related Work

Providing users a remote desktop service where they can run applications imposes several challenges, namely: management of resources such as virtual machines, applications, files; quality of the remote display responsible for streaming the applications graphical data to the client; communication between applications from different operating systems. This challenges can be divided into many sub-challenges. Current works already address some of them as discussed in this chapter. Given that, each of these related works focus on particular aspects of these challenges, the following separates these three challenges into different sections. As such, Section 3.1 describes Virtual Desktop Infrastructures (VDI) which works propose efficient ways to manage and provide remote desktops to end-users. Section 3.2 is focused on the remote display and presents works that introduce effective ways to stream applications (screens) to end-users. Section 3.3 introduces interoperability and current works that focus on enabling communications between separated applications.

3.1 Virtual Desktop Infrastructure (VDI)

Virtualization technology (described in section 2.1) has made an enormous impact on building flexible and scalable systems which enabled the existence and success of cloud computing. It improves the efficiency of computing resources utilization, and offers users a personalized computing environment. Desktop virtualization has emerged as a way to execute operating systems on the cloud with the objective of providing remote desktops to end-users. Remote desktops were described in Section 2.3 and this section analysis how current works provide and manage these remote desktops. The practice of hosting, maintaining and managing desktop operating systems running within virtual machines on the cloud is called a Virtual Desktop Infrastructure (VDI).

Comparing to a local desktop, using a remote desktop provided by a VDI can bring benefits such

as computing resource flexibility, centralized maintenance, ubiquity, availability, ability to use multiple operating systems at the same time without decreased performance and energy efficiency. Although it may bring many benefits, building a high performance, scalable and cost-effective VDI is not a trivial task. It brings many challenges such as:

1. Management of virtual machines and execution environments,
2. Management and efficient deployment of users computing environment and files,
3. Optimization of resource utilization,
4. Quality of the remote display and application usage

The rest of this section outlines a number of works that have been made which tackle the challenges presented in items 1) 2) and 3) from the list above. Item 4) is described on section 3.2.

D. Wang et al. [36] propose a VDI capable of scaling to multiple users with a precise control of the used resources. Users access virtual machines running Windows where they can install and run applications. These virtual machines only use computing resources as they are needed achieving greater flexibility which leads to a more efficient consumption and decreased costs.

W. Wei et al. [1] propose a VDI with a web-based user interface to be used by both end-users and system administrators. End-users can use this interface to access and manage their virtual machines and also to manage their files. This file system is mounted in the form of network drives to the user's virtual machines, which provides a global file system. The administrator uses the interface for managing all virtual machines, users and storage space of the system.

M. Fuzi et al. [3] introduce a virtual desktop environment for laboratories at schools and universities. The objective is to give students an ubiquitous platform capable of running the most common school applications belonging to multiple operating systems. A VNC software was installed on every operating system so that students can access the virtual machines.

S. Kibe et al. [37] propose a system capable of delivering remote desktops featuring multiple operating systems namely CentOS, Ubuntu and Windows. Their goal is to enable students to access an educational cloud, where they can use multiple cloud desktops to do experiments and other tasks. They use VNC and RDP to access these remote operating systems from any client OS. They conclude that scaling the infrastructure to more users can reduce costs, but note that performance degrades.

Redar [4] develops a VDI where applications run in individual nodes, this way instead of accessing a full operating system, users only see individual applications. The way they accomplish this is by running and then joining applications screens into one frame and sending that frame to the client. This work also features a distributed file system that is mounted and accessed by every application. Currently, they feature only Linux and is aimed at office-like applications.

G. Lai et al. [2] propose a virtual desktop infrastructure with the objective of providing efficient services to users. Their contributions include: a remote display focused on applications instead of the entire desktop environment; a new protocol for controlling interaction flows between clients; and finally, a high performance allocation of resources. In this system, applications run on a Windows or Linux virtual machine. Each user accesses a set of pre-installed applications. To manage the virtual machines, they present a novel scheduling algorithm to monitor operation loads. The user interface is a software that the user installs and presents a list of his applications.

To do a comparison of the presented works, table 3.1 illustrates an overview of the different features provided by each one.

	Execution environment of user applications	Specialized resource management component	Distributed file system	Browser client	Multiple OSes	Interoperability between different OSes
D. Wang et al. [36]	Run inside one VM	Yes	No	No	No	No
W. Wei et al. [1]	Run inside one VM	Yes	Yes	Yes	No	No
M. Fuzi et al. [3]	Run inside one VM	No	No	No	Yes	No
S. Kibe et al. [37]	Run inside one VM	No	No	No	Yes	No
Redar [4]	Run inside single nodes	No	Yes	No	No	No
G. Lai et al. [2]	Run inside single nodes	Yes	No	No	Yes	No

Table 3.1: Comparison of VDI related works

The first entry in this table is the application execution environment, this is a major factor that differentiates VDIs. Two approaches are observed, namely:

1. Full Virtual Machines: The most general approach is to provide each user access to a virtual machine running an operating system. Users then, can install and run applications in these operating systems. When a user exits the session, the VDI has to turn off the virtual machine and save its image. When a user wants to access it again the VDI has to retrieve the virtual machine image and start it.

2. Individual nodes: Another approach is to run applications inside individual nodes distributed between available virtual machines. Instead of accessing full operating systems, users only see the applications. When a user exits an application, the VDI only has to store the state of the application and retrieve it when the user accesses it again.

Comparing the two approaches, virtual machines take much longer to start because they have to load a full operating system but they are more secure, since applications run on an isolated environment. Also, having to store a full virtual machine image leads to increased costs. The second approach leads to a faster start time, since it does not need to start an entire operating system, and does not have to store the entire image, only the application state. However, this approach leads to decreased security, since applications share the same environment. Redar [4] and G. Lai et al. [2] are examples where, although they provide applications running on individual nodes, the applications share the same OS environment without any constrains leading to security hazards. Furthermore, this applications come pre-installed and are defined by them, they do not provide a method for users to install new applications. For these two reasons, Redar [4] and G. Lai et al. [2] do not fully provide an acceptable alternative to running applications directly on virtual machines.

The second entry in the table, specialized resource management component, refers to how VDIs resources are managed, if they have an efficient way of allocating and managing virtual machines to users. The distributed file system entry in the table indicates whether VDIs have a global file system where every user application has access to the same files, even if running on separated machines. Browser client indicates if the applications can be accessed through a browser, which means that users do not have to install any software on the local device. The multiple OSes entry indicates if the VDI provides more than one operating system. Interoperability between different OSes indicates if an application from one OS can communicate or interact with another application from a different OS. Note that none of the existing works provide this capability.

3.2 Remote Display

The state of the art reported on the last section referring to VDI, fail to address one major problem which is the quality of the remote display in particular how efficiently are the applications screens streamed to the user. Their objective is to stream office-like applications with static graphics but are unsuccessful when it comes to streaming high-end graphical applications like video or games. The quality of the remote display can be related to how many frames are displayed to the user in a certain amount of time. This is called frame rate, usually measured in frames per second (FPS). More frames per second means higher quality. The professional frame rates for movies is 24 FPS and for television is 30 FPS (in the U.S). On the gaming world, it is either 30 FPS or 60 FPS. Games that run at 60 FPS are more fluid, the movement is more realistic and the experience is overall more enjoyable. The reason behind this is that with 30 FPS, it takes 33.3 ms to display the next frame while with 60 FPS it only takes 16.6 ms, resulting in a much more accurate and fluid motion. Given this, 60 FPS is the value to aim when considering the quality of the streaming in games. The approach taken by many VDI services, including

the ones described in Section 3.1, is to use a remote display protocol such as RDP or TeamViewer or Vnc but this type of remote display protocols are only capable of delivering 13.5 FPS [38] which is reasonable for office-like applications but very bad when it comes to graphical applications such as games or video. One of the requirements of this thesis is to provide users with a suitable remote display for any type of application whether it has low graphics needs or it is very graphically demanding. To analyse existent works, we decided to focus on cloud gaming services, since their job is to deliver high-end graphical applications (games) over the internet to users. The service is very similar to VDIs but they only focus on games and users cannot install any new application, they can only choose to run a game from a catalogue. Their objective is to maximize the quality of the game streaming. This is important because for most games, a frame rate of 7 FPS is considered unplayable while a frame rate of 60 FPS can actually increase users' performance [39]. So, it is on their interest to build a system that maximizes streaming quality. The rest of this section describes some of these works.

GamingAnywhere [40] is an open-source gaming system based on the cloud. It adopts the H.264 standard to encode applications frames via CPU. The frames are sent to the client using RTSP/RTP and are played on a multimedia player software. It currently features Windows and Linux on their server and Windows, Linux and OS X on the client-side. Tests [41] show that GamingAnywhere achieves on average 22 FPS with a bitrate of around 8 Mbps.

Onlive [42] was a commercial cloud gaming service that closed in April 2015. It is reported that the system lacked efficient computing resource allocation. Consequently, servers would only take one single-game instance at a time which caused Onlive to expand the number of their servers which proved to be to too expensive to maintain. Onlive achieved 25 fps with a 5 Mbps bitrate as stated on [43].

StreamMyGame [44] is a software made for game streaming that enables users to play Windows games remotely. StreamMyGame provides a client and a server software that users have to install. The server software has to be installed on a Windows computer, which automatically searches and lists the installed games. The client can be installed on both Windows or Linux. Tests [45] showed that it can achieve 25 fps with a 6 Mbps bitrate.

CloudUnion [46] is a popular cloud gaming system in China with a geo-distributed network of data centres. For every data centre there is a gateway that maintains a waiting queue of users wanting to play a game. If the data centre has available computing resources, then a gaming server is launched and the user can start playing. If not, users may choose to wait or access other data centre. The queuing delay may last several hours. CloudUnion supports streaming to web browsers so users do not have to install any software, furthermore, it works on any device and OS with a browser installed. Tests show that CloudUnion achieves 20 fps with a 6 Mbps bitrate as measured on [47].

All of these solutions suggest an adequate remote display performance with a reasonable bitrate. A comparison with the proposed system is presented on the evaluation chapter (section 6.3).

3.3 Interoperability

Existing remote desktop solutions are capable of running applications from multiple operating systems. However, applications from different OSes do not communicate out of the box, as applications running on a single OS do. In order for applications from a giver operating system to interact with other applications from a different operating system, an interoperability component is needed.

Interoperability is the ability of systems to exchange information and use each other's services effectively [48]. Although communication between applications already happens within the same operating system, in this project, applications should be able to interact with other applications not only from the same operating system but also from different operating systems. Moreover, these applications can reside remotely and separate from each other.

The communication between applications or processes occurring within an operating system is called inter process communication. There are several methods that enable this type of communication, namely, pipes, file system, sockets and shared memory mechanisms [49]. A pipe is used as a data channel between two processes where the output of one process is converted into the input of the other. Processes can also make use of the file system to communicate where one process writes to a file and the other process reads that file. Process communication over a network environment is accomplished through sockets where a process can send a data stream to another process on the same computer or to a process residing in another remote computer. For example, a web server process uses sockets to communicate with a browser process on another computer over the internet.

Realizing the type of interactions where an application is able to call another one on a different machine and receive its output is not a trivial task. To analyse existing works that accomplish this, we focused on cyber foraging because they master this type of systems and although it is not their main objective to provide interoperability between different operating systems, it is important to take a look at this field.

A remote desktop allows the access to a remote operating system and its applications by a local device. This means that all of the processing of the applications happens entirely on a remote server or cloud. As a consequence, the applications depend only on the processing power and other computing resources of the remote server and not of the local device. When it becomes beneficial in terms of costs or performance to run an application on a remote server, instead of the local device, it is called cyber foraging. For example, if a mobile phone with very low CPU power needs to apply complex photo effects to 100 images it can be beneficial to, instead of executing this task locally, execute it on a stronger remote server and get the results back. Mobile phones or other resource-constrained devices can benefit from cyber foraging in the following ways:

1. Compute tasks faster

2. Perform tasks that would normally not be feasible to perform on the local device

3. Preserve energy

A remote desktop does not apply cyber foraging implicitly because users can still access Paint even if their local computers can run Paint efficiently. Given this, although the goal of this thesis work is not exactly the same as in cyber foraging, techniques used by cyber foraging related works can be very relevant to this thesis, since communications between applications that are separated from each other is a key issue on the proposed system.

Cyber foraging consists of utilizing remote computing resources to save battery or to enhance performance on local devices. With cyber foraging, resource-constrained devices can offload tasks to servers, computing clouds, or other devices with more computational power (called surrogates). Choosing which remote machine to use and which tasks to run remotely, in order to get the most optimized solution is the most important part of a cyber foraging system and the main focus of cyber foraging related art. The entire process is delineated by the following steps: 1) surrogate discovery; 2) monitoring of the local environment; 3) tasks partitioning into local and remote tasks; 4) scheduling of tasks so that they are only executed remotely when beneficial; 5) remote execution and reception of results. There are a number of factors to take into account for each of these steps. M. Sharifi et. all. [50] did a thorough work classifying and identifying all of these factors which differentiate cyber foraging systems.

Locusts [51] is a framework designed to allow developers to easily integrate cyber foraging into their applications. Locusts is responsible of finding possible surrogates, which is done using UDP over Wi-Fi, to quickly detect them and keep a list of available surrogates throughout the application life time. Developers have access to an API to easily offload tasks to the surrogates. Surrogates and local devices need to have a Locusts software pre-installed and communicate via RPC. Locusts gives developers the responsibility of identifying which tasks should execute locally and remotely, instead of doing it automatically. As a consequence, applications need to be modified to use Locusts, which take developers time and not allowing old applications, not prepared to use it, to explore the capability.

CloneCloud [52] makes use of cloud computing to offload the right portions of applications. It automatically identifies which parts of an application should be executed remotely with no help or modifications needed by the developer. CloneCloud examines the execution condition of the local device (CPU, workload and network performance) and determines which tasks should run remotely. On the cloud side, it provides a virtual machine which mimics the local execution environment so that there are no compatibility issues. These virtual machines run at the application level like the Java VM. This means that it is easy to migrate and to manipulate application executables. CloneCloud actually needs to modify the application executable and so that when it is running, it is able to analyse and choose individual threads to run remotely on the cloud's virtual machine.

MAUI [53] is another work similar to CloneCloud, which takes advantage of code mobility provided

by environments like Java, but it does not automatically partition the application, the developer needs to do it by hand. Nevertheless, MAUI includes a profiler that examines if the task that was allocated to run remotely is actually beneficial or not.

ThinkAir [54] is similar to CloneCloud but instead of using application-layer virtual machines, it uses full operating system virtual machines. It does this by cloning a full Android operating system on the cloud. Thinkair is capable of dynamically create and delete multiple virtual machines on the cloud for on-demand resource allocation and scalability. The application partition is not done automatically, an API is provided. Using this API, developers need to specify which methods should run remotely.

DiET (Distributed Execution Transformer) [55] modifies Java bytecode to offload computationally heavy parts of applications to remote servers. DiET automatically partitions a Java application and selects its heavier methods. Afterwords, it looks for nearby servers to execute them. Since the only software needed to run those methods are the Java VM, any operating system that has it installed can serve as a remote environment. Also, developers do not have to change the code of their application, DiET makes the application automatically distributed by transforming its bytecode. To accomplish this, it uses a set of Java optimization frameworks that allows to analyse bytecode and select which methods should run on a remote server. After that, DiET replaces them with a function that is responsible for executing the original methods on the remote servers and for receiving the response.

Presented works allow communication between processes running on different machines but need applications to be modified. This modification may be implemented manually by a developer or be automatic. Although when it is automatic it does not work for all types of applications, e.g., only applications implemented in Java. Having to modify applications increases developer's work and consumes time. Moreover, it is not assured that it works for all applications. Given this, we can conclude that communications do not happen transparently as required for this thesis.

3.4 Summary

This chapter introduces the state of the art related to the targeted work. These are separated into three sections. The first one describes works that implement Virtual Desktop Infrastructure (VDI). A VDI is a platform that manages and provides remote desktops to users. Most of these works provide a full operating system that is running on a virtual machine made available to users via remote display. The user can then install and run applications in them. These works focus on efficient ways of managing these virtual machines. The second section describes works that implement high performance remote displays. These works provide a cloud gaming service to the users. The performance of the remote displays can be measured in frames per second (FPS) which is the rate of frames that appear on the user screen per second. Because games are very sensitive to the frame rate, these works focus

on maximizing the quality of the remote display. The average frame rate for these works is 23 FPS with a bitrate of 6 Mbps. The third section describes works that focus on communication between processes. One of the discussed techniques was cyber foraging providing communication between applications or processes on different machines. This communication does not occur transparently because applications need to be modified.

4

Design

This section describes the design of the proposed system, called Centroid. Centroid's objective is to deliver a remote desktop to users where they can run applications from multiple operating systems. With this service, users are able to access their computing environment, like applications and files, any time and anywhere. Furthermore, because it is possible access multiple operating systems, users can run their favourite application no matter their operating system. However, remote desktops fail to be adopted as the primary computing environment to run applications comparing to the local computer because they incur several problems. In particular, we observe that the overhead of such systems is very noticeable, for example, users have to wait for a full virtual machine and OS to load before they can use an application which can take several minutes. And, unlike a local computer that is only used by one user at a time, the infrastructures that provide and manage these remote desktops have to serve multiple users. If they cannot efficiently scale, the waiting time increases even more. Furthermore, because applications run remotely, they have to be streamed to the user's device so he can interact with them, this is called remote display. We observe that the quality of the remote display is not suitable for applications more graphically demanding. This restricts the types of applications that users can run, consequently, they prefer using a local computer. Furthermore, although it is possible to run applications on different operating systems, the applications are not able to interact between each other. Moreover, even if these problems the system must also be cost-effective to be used on a real life scenario.

To address these challenges, it was set four requirements for Centroid: communication between applications from different operating systems; scalability; high remote display performance; and reduced costs.

In this section, we propose a design with the objective of overcoming those challenges and achieving the requirements. As such, the design for Centroid is focused on performance, scalability, costs and application communication. Section 4.1 discusses the solution for the performance, scalability and cost

challenges and Section 4.2 discusses the solution for being able to provide communication between applications from different operating systems.

4.1 Overcoming performance, scalability and cost challenges

To better understand the expected usage model it is presented a number of examples bellow describing operations such as opening applications, installing applications and dealing with files.

Launching an application - The first thing that users see when accessing Centroid is the user interface (much like other operating systems). This interface consists of a web application that a user can access from a browser with any device. From this web application, the user will be able to launch applications and access them in the browser. An example of the steps of launching a application would be the following:

1. A user opens the browser and accesses the Centroid web site.
2. The user logs in to the web site and accesses the Centroid web application where he can be chose what application to launch.
3. The user chooses to launch, for example, Paint.
4. The system checks what operating system the application belongs to, in this case, it is Windows.
5. Paint is executed on Windows on the cloud.
6. The screen image is transmitted back to the user to a new browser window.
7. The user uses Paint normally on that browser window.

Installing an application - A similar process has to happen when a user wants to install an application. The following example demonstrates the steps of installing the Microsoft Word application on Windows:

1. User logs in and accesses the Centroid web application from a web browser.
2. User clicks on a button to install a new application and is prompted for the installation files.
3. User wants to install Microsoft Word on Windows so the user sends the proper setup file.
4. Microsoft Word setup is executed in Windows on the cloud.
6. The screen image is transmitted back to the user to a new browser window where he can follow the installation process.
7. After the installation is done, a Microsoft Word icon appears on Centroid web application where the user can click to execute it.

Opening files - Now, if a user wants, for example, to open a Word document file with Microsoft Word, he would have to execute that application from the Centroid web application which would launch a Windows operating system on the cloud to execute it. The only way that the Word application could have access to the user file, is if the user sends the file to that remote Windows environment so that

Word can open it. This is very impractical, especially when users want to open multiple files, furthermore, users would have to have that files on their local computer so they could send them which goes against the cloud model. So, Centroid features a cloud distributed file system very similar to Google Drive or Dropbox. Therefore, the Centroid web application not only shows what applications the user has installed but also what files he has on Centroid. If a user wants to open a file on Microsoft Word the steps would be:

1. User logs in and access the Centroid web application where he can see what application he has installed and also his files.

2. The user chooses a Word document file to open and tells Centroid to open it with Microsoft Word.

3. The system checks what operating system belongs to Word, in this case, it is Windows and launches it on the cloud.

4. The system opens that user Word document file with Microsoft Word and transmits back the screen image.

Centroid is hosted on the cloud which means users access it using the internet. A local computer has to work for only one user at a time, but Centroid has to work for multiple users at the same time. Some of the advantages that Centroid has over a local computer is that it can be accessed any time anywhere over the internet and it runs applications from multiple operating systems transparently. But, this advantages come with challenges, and if those are not overcome, there is no advantage in using Centroid. The challenges about using multiple different operating systems are discussed in section 4.2. In this section we discuss the solutions we devised to overcome the challenges that come with building an infrastructure capable of providing and managing efficiently remote desktops. These challenges include:

- Performance: Operations such as opening an application on Centroid have an inherent overhead comparing to a local OS. For example, if people have to wait 10 times more for an application to open on Centroid that it would on the local OS than the probability of Centroid being used decreases. This is the same for installing an application, opening files, etc. So, the challenge is to design an architecture capable of minimizing this overhead.

- Scalability: Any system that will be used by multiple users at the same type is prone to scalability issues and Centroid is no different. What happens when two users are active is very different to what happens when 2 millions users are active. Centroid has to be able to scale which means maintain the performance when user numbers increase. Having to manage multiple virtual machines, applications, servers and easily scale is a challenge that has to be taken into account on the design.

- Cost optimization: Cloud services have costs and the billing is according to virtual machine time

usage, disk space used, data transfers, among others. Centroid uses these services so it has all of these costs. The costs will be then supported by the users that pay for using Centroid. Consequently, the price needs to be competitive to other works and even ideally to buying an high-end PC. The challenge is to find where we can cut costs and design an effective solution.

The design we devised for Centroid's infrastructure is a sweet spot between these three items. The next section describes its architecture.

4.1.1 Architecture

The architecture features several components each one responsible for a single task. This separation of concerns not only accelerates development, but it makes it easier to scale and optimize the performance of the system. Figure 4.1 illustrates an overview of the system architecture.

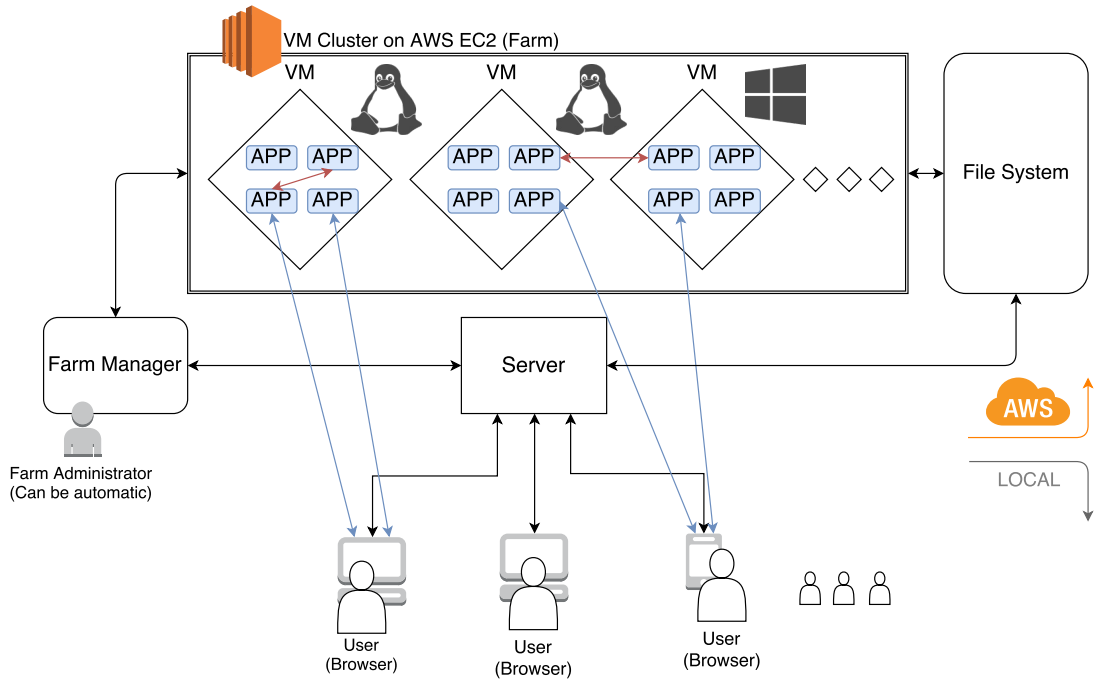

Figure 4.1: Overview of Centroid's architecture

The rest of this section describes the various components (illustrated in the architecture figure 4.1) divided by their major area of influence, namely:

- Application Execution Environments (4.1.2): Describes how **applications** (illustrated in blue on the architecture figure 4.1) run in separate environments and can be efficiently stored, retrieved and started with the objective of optimizing performance, costs and also be easier to scale the system.
- Efficient Resource Management (4.1.3): Describes how **VMs** will have different configurations depending on the various applications types with the objective of minimizing costs. It also describes

how the **farm manager** component will help with the scalability and performance of the system by managing the VMs efficiently.

- User Accessed Resources (4.1.4): Describes the design of the **server** component, *file system* and application access and usage including the **remote display** illustrated by the blue arrows on the architecture figure 4.1. The design of these components has the goal to promote mainly the scalability and performance of the system.

4.1.2 Application Execution Environments

Centroid applications must run isolated from each other. Isolation aims to prevent malicious or ill-behaved applications from accessing or interfering with the execution state of other applications. Such state tends to include both volatile and persistent data. Normally, in traditional operating systems, isolation is achieved by a combination of memory protection and access control mechanisms. However, since Centroid applications may be targeted for different OSes, we cannot rely on the mechanisms of a single OS for isolation enforcement. A straightforward alternative to isolating applications for different OSes is to run each application on a single VM bootstrapped to the OS required by the guest application. However, since cloud providers normally get paid per VM, running an application per VM would bloat the prices to be charged to Centroid end-users. A more conservative approach is to share VMs between multiple applications, such that applications targeting the same operating system run in the same VM, therefore reducing costs. However, in popular OSes like Windows or Linux, applications tend to have inter-dependencies (e.g., by sharing common libraries or requiring pre-existing tools) which may generate conflicts upon installation on shared environments. Furthermore, security breaches may occur since in these OSes (as opposed to Android OS, for example), applications run under the same UID. As a result, a malicious application could easily delete persistent user data generated by other applications, for example. Clearly, this approach offers insufficient isolation guarantees for Centroid applications.

To find a sweet spot between security and cost, we adopt a solution in which applications run inside individual containers. This solution will also allow us to scale and start, stop and save applications efficiently.

Containers - By containerizing applications, we can run multiple applications of different users on the same VM in isolation. The applications illustrated on the architecture figure 4.1 are actually containerized. Sharing the same VM requires applications to be packaged for a common OS and the OS must be enhanced with container support. For Linux, containers are supported by the popular Docker framework; upcoming Windows versions provide container support. Regarding security, containers lie in a middle ground between VMs and standard OS. VMs provide a more robust solution than containers since the former depend on smaller trusted computing base than the latter. However, containers are fully self-contained in terms of required application code and implement individual sandboxes for applications,

therefore addressing the limitations of standard OSes. Furthermore, containers provide good portability and are lightweight enough to be set up, moved around, and stored efficiently. Lastly, given that they are much lighter than VMs it is less expensive to store a container image than a VM image. Thus, we adopt containers in Centroid. Whenever a user launches an application through the browser, Centroid selects a VM from the VM pool, sets up a new container inside that VM and executes the application. The resulting output is seamlessly streamed back and displayed on the browser. This solution is the most effective to balance the security requirement with performance, costs and scalability. It will always have an inherent overhead comparing to launching an application on a local operating system because before the applications are executed, the containers need to be run themselves. But because containers do this so efficiently, the overhead will be hardly noticeable.

4.1.3 Efficient Resource Management

Virtual machines can have a very ample range of configurations offered by the cloud provider. So, we can optimize the virtual machines allocated resources to our specific use case. Furthermore, we can also optimize the number of running VMs that are active at a time. For example, it is not needed to run 100 virtual machines if only 2 users are active. This section describes our proposed solution to optimize virtual machines costs and their management.

VMs - By leveraging containers it is possible to share a single VM between multiple application instances, thereby cutting down expenses. These VMs are illustrated on the architecture figure 4.1 on the VM cluster. Costs can be further reduced by realizing that the price to be charged per VM depends on the VM's hardware specification. In particular, prices tend to increase with the amount of hardware resources allocated by the VM. Thus, since not every application requires the same amount of resources to execute, we can assign applications with different hardware demands to different VMs. In particular, we observe that applications come in a variety of shapes and they all have different computation demands. They can be very graphically dynamic or have no graphics and just run on a terminal. If we allocate a virtual machine with an expensive GPU to run a terminal application, we would be wasting resources and money. With this in mind, we divided applications into three categories as shown on Table 4.1: 1) Graphical; 2) Office; 3) Textual.

Type	Examples	Graphics	Needs real-time response
Graphical	Games, Auto-cad, 3D design	Heavy	Yes
Office	Word, Excel, PDF Reader	Light	Yes but tolerates delays
Textual	Python, Vim, NodeJS	Non-existent	Not required

Table 4.1: Application types in Centroid: Graphical, Office and Textual

The Graphical type refers to graphically demanding applications that are sensitive to latency. Any

noticeable delays make the applications unusable. The Office type refers to any application that uses mostly text and graphics that require low GPU attention. Although it requires a real-time response, infrequent latency is tolerated. The Textual type refers to console applications, it includes any application that runs on a terminal (shell) and does not use a GUI. The response can contain a constant delay but it should be bellow a practical value.

Based on the resource demands by different application types, we define possible virtual machine configurations allocating different amounts of resources (and therefore charging different prices). Our goal is to reduce costs by allocating exclusively resources that are needed depending on the types of applications. In particular, we define two types of virtual machines: High-Quality (HQ) and Low-Quality (LQ). This choice was made taking into account the available virtual machines types on Amazon Web Services (AWS). This is the cloud provider we chose to host all Centroid's infrastructure. Not only because it is one of the most popular cloud providers but because they provide virtual machines equipped with a GPU that is optimized for streaming. The two virtual machines types we defined for Centroid are described on Table 4.2.

VM Type	**Applications types allowed**	**Maximum number of apps running simultaneously**	**VM specifications**
Low-Quality (LQ)	Textual	8	1-core Intel Xeon Family 2.5 GHz 1GiB RAM
High-Quality (HQ)	Office; Graphical	Office - 6; Graphical - 2	GPU NVIDIA GRID K520 with 4GB of video memory. 8-core Intel Xeon E5-2670 3.6GHz. 15 GiB RAM.

Table 4.2: VM types supported by Centroid: Low-Quality and High-Quality.

We defined that High-Quality VMs can support as much as 6 Office application instances or 2 Graphical applications running simultaneously. Keep in mind that each application runs in a separate container. Nevertheless, other application setups are possible, e.g., 1 Graphical and 3 Office applications. Taking into account the VM specifications of the two types, it is easy to see why HQ virtual machines are more economically expensive than LQ virtual machines. Consequently, Graphical applications will be the most expensive applications to run, followed by Office applications on a scale of 3:1. Textual applications will be the least expensive to run because of its VM cheaper price. We also defined Low-Quality VMs can support 8 Textual applications.

This gives users three types of environments on where to run their applications that they can choose according to their needs and budget. For example, if a user can tolerate a small lag, he is allowed to run a game, that is considered a Graphical application, on an Office environment to save money. If he needs a high end environment to run a console application, for example, to run a machine learning algorithm,

that uses more CPU or even the GPU, he can choose to run it on a Graphical environment, even though the app has no graphics. Users can even change the type of application after they install it. This allows them to have a finer grain control on the price and environment of their applications. We defined the types of virtual machines possible, but how they will be managed still presents a challenge. To describe the management of these virtual machines it is introduced the farm manager.

Farm manager - The farm manager is a component made to facilitate cluster operations and is illustrated on the architecture figure (4.1) on the left. The server communicates with it every time a user requests to run an application. For an application to be deployed it needs an available spot on a VM and the farm manager is responsible for efficiently providing this information. A farm administrator supervises the health of the farm, making sure that there are enough available VMs for the system to run smoothly. He manages the virtual machines life-cycle by starting or terminating them. This job can be done manually by a human or automatically by a machine. Nonetheless, it is not a trivial task, predicting exactly how many VMs should be available at a given time raises numerous interesting challenges. It is dependent of several factors including:

- Number of users connected at the present moment;
- Number of users that will be connected in a close future;
- Users' preferences and installed applications.

This is a supply and demand problem. We can relate it, for example, with the electrical grid, too much supply and low demand raises unnecessary infrastructure costs but, too much demand and low supply restrains users from electricity. The complexity raises even more when we add different types of applications and different VMs configurations. If the current connected users use more frequently Textual applications, then it would be probably better to have more Low-Quality VMs available than High-Quality and vice-versa. In this version of Centroid, the farm manager administration is done manually but it would be an interesting subject for other works to create an automated administrator. In terms of scalability and performance, the farm manager can have one or multiple instances and they do not depend on each other. With a load balancer provided by the cloud provider it is very easy to distribute traffic between each farm manager instance. Globally, no matter how many instances of the farm manager are running, they always access the same database. This database stores information about the virtual machines relevant for the farm manager to do its job. The farm manager uses DynamoDB which is a scalable and high performance noSQL database that is provided by AWS. The fields include: ID; Occupancy (how many applications running); Config Type (LQ or HQ); OS (Windows or Linux); IP (ip address of the VM). With this fields the farm manager can check what VM is available according to the applications specifications. Once we introduce the server component on the the next section, it is further explained this process and how the farm manager is used by the server inside Centroid.

4.1.4 User Accessed Resources

The first component users interact is the server which hosts a web application that is the Centroid's user interface. This is where users, for example, order the execution of an application. Once the application is running, users use it directly on the browser and are able to access a their Centroid file system available on every application. This section describes the design of these various components as well as their part on the whole system.

Server - The server (component in the middle of the architecture figure 4.1) hosts a website that users access to interact with Centroid. This website displays a simple user interface with similar goals as expected from a conventional operating system UI. This includes: listing the files and applications belonging to a user, offering a way to open or run them, and providing a method for creating new files and installing new applications. Users are able to use this interface from any web browser by logging in with their account. When a user performs an action on the website, the server is responsible to address it or to delegate it to other component and giving back the response to the user. Apart from the remote display, the server is the only component that communicates directly with the user. The list bellow describes the steps the system performs from the moment the user clicks on the website to open an application until he can see and interact with the application on his screen:

1. Server gets a request to open an application.
2. Server checks a database for the application information such as the type of application (Graphical, Office or Textual), the OS it belongs to and other information that can be relevant.
3. Server sends that information (app type and OS) to the the farm manager and asks where is an available virtual machine that can host that application.
4. The farm manager checks for an available VM on its database and replies to the server with that virtual machine information such as its IP.
5. With that information, the server communicates with the virtual machine by sending it the information about the app and orders it to execute the application container.
6. The virtual machine responds to the server with a url link that can be accessed to see the application.
7. The server finally responds to the user browser with that url and the browser automatically opens it. The user now sees the application by remote display on the browser.

As can be observed in the list, there are two databases: the applications database (accessed by the server) and the virtual machines database (accessed by the farm manager). The information about the applications is also stored on DynamoDB. The applications database stores information about every

application and contains the following fields: ID; name; operating system (Windows or Linux); type (Graphical, Office or Textual); Exec (command to be executed to open the app); Setup (information about how to install the application); Status (running or stopped); Location (where the application container is stored). The fields are self-explanatory and it is easy to see where they fit on the list above.

The system needs to accommodate multiple users and be easy to scale at any time. For the databases we chose DynamoDB because of its scalability and performance. The farm manager, virtual machines and containers are easy to scale at any time as described on previous sections. But the server on the other hand, can be harder to scale. Basically the server is a software that runs on a machine accessible over internet. In this case, the server will run on a virtual machine on the cloud. (Note that this virtual machine has nothing to do with the virtual machines described before for the remote desktops, this is just a machine to host a web server.) To scale the server means instantiating more virtual machines running the server software. Then a load balancer redirects the traffic between the server instances. The hard part is with user sessions, generally sessions are stored on the memory of each server or on a database and neither one of these approaches are fast and easy to scale at the same time. When a user logs in and one server instance saves the session in memory, than all the other servers instances can not access that session because they do not share memory, they are not even in the same virtual machine. The problem arrives when the user then performs a request and it is redirected by the load balancer to other server instance (because of traffic optimization reasons) and this instance does not have the session stored. It will then respond with a forbidden status and the operation is not executed. The most common solutions for this is to always have only one instance of the server on one virtual machine and just upgrade that one virtual machine. But there is a limit to what a single virtual machine can do, it is not scalable. Another solution is to prevent the load balancer to redirect the user to another server instance, keep the user always on one server instance. The problem with this solution is that the traffic will not be optimized, if multiple users share one server instance and if they all are heavy usage users and make a lot of requests then that server instance can fail, this would not happen if the traffic can always be redirected. So, saving sessions in memory can make the server very hard to scale. The other common way to store sessions is in databases. In this case, every time there is a user request, the server instance needs to access the database for the session. This will amount to a lot of databases requests. A database request has costs and is time consuming. And as fast as databases can be, this amount of requests will always have an overhead over accessing the memory. This solution is not scalable also. A recent and more scalable solution comes with the advent of mobile applications: tokens. Tokens save the session information and travel with every request. When a user logs in, the server ciphers the user information with a key, only known by Centroid, and stores this ciphered information on the token. Afterwards the token is sent to the user (browser). Every time the browser sends a request to the server it attaches the token. The server then deciphers the token with

the key and accesses the session info. Because every server instance shares the same knowledge of the private key, the session information is always accessed by every instance making the server very easy to scale. Thus, we use tokens in Centroid.

File system - Applications run separated from each other within containers which means they do not share a common file system. This can be restrictive in some cases, for example, when using two applications that interact with the same files like an IDE and a compiler. The IDE have access to the file that it is editing but when trying to call the compiler, it does not have access to that specific file. With this in mind we decided to implement a distributed file system. This file system resembles to Dropbox or Google Drive. The objective is for every virtual machine to have access to the same file system. The user folder is then mounted in his containers upon application start. Part of this work was made in the scope of a system called Storekeeper. Storekeeper provides a distributed file system that uses cloud storage services, such as Google Drive and Dropbox, to store ciphered files. However, since all of the infrastructure is hosted on AWS, it was an unnecessary overhead to provide files from those storage services instead of directly out of S3 (Simple Storage Service) provided by AWS. Thus, in this work, the distributed file system is going to be the hosted on S3. To mount the file system on the virtual machines we adapt an open-source solution named s3fs [56].

Application Access and Usage - Because of the various types of applications that we defined such as Graphical, Office and Textual, users have different ways of accessing them as well. For Graphical and Office applications, Centroid provides a remote display on the browser. For textual applications, Centroid provides a web application which emulates a terminal on the browser. The reason behind this is cost optimization, Textual applications are always terminal applications so there was no reason for a remote display in this case. If we emulate a terminal on the browser, then we just have to stream text and not the desktop images.

Textual applications run on a terminal, so Centroid has to be able to attach their output and input to the user browser. In the browser there is an web application that emulates a terminal running that is capable of processing that output like a real terminal and capable of getting input from the user. The first challenge is how to listen and attach to the application output and input. The easier way is to have a script on the containers that executes or forks the application and attaches to its stdout, stdin and stderr pipes and stream it to the user. The problem is that there is a some differences when applications are executed from a terminal or from a system or a fork call. Some applications behave differently when they are executed from a terminal, this means output colors, format, verbose, etc then when they are not. Because we want the user to see the application as if it was running on a terminal, we can not just execute the application from a script, we have to execute it with a pseudo-terminal (pty). A pseudo-terminal is a software that emulates a real terminal making the applications ”think” they are being executed from a terminal. Using a pseudo-terminal we can easily read the application output and

write input exactly like it were running on a real terminal. On the browser, it is used a terminal written in javascript that is capable of emulating a real terminal in terms of showing the applications output to the user and also sending its input. The communication between browser terminal window and the pseudo-terminal is can be achieved with web-sockets.

For the Graphical and Office applications, they can be executed with a fork or system exec call because we only want to grab the desktop image and not its processes output. Because they run remotely, the screen image has to be streamed to the user, this is called the remote display. Providing a remote display with quality is a big challenge as described on chapters 2 and 3. This is another possible setback to using Centroid, and the challenge is not so much designing but it is to implement a remote display capable of overcoming this problem. Not only this, but the applications need to be streamed to the browser and not to a software which aggravates the problem. We use the Nvidia Grid GPU cards available on the virtual machines to capture and encode the screen images. Then, they are sent via web-sockets to the browser. On the browser, they are decoded and displayed to the user. The mouse and keyboard are also sent back via websockets to the applications. The main contributions for the remote display are presented on the implementation (section 5.4).

The desktop images or the terminal text is always streamed directly from the applications to the users' browsers. There is no component in the middle, this assures that there is no bottlenecks, every application and every access is distributed. Thus, this increases performance and makes the system easier to scale no matter how many applications are or will be running. The next section describes the design for overcoming interoperability challenges.

4.2 Overcoming interoperability challenges

A fundamental part of an operating system is the communication between applications. But with the choice of having applications from multiple operating systems running on containers possibly on different virtual machines effectively breaks every communications between apps. Consider the following example: A user has two applications installed on Centroid, Geany (which is an IDE similar to eclipse but simpler) and gcc. When he clicks to run Geany, Centroid runs the respective container and on a HQ VM. Then the user clicks to run gcc and Centroid runs the respective container on a LQ VM. Now we have two containerized applications, running on different virtual machines. Geany usually uses gcc for compilation but with this configuration it would be impossible for Geany to use gcc out of the box. Other examples include opening automatically a url on a browser that is running on another container when clicking it inside another app. This would not happen if we had the applications installed on our local operating system so this would be a major disadvantage for Centroid users. To tackle this issue, we first need to specify exactly what a Centroid application is, what it represents and what it consists of. This is

what we defined:

1. Each and every application is standalone: has all the dependencies installed to run by itself without any errors.
2. An application can include a subset of processes. These processes are not considered separated applications, they are considered part of that one application. The same is considered for threads.
3. Direct communication between two applications happens when one explicitly executes the other.
4. Each application runs in its own container. This is to simplify things for the user and not for us. To prove this, every container is customizable and users are allowed to manually install any programs they want in one container. But Centroid will keep treating that container as a single application.

For example, when users open a new tab on Google Chrome, it runs on a different process. On Centroid, this would be a single application and would run on a single container. The processes from Google Chrome tabs would run along with the main process on the same container, not separated. Another example, users from within the Visual Studio Code application can call nodeJS to run a script. But Centroid defines them as separated applications because they are standalone, meaning that Visual Studio Code is usable without nodeJS and vice-versa, they are not dependent in any way. Thus, they would run on different containers. As mentioned before, if users want to run them on a single container they can also do it by customizing the container. But the recommended way is to have them separated so that they can be used by other applications, for example, nodeJS could be used by Visual Studio Code IDE or by Atom IDE at the same time. Much like users would on a personal computer OS.

As mentioned before, because applications are containerized and possibly run on different virtual machines they can not communicate with each other out of the box. They do not even know the existence of each other. On the other hand, an application on a local OS knows all this information and can call other applications or even instruct the system to open a file with a specific application. The challenge is to give Centroid users the same ability as they would have on a local OS. To further aggravate the problem, applications can be hosted by different operating systems. So, not only we need to have communication channels between applications, we have to take into account that operating systems have different ways of executing and managing applications. As explained above, we define communication between applications when one (let's call it the main application) executes another (let's call it the executed application). The main application is then able to input data and get the output from the executed application. And, of course, the executed application has to be able to receive the input and retrieve output to the main application. This would be similar to a pipe operation on a conventional operating system. So, for this to succeed, two things have to happen:

1. The main application has to know that the executed application exists (so it is able to successfully call it).

2. A communication channel has to exist between the two (so data can be transmitted between each other).

Centroid needs to be aware that one application is trying to execute another one that the user has. This is because Centroid needs to check that the executed application is running and is accessible so it can create a channel between the two apps. If the application is not running, then Centroid has to execute it to make sure the executed application is running and is accessible. Note that this process is similar to a user opening the application from the web site, but in this case is another application instead of a user. We also have to take into account that applications may send arguments when calling another one, for example, Geany IDE executes ”gcc -Wall hello.c -o hello”. Not only gcc has to be executed with ”-Wall hello.c -o hello”, it has to have access to that file (hello.c). And this also means sending the working directory so that gcc knows which folder that file, that Geany is editing, belongs to. So, to summarize let’s take the Geany example and go through an overview of the whole process:

1. A user is using Geany editing hello.c, a file that belongs to the user’s file system on Centroid.
2. The user clicks on the compile button which calls ”gcc -Wall hello.c -o hello”.
3. This triggers Centroid to check if gcc is running or if it needs to be opened.
4. Let’s say that it is not running. Then, Centroid needs to do the same steps as if gcc was executed from the Centroid website by a user: 1) check OS and check app type; 2) get info from the farm manager on what VM to run the app; 3) instruct that VM to and run the container (with access to the user’s file system).
5. Centroid also instructs the VM that the application is to be executed with the following arguments ”-Wall hello.c -o hello”.
6. The application (gcc) is executed with the arguments;
7. Centroid attaches to gcc’s stdout, stdin and stderr.
8. The output is returned to Geany.
9. Geany displays the output to the user as it normally does.

The list above describes a briefly overview of what we want to happen, the challenging part is to design a solution that fits what we want. It has to take into account scalability and performance so communications happen efficiently. To accomplish this, there are several steps that we need to overcome:

(a) Making other applications visible/executable from the point of view of the main application even if they are not running on the same container.

(b) Having a software that attaches or can read/write the input and output of both applications and is able to send data between them.

(c) This software (from the previous item) has to distinguish different operating systems and behave differently according to their specifications.

(d) Send data between applications efficiently, in terms of performance and scalability.

On a conventional OS, when one application needs to execute another one and get its output, it just executes a command that opens the application and the operating system is responsible for all the process. It is responsible for finding the application corresponding to that command and if it finds it, executes it, and does all the connections necessary for the main application to get the output from the executed application. On Centroid we defined that any direct communication between applications happens when one explicitly executes the other. This means that it calls the command that would execute the application if it resided on the same OS environment, in this case the same container. Because it does not reside on the same container, Centroid needs to make sure that the application to be executed is somehow visible or executable on that environment. And, every application must be executable from any other application. These applications all belong to the same user of course. Furthermore, Centroid has to be aware that an application intends to execute another one and have a script ready to make the communication between them.

Essentially, we need to devise a solution to make sure that when an application executes another application, it triggers a script that signals Centroid of this. The most straightforward solution is to ”listen” for exec system calls. Any time an application does this call, Centroid intercepts it, checks if this exec is intended for any of the user’s applications and if it is, execute that Centroid application. If it is not, do nothing and let the exec continue normally. Although this may be the first solution that we thought, there is not a straightforward or official way of intercepting exec calls, we would have to rely on third-party software to do this but it is too error prone. Nor it is efficient to intercept every single exec call that an application makes. So another solution needed to be found.

The process to find an effective solution was to think in terms of execution environment instead of application. The execution environment, more specifically the container, runs on a conventional operating system and we can take advantage of what we know about them. We know that if, for example, we install two applications on the same container, they can call each other. So, if we can make the container ”think” that the executed application is also installed, then the main application can easily call it like it normally would conventionally. We do not actually want to install every application that a user has on every container of course, we just want to emulate that every application is installed. To do this, first it is necessary to know how conventional operating systems check if an application exists and is executable. It is also important to note that Centroid knows how to execute every application, it knows the global

command or the path to the executable file. So, on a conventional OS, if an application wants to execute *gcc* it does a similar instruction to *exec("gcc")*. The operating system then checks the PATH variable, which tells the OS which directories to search for executable files. If it finds *gcc*, the OS executes it. Taking advantage of this caveat, if we put an executable file called *gcc*, the operating system thinks it is the real application and executes that file. It just has to be named *gcc*. So, if we put a number of files that have the same name as the real apps we can make the container "think" that this applications exist and can be executed. But these files that "emulate" the real applications are just small scripts. These scripts are responsible for signaling Centroid that a determined app needs to be executed. This will trigger Centroid to run that app. The objective is to make the script, not only signal Centroid, but also behave exactly like it would if it were the executed application. So that, from the perspective of the applications, they are interacting with each other like they were installed on the same operating system or container. But instead it is just a script acting like the real apps. To accomplish this objective the input and output of each application needs to be imitated.

Whenever a Centroid application calls another Centroid application that belongs to a single user, it is in fact calling a script that "impersonates" the real application. Figure 4.2 illustrates this process.

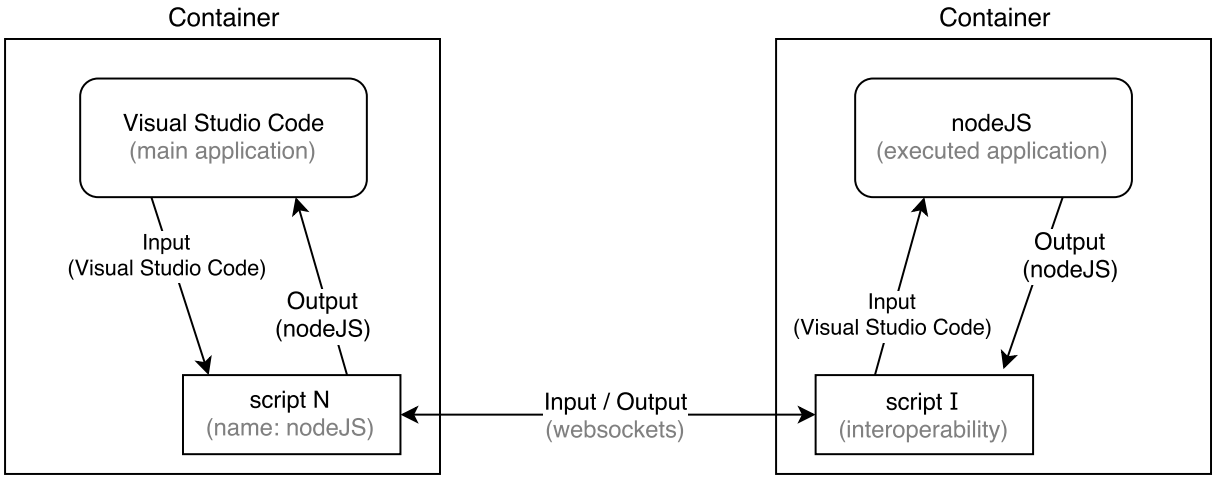

Figure 4.2: Example of communication between two applications. Script N behaves as if it is the nodeJS application. Script I behaves as if it is the Visual Studio Code application.

In the example of the figure 4.2 there are two scripts: script N and script I. The stdout of script N mimics the data that comes out the stdout of the nodeJS app in order for the Visual Studio Code to read it. And the stdin of nodeJS actually receives the same information that script N receives in its stdin (that is written to by Visual Studio Code). This is why the scripts act exactly as the real applications.

The script that "impersonates" the executed applications signals Centroid that this specific application needs to be running. Not only that but because the main application now thinks that the script is the real executed application, the script has to act exactly like it for the interaction to work smoothly. This means that the script needs to: 1) be able to receive input from the main application; 2) redirect it to the

executed application on another container; 3) get the output from the executed application; 4) retrieve that output to main application. The first step is accomplished already because the main application thinks that the script is the real application and sends input to it. But the second step is now a challenge, to accomplish it, another script has to be running in the container of the executed application. These two scripts communicate with each other through websockets passing through input and output from each application. One script acts as if it is the executed application and the other script acts as if it is the main application. For example, consider that Visual Studio Code is running on a container and nodeJS is running on another container, like illustrated on Figure 4.2. In the Visual Studio container there is a Centroid script called ”node” (which is the command to execute nodeJS) and its folder is on the PATH variable. On the nodeJS container, there is a Centroid interoperability script ready for a connection. If the user clicks the run button on Visual Studio Code, it does a similar instruction to ”exec(node hello.js)”. This command makes the operating system call node on that container with the argument hello.js. Because there is a Centroid script named node the OS calls that script. The script communicates with Centroid server and waits for a response to where is the real nodeJS running. After the response, that script connects to the other Centroid interoperability script on the nodeJS container. It sends the arguments (hello.js) and the working directory. The script on the nodeJS container calls nodeJS with that arguments and working directory. Whenever Visual Studio Code inputs something or the nodeJS outputs something this is communicated between the two scripts and mimic that action.

We also have to keep in mind that Centroid has different applications types. Mainly Textual, Office and Graphical. These applications may have different ways of input and output. If the main application executes a Graphical or Office application, not only it may need to access the stdout or stderr output but the user may have to see it on the browser via remote display. For example, if a user is using a Pdf application and clicks on a website link. That link has to be opened by a browser. If he has Firefox installed on another container, the Firefox application needs to run on Centroid and be displayed to the user. So we defined for terminal applications that the input and output is to be redirected to the main application. And for Graphical and Office applications, the same happens plus the desktop display is streamed to the user too. To accomplish this, we can adapt the same procedure that happens when a user opens an app via the Centroid website. The only difference is that instead of the end-consumer being the user, it is the main application, although in the case of a Graphical and Office applications, the remote-display is not consumed by the main application but by the user. Figure 4.3 better illustrates the comparison.

In figure 4.3, there are illustrated 4 situations: U1, U2, M1 and M2. All of these situations describe the steps Centroid takes when executing applications. The difference from the U and M is that the end-consumer is an application (in M) instead of an user (in U). U1 and M1 refer to executing Textual application. Comparing the U1 and M1 situations, it can be observed that Centroid’s steps are exactly

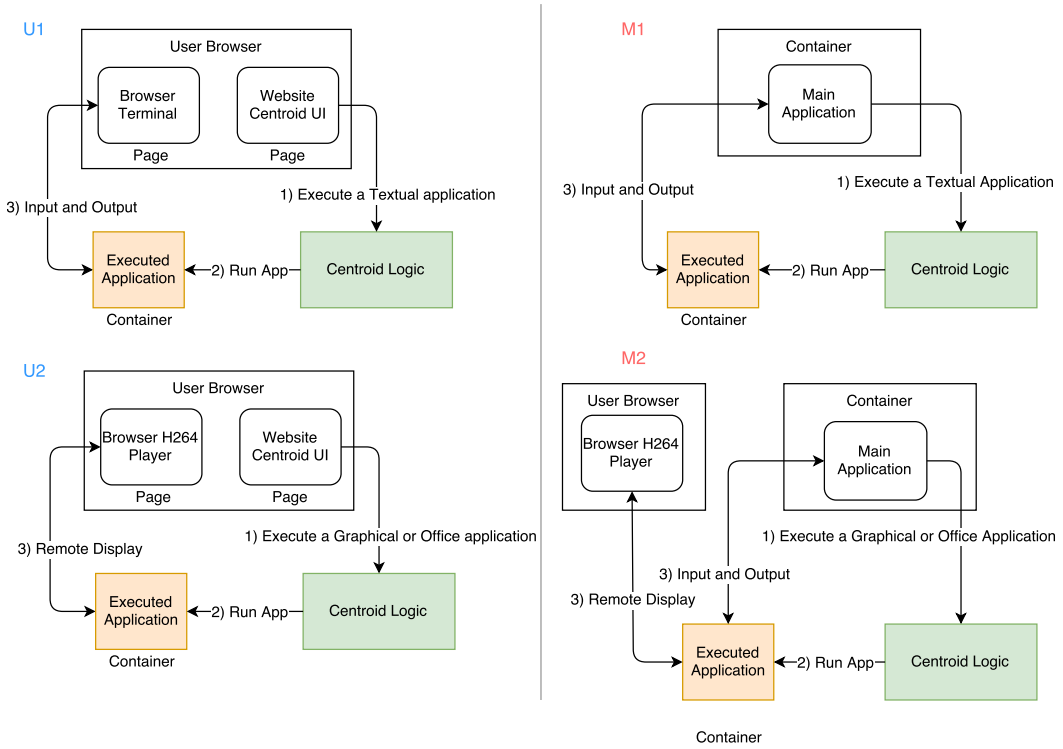

Figure 4.3: Comparison between the end-consumer of an executed application being the user or another application. U1 - User consuming a Textual application. U2 - User consuming a Graphical or Office application. M1 - Main application consuming a Textual executed application. M2 - Main application consuming a Graphical or Office executed application. Note that the Centroid Logic is always the same.

the same. U2 and M2 refer to executing Office or Graphical applications. Comparing the U2 and M2 situations, it can be observed that the only difference is that the input and output of the executed application is redirected to the main application. The other steps are exactly the same.

For Textual applications there is yet another problem. Executing them from a terminal may produce a different output then executing them from another type of application. If gcc is executed from a terminal, the output comes with text formatting like colors. If it is executed from another type of application the output comes with no text formatting, just plain text. Figure 4.4 illustrates the output of gcc if it is not being executed by a terminal. And figure 4.5 illustrates the ouput of gcc if it is being executed by a terminal, it also shows that output displayed on a real terminal.

Like it was mentioned above, whenever a user wants to use a Textual application, Centroid always run that application on a pseudo-terminal (pty) because the output is being displayed on an emulated terminal on the user browser. So, in this case, we want it to have text styling and all the options it would have if it was running on a real terminal. But for application communications, it depends if the main application is a terminal (or uses a terminal) or not. If the main application is a terminal then the process is exactly the same, Centroid runs the executed application using a pseudo-terminal. But if the main application is not a terminal or it does not use terminals then Centroid just needs to execute the

```
Error: Command failed: gcc -Wall hello.c -o hello
hello.c:5:1: warning: return type defaults to 'int' [-Wimplicit-int]
 main()
 ^
hello.c: In function 'main':
hello.c:6:2: error: unknown type name 'asddsa'
 {asddsa
  ^
hello.c:7:12: error: expected declaration specifiers or '...' before string constant
     printf("Hello World");
            ^
```

Figure 4.4: Output of gcc if it is being executed by an application that is not a terminal.

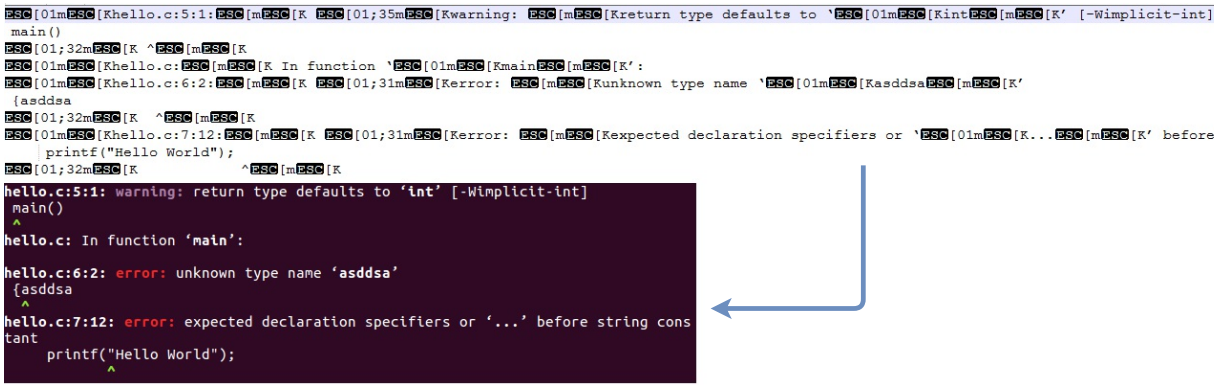

Figure 4.5: Output of gcc if it is being executed by a terminal.

application without the use of a pseudo-terminal.

When an application is a Graphical or Office applications, there is not much difference, Centroid knows the executable path to all application and it just needs to run them. The process is the same for both OSes. For Textual applications, if they are to run on a pseudo-terminal, then there is a difference. On Linux there is a built-in forkpty() function that can be used to run pseudo-terminals on Linux. But on Windows there is not. We will use a third-party tool called winpty that provides a similar concept on Windows.

By leveraging the logic already designed for opening applications to be consumed by the user, we use the same logic that has already been optimized for performance, scalability and costs on applications interoperability. The only major difference is that instead of the executed application send the output to the user browser, it sends to the main applications. And much like the user procedure, the data are sent directly from one application to the other without passing through any more components. This ensures scalability and performance.

Taking advantage of the different operating systems PATH variables and execution operations, we were able to design a solution that enables applications to communicate seamlessly with other applications running on a completely different environment (container, vm and OS-wise). This ability to interact with other different separated environments is what we defined as **interoperability**.

4.3 Summary

This chapter describes Centroid's design. Firstly, it is introduced the architecture where it is delineated different components. These components include containers, virtual machines, farm manager, file system and application access. By using containers we can run multiple applications belonging to different users on the same virtual machine. We divided applications into three categories: Textual, Office and Graphical. Because virtual machines have different possible configurations and consequently different prices we defined two types of virtual machines, Low-Quality (LQ) and High-Quality (HQ). To efficiently manage the virtual machines and containers, it was needed to create a special component that we called farm manager. It was also described that the server hosts a web application for the user to interact with Centroid, the file system that every application has access to and how the different application types could be accessed, Textual via an emulated terminal on the browser and the Office and Graphical through remote display also on the browser. Finally, it was described how applications communicate with each other taking advantage of the different operating systems services. These happens by simulating that all the applications are installed on the containers, but these applications are in fact just a script that notify Centroid to execute the real application. Then when the real application is executed it is created a channel with websockets between both applications. The input and output is redirected between them.

5

Implementation

This chapter describes the implementation of the various Centroid components designed on Chapter 4. Each component has its responsibility. To accomplish a task, such as executing an application, they need to cooperate. The next section describes how they communicate with each other.

5.1 Interaction Between Components

Components need to communicate with each other. To accomplish this we implemented a web server on every component. Figure 5.1 illustrates the different web servers and how they interact.

Centroid's server, farm manager, virtual machines and containers have web servers implemented so they can be reached. Every one of those web servers implements a REST API. A REST API provides a set of urls that when accessed perform a certain operation. For example, after a user accesses Centroid's browser interface, it needs to list the user's applications. To do this the browser interface needs to communicate with the server requesting this list. This is done by performing an http GET request to the API url route "/apps". This url is prefixed with the server ip. Table 5.1 exemplifies the rest of the server API related to applications. All of the components' web servers provide REST APIs related to their responsibility. This web servers were implemented with nodeJS. The next section describes the implementation of the farm manager component.

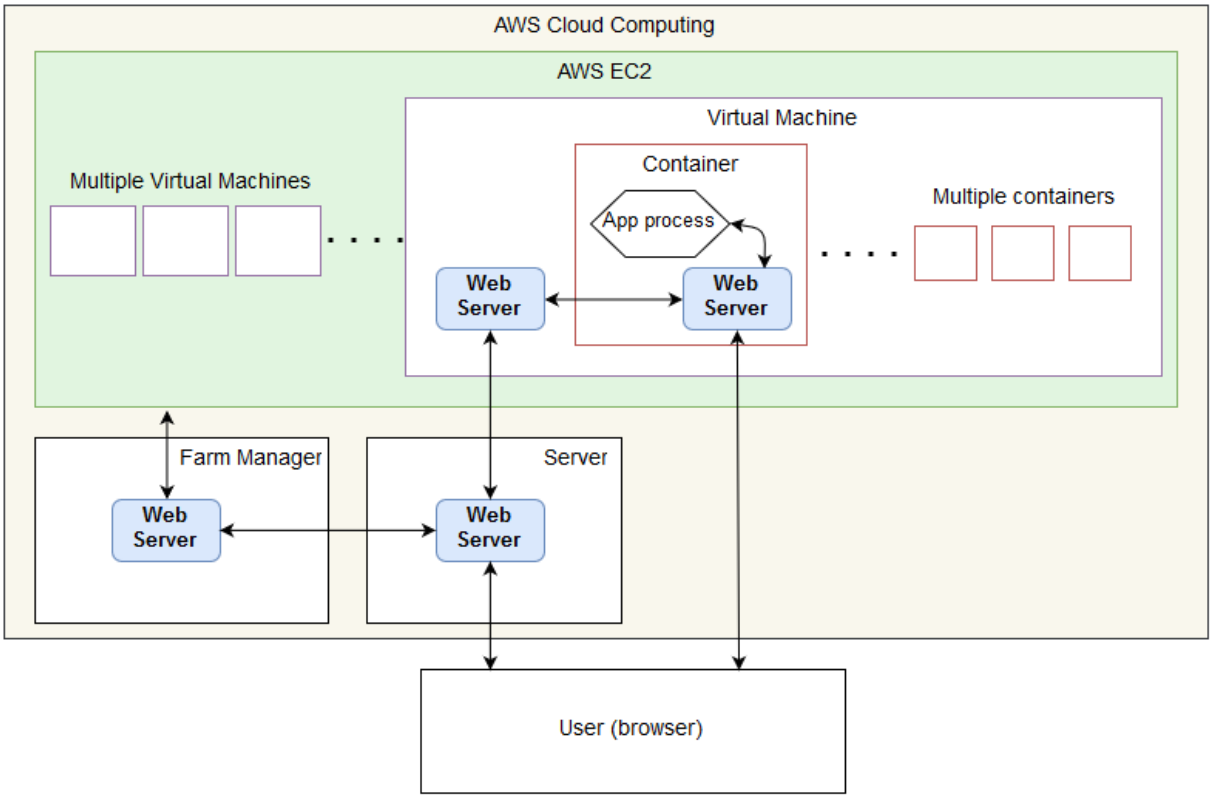

Figure 5.1: Interaction between the different components' web servers.

METHOD	ROUTE	INFO
GET	/apps	Returns the user's list of applications
GET	/apps/*:appID*	Returns the info about the application that corresponds to the *:appID*
POST	/apps	Creates and installs a new application
PATCH	/apps/*:appID*	Modifies a specific field of the *:appID* app. This is the route used to run or stop an application by modifying the status field (either to "running" or "stopped").
DELETE	/apps/*:appID*	Deletes the app that corresponds to the *:appID*

Table 5.1: Server REST API.

5.2 Farm Manager

The managing of virtual machines happens through the farm manager. Whenever the farm administrator wants to start or delete a virtual machine he interacts with the farm manager through its REST API. Then, the farm manager communicates with AWS EC2 [26]. AWS EC2 supports an SDK for javascript which we use through nodeJS. More specifically, the function to start a new VM is ”ec2.runInstances” and to delete a VM is ”ec2.terminateInstances”. These functions return an ID that belongs to the virtual machine. With this ID it is possible to get its ip address and other virtual machine information. Using this, the farm manager keeps a database of all virtual machines.

Centroid features two types of virtual machines, one for applications that need low resources (Low-Quality) and another for applications that need higher computing resources and a GPU (High-Quality). AWS EC2 has several possible virtual machines configurations, the one we chose to represent the LQ VM is called t2.micro and the one we chose to represent the HQ VM is called g2.2xlarge. For this reason, when using the farm manager API to start a new VM, the farm administrator is required to specify its type, Low-Quality (LQ) or High-Quality (HQ). The farm administrator is also required to specify the operating system. Then the farm manager starts the virtual machine according to this information which is then stored on the farm manager database.

The farm manager database will be used for matching an application to a VM. But because virtual machines have different limits for how many applications they can run, this is also stored on the database. It is called occupancy. A LQ virtual machine is able to run 8 Textual applications and an HQ virtual machine is able to run 6 Office or 2 Graphical applications. We needed to find a way for the database queries to check efficiently virtual machines with available spots. We did this by assigning a starting occupancy value of 8 for a LQ VM and 6 for a HQ VM. Whenever an application is executed on a virtual machine, the occupancy value is subtracted by the ”weight” of the application. This ”weight” is 1 for Textual applications, 1 for Office applications and 3 for Graphical applications. For example, if a Graphical application is executed on an empty HQ VM, its occupancy goes from 6 to 3. This way the database knows that in that VM only fits one more Graphical application or 3 Office applications. Once the occupancy goes to 0, the virtual machine is full and no more applications can run there. It is a very efficient query because the database only have to check for VM which occupancy is higher than an application ”weight” value and update it all at once. The next section describes what happens after an application is assigned to a virtual machine.

5.3 VMs and Containers

Whenever a user requests to run an application, Centroid checks for an available VM according to its specifications. Then the server sends a request to that VM to run the application. As it was mentioned

before, this interaction is made through the virtual machine web server. After the VM receives the request, it proceeds to run the container. We use Docker to containerize applications and it is installed on the VM. To execute Docker we use a nodeJS function called ”exec” which starts a new process with a given command. For example, the command to run a docker container is ”docker run”. The exec function returns a callback that we can use to respond to the server informing that the application is running. When a user wants to install an application he sends the setup file or command to the Centroid server. Then this is stored on the user file system on AWS S3 (in case of the file) or on a database (in case of the command). This information is stored on the application databased maintained by the server as explained on the design section.

When the server requests the virtual machine to execute that application it checks if the applications is already installed or not. In the case it is not installed, the virtual machine starts a default container. This container only has the web server running and nothing more. Then, the virtual machine sends a request to that container's web server API to install this specific application, sending its information. We adapted the open source solution s3fs to mount Centroid's distributed file system on the virtual machines. Whenever a virtual machine runs a container it mounts a volume which corresponds only to the specific user file system folder. In Docker this is done using the flag ”-v”. Since the setup file for every application is saved in this file system, the container can access it easily. Also all of the user's files are stored here making sure that all applications have access to them.

A problem that came with running multiple containers on one VM is conflicting ports. Because all containers host a web server that use the port 3000, running multiple containers is not possible because they will all try to use the same port. So, to prevent this the virtual machines attribute different ports to different containers. This can be done when executing a docker container with the flag ”-p”. Now, container A is accessed using port 5000, container B using port 5010, container C using port 5020 and so forth. This ports than redirect to the 3000 port of the container's web server. This way we ensure that we only have to program the containers' web servers once and just redirect ports. It is much easier to manage as well. The next section describes the implementation for accessing and using an application after its container is executed.

5.4 Application Usage and Remote Display

Every container has a web server running. In the case of a Textual application, containers' web servers host a web application which is an emulated terminal. So, whenever the user accesses the container web server, a terminal web app appears on his browser. Also, the application is executed by the container and the output and input is redirected to the terminal web app. This terminal is a port of the chromium javascript terminal that we were able to adapt and use in Centroid. According to the design

section we have to use a pseudo-terminal to run the applications. On Linux we use a nodeJS library called pty.js and on Windows we use ptyw.js. This ensures that the application acknowledges it is being executed by a terminal. It is also easier to attach to its output and input. Once any output comes from the application, it is redirected to the browser via websockets and written on the terminal web app using write and read functions provided by the chromium javascript terminal. The opposite happens when the user inputs something on the browser terminal, in this case the pty.js and ptyw.js also provide write and read functions.

For Graphical and Office applications, containers' web servers host a web application for the remote display. We built almost from the ground up a new remote display software capable of streaming high-end applications to the browser. The HQ virtual machines provided a Nvidia GRID GPU. This GPU has an SDK that is possible to use to encode frames. Apart from the provided hardware, all the rest of the remote display components were implemented by us. We implemented a c++ program that uses the Nvidia GRID GPU to grab the desktop images and encode them to H264 frames. We were able to make a function in this program to return a single frame. And then we adapted it so that it could be called from node JS. This was accomplished using node-gyp addon, which is a way to use c++ inside nodeJS. Now we are able to grab and encode H264 frames from the nodeJS web server in a controllable manner. We used websockets to connect the browser to this web server which we used to send the frames. The challenge then, was to play those frames on the browser because it does not support raw H264 out of the box. Browsers only support H264 if it is wrapped with MP4. We tried to wrap the H264 on a MP4 container on the web server before sending it to the browser and although it worked, it slew down the stream. It would be fine only for non real time use cases but it does not work for a remote display. The only option was to use a H264 decoder on the browser. We used Broadway JS which is a port of Android's H264 decoder compiled with Emscripten to javascript. The problem now is that Broadway JS plays H264 files but not streams. We had to adapt the frames and the decoder so that we were able to play H264 streams. Another option was to use flash but we wanted to be plugin free so it would be compatible with all devices including mobile phones. We also used websockets to send the keyboard and mouse information from the browser to the web server. This ran on a different port to not interfere with the remote display stream. On the application side, we used a java library called robot to emulate the mouse and the keyboard. The java functions were called via nodeJS using another addon called node-java that, similar to the c++ addon, enabled us to call java code from nodeJS. We used this java robot library because it had the best relation between performance and maturity. We also made sure that the mouse on the server side was not rendered so the user only sees his mouse on the client-side. Whenever he drags the mouse or clicks the position is updated.

Also because every container is customizable, they also provide on a different port, a web application which is a terminal web app. But in this case it does not attach to any application in particular. It is just

a window to the container very similar to ssh. The next section describes the technologies used for application communication.

5.5 Application Communication

The solution for enabling application communication is described on section 4.2. Designing the solution included having to define most of the technology to be used consequently the implementation was follows very closely the design. It was described that every container has a folder with several scripts that are named after the users' applications. This folder is on the PATH variable which can be set both on Windows and Linux. This makes the applications visible so they can be executed. Then, it starts the interoperability process. The scripts are nodeJS scripts that once called, interact with Centroid server API to check where the desired application is running or if it needs to run. After this, the script connects to the web server container of the executed application. This triggers the container to execute the application and stream its output via websocket to the first application. By using a nodeJS function called "spawn" we can execute an application and easily attach to its output and input.

5.6 Summary

This chapter describes how the prototype is actually implemented. First it is described how the different components interact with each other. For this it was implemented web servers with an REST API in every component. To access the different types of applications, it was implement a web application which emulates a browser terminal for Textual applications and a remote display for Graphical and Office applications built almost from the ground up. There were a lot of challenges that were needed to overcome that were described. In the end, it was used Nvidia GRID GPU to accelerate the encoding of the screen images, websockets to send images to the user and a decoding library on the browser to decode the images. Then, it was described how the application communication was implemented.

6

Evaluation

This section evaluates quantitatively the four requirements that were set for Centroid: communication between applications from different operating systems; scalability; high remote display performance; and reduced costs. Centroid's objective is to deliver applications to users, no matter their operating system and with full communication between them. Communication between applications, which was achieved by developing an interoperability component that made possible applications from different operating systems to interact between each other, is tested (on section 6.1). It is evaluated its performance and then compared to the performance of same OS application communications.

However, achieving interoperability may not suffice if users have to endure long waiting times so they can start using an application like it happens with existing remote desktop services. This waiting time is evaluated on section 6.2. Then we do a scalability test to assess multiple user situations.

A major obstacle that comes with remote desktops is the fact that they are "remote". The desktop screen images have to be streamed to the local user device so he can see the applications. This is called the remote display. Having a low-quality remote display can be a huge setback for people because there are applications, like games or video, that are impossible to use if the stream is bellow a certain frame rate level. And if people are restricted from using certain types of applications they tend to just use the local PC for every application. Centroid's remote display was implemented with this in mind and is thoroughly evaluated on section 6.3 including multiple application situations.

Finally, for Centroid to be used on the real world, it has to take into account costs. So one of Centroid's focus was to optimize costs without decreasing quality. Section 6.4 evaluates Centroid's costs and how they would be without any optimizations.

The experiments were performed using AWS EC2 virtual machines which we called Low-Quality (LQ) and High-Quality (HQ) VMs introduced in section 4.1.3. Their specifications are depicted on table 6.1. The virtual machines operating systems were Linux (Ubuntu 14.04 distribution) and Windows Server

2012 and the AWS region was Ireland. We also used a local computer running Windows 10 that has an Intel Core i7-4720HQ Processor (2.60GHz 1600MHz 6MB), 8.0GB PC3L-12800 DDR3L SDRAM 1600 MH and a NVIDIA GeForce GTX 960M 2GB.

VM Type	VM specifications
Low-Quality (LQ)	1-core Intel Xeon Family 2.5 GHz 1GiB RAM
High-Quality (HQ)	GPU NVIDIA GRID K520 with 4GB of video memory. 8-core Intel Xeon E5-2670 3.6GHz. 15 GiB RAM.

Table 6.1: LQ and HQ virtual machines specifications

6.1 Interoperability

One of Centroid's requirements was to enable communications between applications. These applications run on containers so they are isolated from each other, furthermore, they can be running on separated virtual machines and even be running on different operating systems. To accomplish this it was developed an interoperability component that featured websockets providing a channel for applications to interact. This section evaluates communication performance between two applications. The tests are divided in two different environments: 1) both applications running on the same operating system; 2) both applications running on different operating systems. Note that since Windows containers are not yet available, Windows applications ran without them. Experiments also took into account that applications can run on containers on the same and on separated Low-Quality or High-Quality virtual machines. Figure 6.1 illustrates the communication performance between the same operating system, Linux or Windows.

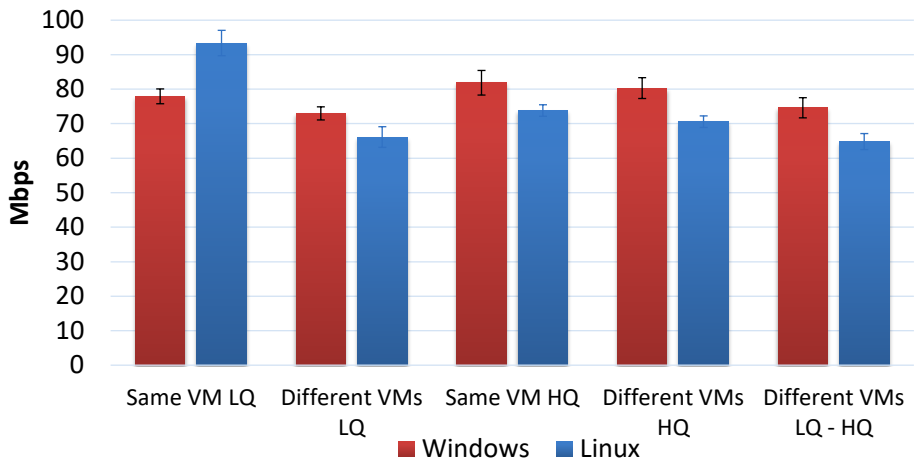

Figure 6.1: Performance of communications between applications on the same operating system.

The fastest communication speed is 93 Mbps and it happens between containers on the same Linux LQ VM and the slowest communication speed is 64 Mbps and it happens between containers residing on different Linux virtual machines, specifically one on a LQ VM and the other on a HQ VM. Communications between Windows applications are very similar in performance to each other for every situation. Even so, communication performance do not differ very much in the two operating systems. If these communications were to happen on a conventional way, by pipes, the speed would be on average 6581Mbps which is several times faster than Centroid communications. So, if applications really need a very high speed for communication, users should install them on the same container. For most cases, even the slowest performance which is 64 Mbps is adequate for the generality of applications. We also performed tests for application communication between different operating systems. Meaning that one application would run on Windows, and the other on Linux. The tests took into account that applications can run on different types of virtual machines (HQ and LQ) on both operating systems. Figure 6.2 illustrates this evaluation.

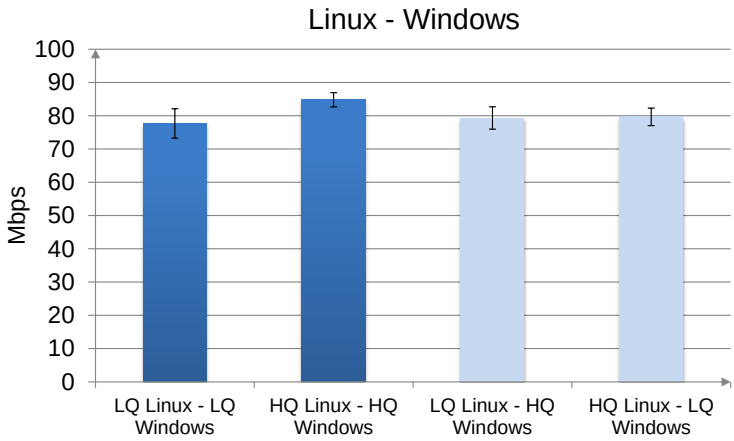

Figure 6.2: Performance of communications between a Linux application and a Windows application.

On average, the communication performance is 80 Mbps for the different cases which is very similar to the communication performance between applications on the same operating system on Centroid. Herein the fact that one application is running on Windows and the other on Linux does not influence negatively the performance. This is a positive result showing that interoperability between different operating systems can be achieved without performance loss.

6.2 System Performance and Scalability

Current remote desktop works use virtual machines to directly run applications. Each user computing environment, like applications and files, reside inside one virtual machine. When a user wants to access this environment, the virtual machine has to be started and then the interaction is done via remote

display. Centroid on the other hand, uses containers to run applications. The user instead of accessing a whole virtual machine environment, accesses individual applications that are containerized. When it comes to performance, the difference between containers and virtual machines is the time it takes to start each one. Having to wait a long time, when comparing to a local pc, for an application to start may alienate users from using remote desktops. To show exactly the difference it would make if Centroid used virtual machines instead of containers to run applications, we ran an experiment with Linux containers of several sizes and Linux virtual machines on AWS EC2 cloud.

Because applications may have several sizes which leads to containers having also several sizes we chose three different sizes for this experiment. They meant to represent the different types of applications that Centroid has defined are: Textual - 100MB; Office - 1GB; Graphical - 5GB. Because at the time of the writing of these thesis, only Linux has container technology available, we did not test containers for Windows. Table 6.2 the time it takes for each container and the time it would take for a full virtual machine to start.

Centroid			
Container 100 MB	**Container 1 GB**	**Container 5 GB**	**Virtual Machine**
0.344 seconds	0.346 seconds	0.349 seconds	51.56 seconds

Table 6.2: Time it takes to start containers vs starting virtual machines

On the background Chapter (section 2.1.2) it is stated that containers are much more lightweight than virtual machines and experiments show that a virtual machine takes almost 1 minute to start while a container, even a 5 GB container, takes less than half a second. The virtual machine depicted on the table is a Low-Quality (LQ) one. We also tested the HQ VM and it takes 82 seconds which is more than a minute. This virtual machine takes more time probably because it has more complex resources like a GPU that the cloud infrastructure needs to attach. We also tested starting a Windows virtual machine, both HQ and LQ and they took on average 4 minutes. We cannot compare to Windows containers because they are not yet available but they will not probably take nearly as long. Even for the 1 minute that the Linux VM takes, this waiting time is much more than it would take to start an application on a local PC. Although it would only have to start it once, because that virtual machine would have all the applications. But if the user wants multiple operating systems then it would also mean multiple virtual machines, one for each OS. On the other hand, containers are much more flexible and fast, almost with an unnoticeable overhead.

Before a container is actually started Centroid has to do several operations. When a user clicks to start an application on the Centroid web app, the server needs to get the application info, then it communicates with the farm manager to check for an available virtual machines, then the virtual machines have to configure the application environment like the user file system and the interoperability component. This is also an overhead that a local PC would not have to endure. Table 6.3 describes the times in

milliseconds of the several steps.

Steps	Values [ms]
1) Get application info	25.7
2) Get available VM from the farm manager	102.9
3) Make user file system accessible	72
4) Make other user applications visible for interoperability	2.3
Total	**202.9**

Table 6.3: Overhead (in milliseconds) introduced by Centroid when executing an application

The total overhead of Centroid is less than 1/3 of a second. It is hardly noticeable. What each step stands for is explained in detail on the design section 4. The last two steps happen inside the virtual machine and virtual machines only handle at maximum 8 applications at a time. On the other hand, the initial two steps happen on the server component and on the farm manager component and these two components take all of the requests without any limit. The most critical one is the farm manager. It takes more time because it has to check for an available VM and also update its occupancy status.

To assess a multi-user situation we did a scalability test. The scalability of the farm manager is very critical, it is one of the components that will take the most load. Our design tried to promote scalability, and this section will evaluate the result. The farm manager is responsible for checking and updating the occupancy of a VM. The occupancy is the number of applications (containers) that are running on a VM. An HQ VM has an occupancy limit of 6 in which a Graphical application would take 3 and Office 1. And a LQ VM has the occupancy limit of 8 Textual applications. We performed a stress test on the farm manager server and database with up to 100 requests at the same time.Figure 6.3 illustrates this evaluation.

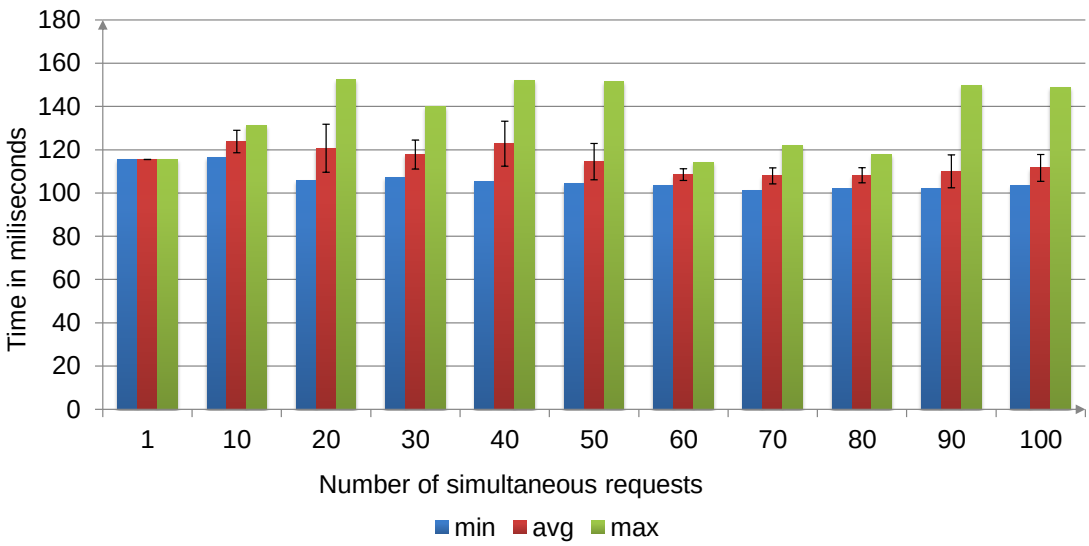

Figure 6.3: Farm manager stress test. Represents the time it takes to respond to simultaneous requests at the same time.

On a real scenario, 100 requests would would happen if 100 applications were being executed in one second. We defined this number because it depicts a high usage situation of the system. Observations show that the response time does not increase with the number of simultaneous requests. Herein we can conclude that it is scalable.

6.3 Remote Display

The remote display is implemented in three parts: encoder; stream; decoder. The encoding is done on the VM using the Nvidia GRID SDK. It grabs the desktop image and encodes it to a H264 frame. Then, the frames are transmitted to the client browser. This is done with WebSockets via nodeJS. But, the transmission of frames are dependent on the encoding, because a remote display only transmits a frame after it is encoded. It does not make sense to send the same frame over and over, so it has to wait for a new frame to be encoded. That is why the encoding is included on the streaming and have to be evaluated together. To be easier we will use the word ”streaming” but keep in mind that this is the combination of ”encode + transmission” of the frames, in another words, it is the time it takes for a frame to reach the user's device. The browser then decodes the frames and displays them. The final frame rate value corresponds to the slowest operation between streaming and decoding.

The quality of the remote display can be related to how many frames are displayed to the user in a certain amount of time, this is called frame rate. This is usually measured in frames per second (FPS). More frames per second means higher quality. Section 3.2 describes what are considered bad or good values for a frame rate. To summarize these values and also to define comparable conditions for this evaluation, Table 6.4 presents a list of such values. We will take these values into account to assess Centroid remote display performance.

Frames per second (FPS)	**Quality**
24 or less	Very poor
between 24 and 30	Poor
between 30 and 60	Good
60 or more	Excellent

Table 6.4: Quality of the remote display in function of the frame rate.

We chose three types of applications to evaluate the performance of Centroid remote display: 1) LibreOffice Writer that represents a low GPU usage applications with mainly text and static images; 2) Firefox with a youtube clip[1] running in fullscreen that represents an application with dynamic images (video). A definition of 4K was choosen for the youtube clip; 3) A game that represents a GPU intensive application. The game is Heaven Benchmark produced by Unigine which simulates a real game. The

[1] https://www.youtube.com/watch?v=iNJdPyoqt8U

definition chosen were ”ultra” so we could simulate a very high-end game. The resolution for all applications is 1280x720. Figure 6.4 illustrates the three applications according to the maximum speed of three different remote display operations: encoding; streaming; decoding.

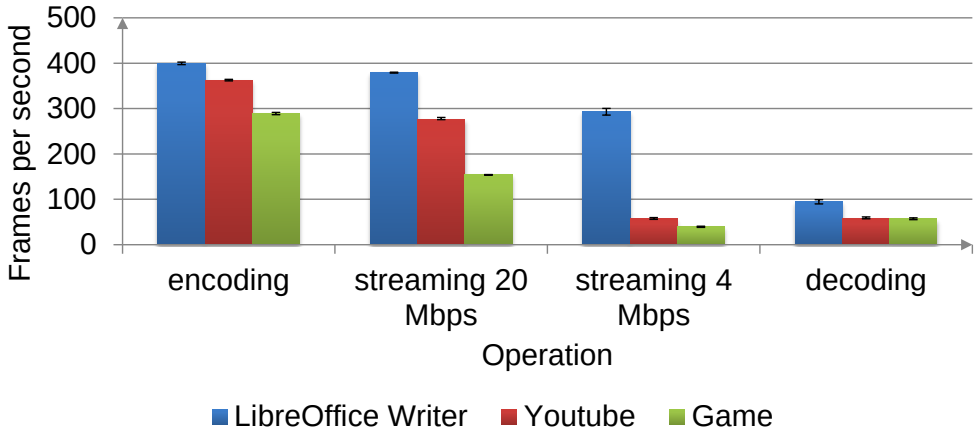

Figure 6.4: Encoding, streaming and decoding maximum performance for three different types of applications.

The fastest operation is encoding the images as it is done with help of the Nvidia GRID GPU hardware. It is able to encode the desktop images of LibreOffice Writer at 400 FPS, Youtube clip at 360 FPS and the game at 290 FPS. As expected, the LibreOffice Writer application is quite static so it is very fast to encode, on the other hand, the game is much more dynamic and GPU intensive so it is slower to encode. The decoding is done on the client device and, like the encoding operation, the more graphically intensive the application is, the slower it is to decode. This decoding is made on the browser like described on section 5.4. Instead of the browser we could have implemented a software that the user would install. We chose the browser for its cross-platform nature and also convenience because the user do not have to install anything. The challenge is that JavaScript, which is the browser script language, is inherently slower than, for example c or java which would be used on the software. Nonetheless, we were able to decode at at least 60 FPS for the game and as high as 90 FPS for LibreOffice. So, the quality is excellent according to Table 6.4 even using the browser.

The most important part of the remote display is the streaming because it is usually the bottleneck. And this is the part that works try to improve and is the most evaluated. For OpenOffice Writer, Centroid is able to stream at 380 FPS with a bitrate of 20Mbps and at 293 FPS with 4 Mbps. This is several times greater than the threshold for an excellent quality which is 60 FPS. But this is only an office application, the challenge is greater for graphically intensive applications. Centroid is able to stream the Youtube clip at 276 FPS with a bitrate of 20Mbps and 57 FPS with a bitrate of 4 Mbps. Even with a low bitrate of 4Mbps, Centroid is able to stream at almost 60 FPS a full screen Youtube clip providing an excellent quality even for a graphically dynamic application. For the game, Centroid is able to stream at 153 FPS with a bitrate of 20Mbps and 40 FPS with a bitrate of 4Mbps. Now, we can observe that with a good

internet speed, it is easy to get more than 60 FPS. Fortunately, internet speed keeps getting higher every year so this proves it is possible, even with 20Mbps which is not that high, to have an excellent remote display quality. But, even with a low bitrate such as 4Mbps, Centroid is able to stream at 40 FPS which is not excellent but it is higher than 30 FPS which is also a good stream experience.

Current works, described in section 3.2, were able to stream an average of 23 FPS with a bitrate of 6 Mbps. In Centroid, the worst situation is 40 Fps with 4 Mbps. Herein, Centroid is able to stream at almost double the frame rate needing only 2/3 of the bitrate. This is a major improvement showing again that with the right hardware and software implementation is possible to have a great remote display experience even with low bitrates. But, this is only for one application at a time, we now evaluate the situation where multiple applications are used and streamed to the same user at the same time.

To illustrate scenarios where multiple applications are streamed to the user, we evaluated two applications. These two applications are the two that perform worst on the last section, the Firefox with the youtube clip and the game. This evaluation will also explain why we set the limit of 6 for Office applications and 2 for Graphical applications. Also, these applications are running containerized on the same virtual machine. Figure 6.5 and Figure 6.6 the scenarios described for each application.

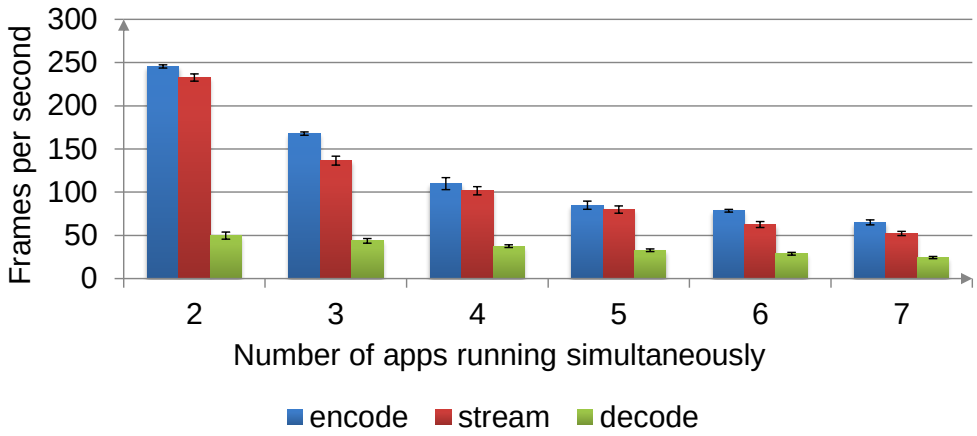

Figure 6.5: Scalability test for up to 7 Firefox Applications running a youtube clip in full screen.

We set the bitrate to 20Mbps and evaluated from 2 to 7 youtube applications running at the same time, the performance is described on the first figure. Centroid is able to encode at 245 FPS, stream at 232 FPS and decode at 50 FPS. At 6 youtube apps, the streaming is at 60 FPS and encoding 78 FPS but the decoding is 30 FPS which is the threshold for a good experience. We can conclude that when we scale, the decoding on the browser becomes the bottleneck, not the streaming. Nonetheless we can run up to 6 video applications on the same VM and decode them on the user browser with a good quality. This means that this type of applications, even being graphically dynamic but not as GPU intensive as a game, can run 6 at the same time. We took this into account to set the limit of Office applications to 6. This ensures that the Office type applications run with good quality always. The game

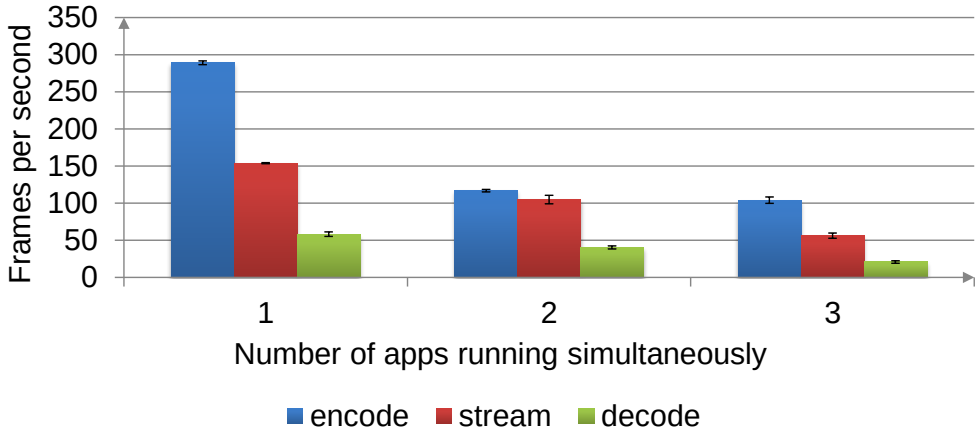

Figure 6.6: Scalability test for up to 3 GPU intensive Gaming Applications.

is different, the decoding really becomes a major challenge for more than 2 running at the same time. When running 2 games is 40 FPS which still qualifies for good experience. But for 3 games it is 20 FPS which is lower than 30 FPS and not recommended for this type of application. This is the reason we set the limit of 2 Graphical applications per VM ensuring great quality even for the most graphically dynamic applications.

6.4 Costs

The previous sections evaluated performance and scalability of several Centroid components. Even though the results were positive, they may not suffice if costs are not competitive with other works or even with just buying a high-end PC. That is why one of the focus and requirement of Centroid was cost optimization. To calculate the costs we assumed the following situation:

- **Number of simultaneous users:** 100.
- **Applications running at the same time per user:**
  - 1 Textual - Linux;
  - 1 Office - Windows;
  - 1 Graphical - Linux.
- **Total of hours using Centroid per month per user:** 6 hours per day * 30 days = 180 hours.
- **Total of storage space used by each user on Centroid:** 500 GB.

This situation simulates a usage of the three types of applications and different operating systems. We assumed 6 hours using Centroid per day per month which is a normal usage. And also 100 users for an initial calculation. We will increase these numbers afterwards.

We are using Amazon Web Services (AWS) to deploy Centroid so we will calculate costs based on their prices. There are basically three types of costs: data transfer; storage space; virtual machine time usage. The easiest to calculate is the storage space costs. We defined 500 GB per user and 100 active users. This amounts to 50 TB (terabytes). We used AWS Simple Monthly Calculator [57] to calculate the costs. 50TB on S3 costs 1510.92 dollars per month. Because Centroid's costs are supported by the users, we divided by 100 to determine the cost per user which is 15.11 dollars per month. The data transfer is the amount of bytes transferred from EC2 to the users. In Centroid this data is the one from the remote displays. More specifically, the frames of the applications that are transferred from the AWS to the users' browsers. We estimated the data usage for the three applications that we defined based on the usage hours and the frame size. We calculated that 100 users would transfer 62.54 TB per month. Using AWS Calculator, the overall cost is 5263.27 dollars per month per 100 users. Again, dividing by 100, the cost is 52.63 dollars per user per month.

Centroid uses two types of virtual machines, the t2.micro (LQ) and the g2.2xlarge (HQ). The price for the t2.micro is 0.014 dollars per hour. For the g2.2xlarge, the price is a lot higher. But there is a method to decrease this cost which is by using Spot Instances. Spot Instances are spare virtual machines that AWS provide on a discounted price. The price fluctuates based on the supply and demand of available EC2 capacity. The way to get this virtual machines is by submitting bids that are higher than that price. The most we spent for g2.2xlarge in Spot Instances were 0.2 dollars per hour. We will use this price for this virtual machine. There are 100 users and each user is running 1 Linux Graphical application, 1 Windows Office applications and 1 Linux Textual application. For the 100 Graphical Linux applications, Centroid needs 50 g2.2xlarge virtual machines because each one runs 2. For the 100 Office Windows applications, Centroid needs 17 g2.2xlarge because each one runs 6. For the Linux Textual application, Centroid needs 13 t2.micro virtual machines because each one runs 8. So in the end, Centroid needs a total of 67 g2.2xlarge and 13 t2.micro virtual machines costing 2431.5 dollars per month. Dividing by 100, the cost is 24.32 dollars per user per month.

Comparing to related works 3.1, which approach do not distinguish between applications types, it would not exist Textual, Office or Graphical, it would all be the same and all would be streamed via remote display. On Centroid, a Textual application does not use remote display so we are able to decrease costs. The cost reduction for this type of applications is about 260 dollars per month per 100 Textual apps. Also, the approach taken by related works (3.1), which consists of providing a single virtual machine for users to run and install applications, are compared to Centroid's container approach. With their approach, 100 users would need 100 g2.2xlarge Linux VM for the Graphical application, 100 g2.2xlarge Windows VM for the Office application because it is two different operating systems, and 0 t2.micro Linux VM because the Textual applications could run on the g2.2xlarge Linux VM. This would amount to 200 g2.2xlarge virtual machines costing 7200 dollars per month. Dividing by 100, it would

cost 72 dollars per user per month. Given this, Table 6.5 describes the total cost per user per month of Centroid and these differences of approach of related works as well as how much is Centroid able to save.

Costs	Centroid	Other works' approach	Savings
Storage	15	15	0%
Data transfer	53	55	4%
Virtual machine	24	72	66%
Total	92$ or 82€	142$ or 127€	35%

Table 6.5: Total costs per user per month (when the total number of users is 100).

In this situation with 100 users, Centroid is able to save 50 dollars per month for each user, this is a 35% cost reduction. In this scenario, the total cost per user for using Centroid would be at least 92$ or 82€. But, prices on AWS decrease if the usage is increased, for example, the data transfer price is 0.09$ per GB for up to 10 TB of transferred data per month. But after 10 TB the price reduces to 0.085$ per GB. After 350 TB of transferred data per month the price reaches 0.05$ per GB. That is almost half the initial price. This means that the more users Centroid has, the less costs it has per user. Given this, Figure 6.7 illustrates this cost reduction from 100 to 1 000 000 (one million) users.

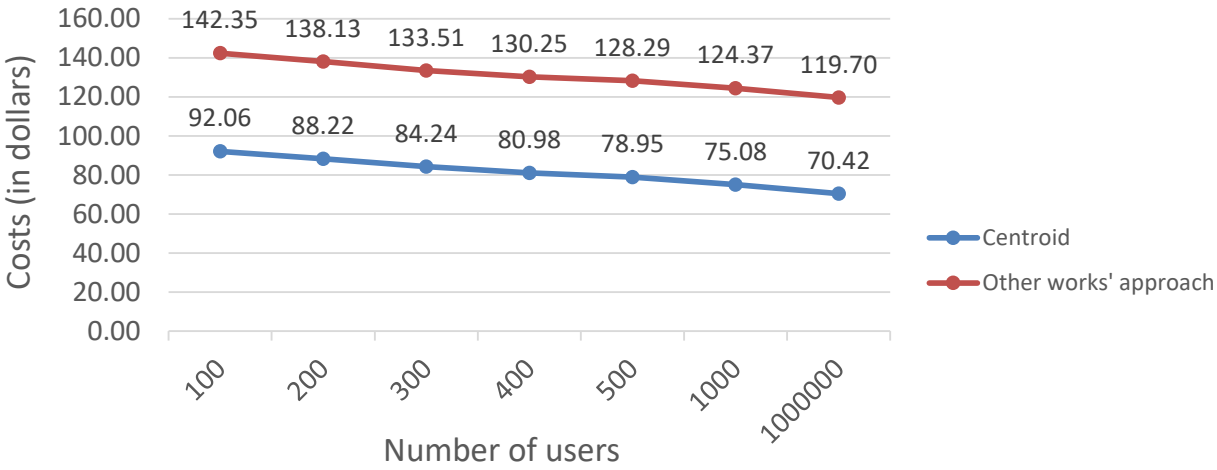

Figure 6.7: Costs reduction when number of users increase.

When the number of users increase, the total cost of using Centroid for each user per month can be as low as 70 dollars or 62 euros. And also, Centroid is able to save more than 40% with our cost optimizations. We also calculated the costs for an 8 hour daily usage instead of 6 hours and the cost per month per user would be 78 dollars or 70 euros, saving more than 45%. To compare with buying hardware or a computer with a GPU, CPU and RAM that could compete with Centroid's virtual machines specifications would cost more than 2000 euros not counting electricity costs. Centroid would cost 744 euros for a 6 hour daily usage or 840 euros for a 8 hour daily usage per year and provide an always available desktop over internet and multiple operating systems with interoperability support.

6.5 Summary

This chapter evaluates quantitatively Centroid's requirements including application communication, performance and scalability of the system, the remote display and costs. Experiments show that the application communication on different operating systems is not degraded compared to communications on the same operating system. Executing an application is also much faster by using containers than using virtual machines and the overall overhead of Centroid is almost unnoticeable. Even when taking into account multiple users, the system is scalable. Remote display performance was very positive achieving double the frame rate using just 2/3 the bitrate comparing to related works. It was also evaluated the costs of using Centroid and results show that Centroid is able to reduce costs by almost 40% and it is a competitive solution to buying a high-quality computer.

7

Conclusions and Future Work

A way to augment and to take further advantage of the emerging cloud computing phenomenon, where people can access their data from any device over the Internet, is with the use of remote desktops. Remote desktops provide virtualized operating systems, where people can run applications and files, that can be accessed anywhere and anytime over the Internet. Some works even provide multiple operating systems expanding the range of applications that the user can run. However, current remote desktop works still incur several problems. Their overhead produces long waiting times, they are not capable of efficiently streaming graphically demanding applications and when applications run on different operating systems, they cannot communicate with each other.

In this thesis, we described the design and implementation of a system, named Centroid, that provides remote desktops where it is possible to run applications from multiple operating systems. It does this by running applications on containers distributed among available virtual machines and streaming them via a novel remote display to a browser interface. By developing an interoperability component, it enabled communications between applications that were running separated or on different operating systems.

The experimental evaluation of the prototype shows that it is possible to reduce, to a nearly unnoticeable value, the overhead of the system by using containers instead of directly running applications on virtual machines. Also, with a careful implementation of components such as the farm manager, the system demonstrated to be scalable. A remote display, built almost from the ground up, proved that a combination of good hardware with good software, both using modern technologies could stream applications with excellent quality. More specifically, it performed twice as good needing only 2/3 the bitrate comparing to related works. By defining different types of applications such as Textual, Office and Graphical, and defining different types of virtual machines namely HQ and LQ, it was possible to allocate exclusively resources that are needed. Also, the fact that we were running applications containerized

allowed for reducing the number of virtual machines needed to be available. These two factors resulted in saving more than 40% in costs.

A good feature of Centroid is that, although it uses Windows and Linux to run its applications, they are abstracted from the users. In the end, users do not even know nor see which OS is running beneath the applications, they just work. A natural direction in the future is to add support for more operating systems. This would give users a way to run their favourite applications no matter their operating system. In other words, people would not have to worry about operating systems anymore, just run applications. Centroid was built for this to be done easily, for example, the new operating system would only have to implement the operations of the VM REST API and it would work right away. Secondly, in the future it would be beneficial to implement a full integration with the system proposed on our paper, named Storekeeper. This would add support for more cloud storage services for the distributed file system and increased security. Thirdly, an algorithm could be implemented to handle the complex operations of the farm manager automatically.

Bibliography

[1] W. Wei, Y. Zhang, Y. Lu, P. Gao, and K. Mu, "A vdi system based on cloud stack and active directory," in *2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*, Aug 2015, pp. 151–154.

[2] G. Lai, H. Song, and X. Lin, "A service based lightweight desktop virtualization system," in *2010 International Conference on Service Sciences*, May 2010, pp. 277–282.

[3] M. F. M. Fuzi, R. S. Hamid, and M. A. Ahmad, "Virtual desktop environment on cloud computing platform," in *Control and System Graduate Research Colloquium (ICSGRC), 2014 IEEE 5th*, Aug 2014, pp. 80–84.

[4] Y. Zhang, Z. Song, D. Zhu, Z. Chen, and Y. Sun, "Redar: A remote desktop architecture for the distributed virtual personal computing," in *2006 Fifth International Conference on Grid and Cooperative Computing (GCC'06)*, Oct 2006, pp. 1–8.

[5] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: A survey on concepts, taxonomy and associated security issues," in *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, April 2010, pp. 222–226.

[6] A. Desai, R. Oza, P. Sharma, and B. Patel, "Hypervisor: A survey on concepts and taxonomy," *International Journal of Innovative Technology and Exploring Engineering (IJITEE) ISSN*, pp. 2278–3075, 2013.

[7] VMware, "Vmware," http://www.vmware.com/, 2016, [Online; accessed 01-Sep-2016].

[8] Xen, "Xen," http://www.xenproject.org/, 2016, [Online; accessed 01-Sep-2016].

[9] Oracle, "Virtual box," https://www.virtualbox.org/, 2016, [Online; accessed 01-Sep-2016].

[10] Parallels, "Parallels," http://www.parallels.com/, year = 2016, note = "[Online; accessed 01-Sep-2016]".

[11] X. Zhao, K. Borders, and A. Prakash, “Virtual machine security systems,” *Advances in Computer Science and Engineering*, pp. 339–365, 2009.

[12] R. Morabito, J. Kjallman, and M. Komu, “Hypervisors vs. lightweight virtualization: A performance comparison,” in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, March 2015, pp. 386–393.

[13] M. Xavier, M. Neves, F. Rossi, T. Ferreto, T. Lange, and C. De Rose, “Performance evaluation of container-based virtualization for high performance computing environments,” in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, Feb 2013, pp. 233–240.

[14] Docker, “Docker,” https://www.docker.com/, 2016, [Online; accessed 01-Sep-2016].

[15] C. Anderson, “Docker [software engineering],” *Software, IEEE*, vol. 32, no. 3, pp. 102–c3, May 2015.

[16] R. Dua, A. Raja, and D. Kakadia, “Virtualization vs containerization to support paas,” in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, March 2014, pp. 610–614.

[17] A. Gajbhiye and K. Shrivastva, “Cloud computing: Need, enabling technology, architecture, advantages and challenges,” in *Confluence The Next Generation Information Technology Summit (Confluence), 2014 5th International Conference* -, Sept 2014, pp. 1–7.

[18] B. Rimal, E. Choi, and I. Lumb, “A taxonomy and survey of cloud computing systems,” in *INC, IMS and IDC, 2009. NCM ’09. Fifth International Joint Conference on*, Aug 2009, pp. 44–51.

[19] Amazon, “Amazon web services (aws),” https://aws.amazon.com/, 2016, [Online; accessed 01-Sep-2016].

[20] Google, “Google cloud platform,” https://cloud.google.com/, 2016, [Online; accessed 01-Sep-2016].

[21] Microsoft, “Microsoft azure,” https://azure.microsoft.com/, 2016, [Online; accessed 01-Sep-2016].

[22] M. Armbrust, O. Fox, R. Griffith, A. D. Joseph, Y. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, “Above the clouds: a berkeley view of cloud computing,” 2009.

[23] F. Sabahi, “Cloud computing security threats and responses,” in *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, May 2011, pp. 245–249.

[24] Y. Jadeja and K. Modi, “Cloud computing - concepts, architecture and challenges,” in *Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference on*, March 2012, pp. 877–880.

[25] Amazon, “Amazon ec2 service level agreement,” https://aws.amazon.com/ec2/sla/, 2016, [Online; accessed 01-Sep-2016].

[26] AWS, “Amazon web services ec2 (elastic compute cloud),” https://aws.amazon.com/ec2/, 2016, [Online; accessed 30-Sep-2016].

[27] C. Hoefer, “Taxonomy of cloud computing services.”

[28] D. Shimada, M. Hashima, and Y. Sato, “Image compression for remote desktop for engineering cloud,” in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, March 2014, pp. 478–483.

[29] R. Komatsu and S.-I. Kuribayashi, “Impact of network quality deterioration on user’s perceived operability in remote desktop environments,” in *Network-Based Information Systems (NBiS), 2014 17th International Conference on*, Sept 2014, pp. 540–544.

[30] J. Rouzaud-Cornabas and N. Viot, “Secured architecture for remote virtual desktops,” in *Collaborative Technologies and Systems, 2007. CTS 2007. International Symposium on*, May 2007, pp. 80–87.

[31] Teradici, “Pc-over-ip,” http://www.teradici.com/pcoip-technology, 2016, [Online; accessed 01-Sep-2016].

[32] HP, “Remote graphics protocol (rgs),” http://www8.hp.com/us/en/campaigns/workstations/remote-graphics-software.html, 2016, [Online; accessed 01-Sep-2016].

[33] RealVnc, “Vnc,” https://www.realvnc.com/, 2016, [Online; accessed 01-Sep-2016].

[34] Microsoft, “Remote desktop protocol (rdp),” https://msdn.microsoft.com/en-us/library/windows/desktop/aa383015%28v=vs.85%29.aspx, 2016, [Online; accessed 01-Sep-2016].

[35] S. Sazawa, M. Hashima, Y. Sato, K. Horio, and K. Matsui, “Rvec: Efficient remote desktop for the engineering cloud,” in *Advanced Information Networking and Applications Workshops (WAINA), 2012 26th International Conference on*, March 2012, pp. 1081–1088.

[36] D. Wang and L. Chen, “Research on virtual desktop management based on cloud computing,” in *Logistics, Informatics and Service Sciences (LISS), 2015 International Conference on*, July 2015, pp. 1–5.

[37] S. Kibe, T. Koyama, and M. Uehara, “The evaluations of desktop as a service in an educational cloud,” in *2012 15th International Conference on Network-Based Information Systems*, Sept 2012, pp. 621–626.

[38] Y. C. Chang, P. H. Tseng, K. T. Chen, and C. L. Lei, “Understanding the performance of thin-client gaming,” in *2011 IEEE International Workshop Technical Committee on Communications Quality and Reliability (CQR)*, May 2011, pp. 1–6.

[39] K. T. Claypool and M. Claypool, “On frame rate and player performance in first person shooter games,” *Multimedia Syst.*, vol. 13, no. 1, pp. 3–17, Sep. 2007. [Online]. Available: http://dx.doi.org/10.1007/s00530-007-0081-1

[40] C.-Y. Huang, C.-H. Hsu, Y.-C. Chang, and K.-T. Chen, “Gaminganywhere: An open cloud gaming system,” in *Proceedings of the 4th ACM Multimedia Systems Conference*, ser. MMSys ’13. New York, NY, USA: ACM, 2013, pp. 36–47. [Online]. Available: http://doi.acm.org/10.1145/2483977.2483981

[41] X. Liao, L. Lin, G. Tan, H. Jin, X. Yang, W. Zhang, and B. Li, “Liverender: A cloud gaming system based on compressed graphics streaming,” *IEEE/ACM Transactions on Networking*, vol. 24, no. 4, pp. 2128–2139, Aug 2016.

[42] Onlive, “Onlive,” http://onlive.com/, 2016, [Online; accessed 28-Sep-2016].

[43] M. Claypool, D. Finkel, A. Grant, and M. Solano, “Thin to win? network performance analysis of the onlive thin client game system,” in *Network and Systems Support for Games (NetGames), 2012 11th Annual Workshop on*, Nov 2012, pp. 1–6.

[44] StreamMyGame, “StreamMyGame,” http://streammygame.com/smg/index.php, 2016, [Online; accessed 28-Sep-2016].

[45] K. T. Chen, Y. C. Chang, H. J. Hsu, D. Y. Chen, C. Y. Huang, and C. H. Hsu, “On the quality of service of cloud gaming systems,” *IEEE Transactions on Multimedia*, vol. 16, no. 2, pp. 480–495, Feb 2014.

[46] CloudUnion, “CloudUnion,” http://www.cloudunion.cn/, 2016, [Online; accessed 28-Sep-2016].

[47] Z. Xue, D. Wu, J. He, X. Hei, and Y. Liu, “Playing high-end video games in the cloud: A measurement study,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, no. 12, pp. 2013–2025, Dec 2015.

[48] Y. Bin, T. Yang, and Y. Haihua, “Research on interoperability testing between operating systems,” in *Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference on*, July 2011, pp. 39–42.

[49] X. Peng, L. Yuanyuan, and D. Wu, “A model of distributed interprocess communication system,” in *Knowledge Discovery and Data Mining, 2009. WKDD 2009. Second International Workshop on*, Jan 2009, pp. 276–279.

[50] M. Sharifi, S. Kafaie, and O. Kashefi, “A survey and taxonomy of cyber foraging of mobile devices,” *IEEE Communications Surveys Tutorials*, vol. 14, no. 4, pp. 1232–1243, Fourth 2012.

[51] M. D. Kristensen and N. O. Bouvin, “Developing cyber foraging applications for portable devices,” in *Portable Information Devices, 2008 and the 2008 7th IEEE Conference on Polymers and Adhesives in Microelectronics and Photonics. PORTABLE-POLYTRONIC 2008. 2nd IEEE International Interdisciplinary Conference on*, Aug 2008, pp. 1–6.

[52] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, “Clonecloud: elastic execution between mobile device and cloud,” in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 301–314.

[53] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “Maui: making smartphones last longer with code offload,” in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62.

[54] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, “Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading,” in *INFOCOM, 2012 Proceedings IEEE*, March 2012, pp. 945–953.

[55] S. Kim, H. Rim, and H. Han, “Distributed execution for resource-constrained mobile consumer devices,” *Consumer Electronics, IEEE Transactions on*, vol. 55, no. 2, pp. 376–384, May 2009.

[56] s3fs, “s3fs,” https://github.com/s3fs-fuse/s3fs-fuse, 2016, [Online; accessed 30-Sep-2016].

[57] Amazon, “Aws simply monthly calculator,” https://calculator.s3.amazonaws.com/index.html, 2016, [Online; accessed 30-Sep-2016].