Centroid: A Remote Desktop with Multi-OS Interoperability Support

André Pimenta Alves
andre.pimenta.alves@tecnico.ulisboa.pt

Supervisors: Nuno Santos & Ricardo Chaves
Instituto Superior Técnico, Lisboa, Portugal

November 2016

Abstract

Remote desktops provide virtualized operating systems, where people can run applications and files, that can be accessed anywhere and anytime over the Internet. 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. Therefore, our goal is to propose a solution that overcomes these issues. We propose and implement a system named Centroid 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. Results show that Centroid is scalable, application communication is efficient, the remote display performs twice as good as related works needing only 2/3 the bitrate, and is able to reduce costs by over 40%.
Keywords: Remote Desktop, Cloud Computing, Multi-OS, Interoperability, Remote Display

1. Introduction

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. Following this trend, there are remote desktops. Remote desktops allow people not only to access their files, but also their applications anytime and anywhere 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. [17], 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 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 M. Fuzi et al. [8], support remote desktops with multiple operating systems. This extends the range of applications that users can run.

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. 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, consequently, users may have to choose only one operating system if they need applications to communicate.

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)*;

R2: *Scalable*;

R3: *Remote display performance suitable for any type of application*;

R4: *Lower cost when compared to other related works approaches*;

This paper 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. It also produced a functional prototype of Centroid deployed on the Amazon Web Services (AWS) cloud. 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%.

2. Related Work

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 operate by sending the screen images of the applications to the browser and receiving back the users' mouse and keyboard operations. The practice of hosting, maintaining and managing desktop operating systems running within virtual machines on the cloud is called a Virtual Desktop Infrastructure (VDI). A number of works have proposed such systems.

D. Wang et al. [16] propose a VDI that which VMs only use computing resources as they are needed achieving greater flexibility which leads to a more efficient consumption and decreased costs. They support only Windows. W. Wei et al. [17] also supports Windows and also provide a web-based interface to be used by both end-users and system administrators

M. Fuzi et al. [8] introduce a virtual desktop environment for laboratories at schools and universities and supports multiple operating systems, namely Windows and Linux. S. Kibe et al. [10] propose a system also delivering remote desktops featuring multiple operating systems namely CentOS, Ubuntu and Windows. They conclude that scaling the infrastructure to more users can reduce costs, but note that performance degrades.

The state of the art presented above focus on developing an efficient way of delivering remote desktops. However, they are only capable of streaming office-like applications with static graphics. To present related works that focus on streaming high-end graphical applications we decided to analyse cloud gaming services. Their objective is to maximize the quality of the game remote display (streaming). The quality 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). 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 [4]. So, it is on their interest to build a system that maximizes streaming quality.

GamingAnywhere [9] is an open-source gaming system based on the cloud. It adopts the H.264 standard to encode applications frames via CPU. Tests [13] show that GamingAnywhere achieves on average 22 FPS with a bitrate of around 8 Mbps. Onlive [14] was a commercial cloud gaming service that closed in April 2015. It achieved 25 fps with a 5 Mbps bitrate as stated on [5]. StreamMyGame [15] is a software made for game streaming that enables users to play Windows games remotely. Tests [2] showed that it can achieve 25 fps with a 6 Mbps bitrate. CloudUnion [6] supports streaming to web browsers so users do not have to install any software Tests show that CloudUnion achieves 20 fps with a 6 Mbps bitrate as measured on [18].

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 (section 9.3).

For related works that focus on communications between processes that run separated from each other we analysed the cyber foraging field. With cyber foraging, applications are partitioned into a part that runs locally and another that runs remotely. Than the two must communicate. Their objective is to take advantage of stronger remote servers to enhance the capabilities of the local device. Locusts [12], DiET [11], MAUI [7] and CloneCloud [3] are some of these works. Generally RPC is used for communications but all of them need applications to be modified in order for communications to happen.

3. Design

Our objective is to deliver a remote desktop to users where they can run applications from multiple operating systems. To better understand its expected usage model, we proceed to demonstrate how a user would open applications. 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 via remote display. 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.

3.1. 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 1 illustrates an overview of the system architecture and its components. Afterwards they are described in detail.

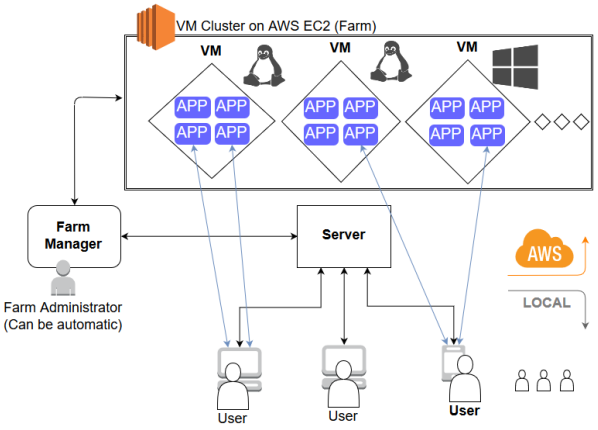

Figure 1: Overview of Centroid's architecture

a) Applications - 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. 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, 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. By containerizing applications, we can run multiple applications of different users on the same VM in isolation. The applications illustrated on the architecture figure 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 Server versions provide container support. Regarding security, containers are fully self-contained in terms of required application code and implement individual sandboxes for applications, therefore addressing the limitations of standard OSes. Thus, we adopt containers in Centroid.

b) 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 1 on the VM cluster. Costs 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, applications can be very graphically demanding 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: 1) Graphical; 2) Office; 3) Textual. The Graphical type refers to graphically demanding applications that are sensitive to latency. The Office type refers to any application that uses mostly text and graphics that require low GPU attention. The Textual type refers to console applications, it includes any application that runs on a terminal (shell) and does not use a GUI. 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), in particular, the LQ VM corresponds to the t2.micro AWS VM, and the HQ VM corresponds to the g2.2xlarge AWS VM. We also defined that, High-Quality VMs can support as much as 6 Office application instances or 2 Graphical applications running simultaneously. We also defined Low-Quality VMs can support 8 Textual

applications.

c) Farm manager The farm manager is a component made to facilitate cluster operations and is illustrated on the architecture figure (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. In this version of Centroid, the farm manager administration is done manually.

d) Server The server (component in the middle of the architecture figure 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. To make sure that the server is scalable, we use tokens to save user sessions. 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.

e) 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. Furthermore, 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.

3.2. Interoperability

With the choice of having containerized applications running on multiple operating systems and possibly on different virtual machines effectively breaks every communications between them. 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; 3) Direct communication between two applications happens when one explicitly executes the other; 4) Each application runs in its own container.

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 do execute it. To accomplish this, there are several steps that we need to overcome:

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

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

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

4. Send data between applications efficiently, in terms of performance and scalability.

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. So, for example, 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 it. 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 and 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. Figure 2 illustrates this process.

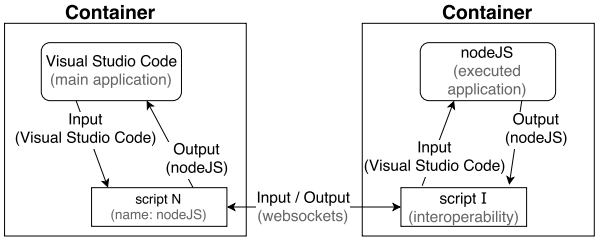

Figure 2: Example of communication between two applications.

In the example of the figure 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.

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. Implementation

This section presents the implementation of the various Centroid's components described on the design. Each component has its responsibility. To accomplish a task they need to cooperate with each other. To accomplish this we implemented a web server on every component. Figure 3 illustrates the different web servers and how they interact.

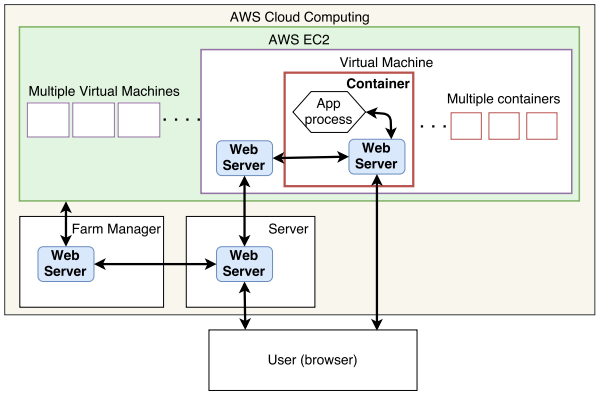

Figure 3: Interaction between the different components' web servers.

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 1 exemplifies the rest of the server API related to applications. All of the components' webservers provide REST APIs related to their responsibility. This webservers were implemented with nodeJS. The next section describes the implementation of the farm manager component.

5. Farm Manager

The managing of virtual machines happens through the farm manager. Whenever the farm administrator wants to spawn or delete a virtual machine he interacts with the farm manager through

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 appliation by modifying the status field (either to "running" or "stopped").
DELETE	/apps/*:appID*	Deletes the app that corresponds to the *:appID*

Table 1: Server REST API.

its REST API. Then, the farm manager communicates with AWS EC2 [1]. 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. This 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.

6. 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 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 uses that container's web server API to send a request to install this specific application sending its information.

7. 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 similar to a terminal. So, whenever the user accesses the container web server, an emulated terminal 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 web app 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 browser terminal using write and read functions provided by the chromium terminal port. 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. We used the Nvidia GRID GPU residing in the HQ virtual machines to grab the desktop image and encode it to H264. We implemented a c++ program to do this. This program returns 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. 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 highest performance and was the most mature. 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.

8. Application Communication

The solution for enabling application communication is described on section 3.2. Designing the solution included having to define most of the technology to be used consequently the implementation follows it closely. It was described that every container has a folder with several scripts that are named after the user's applications. This folder is on the PATH variable. 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.

9. Results

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.

The experiments were performed using AWS EC2 virtual machines which we called Low-Quality (LQ) and High-Quality (HQ) VMs. Their specifications are depicted on table 2. 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 2: LQ and HQ virtual machines specifications

9.1. Interoperability

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 4 illustrates the communication performance between the same operating system, Linux or Windows.

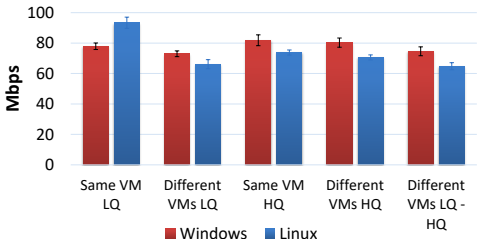

Figure 4: Performance of communications between on the same operating system.

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 5 illustrates this evaluation.

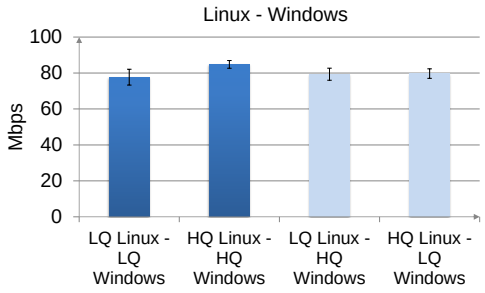

Figure 5: Performance of communications a Linux application and a Windows application.

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.

9.2. System Performance and Scalability

Current remote desktop works use virtual machines to directly run applications. When a user wants to access this environment, the virtual machine has to be started. Centroid on the other hand, uses containers to run applications. 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 at the time of the writing of these thesis, only Linux has container technology available, we did not test containers for Windows. Table 3 the time it takes for each container and the time it would take for a full virtual machine to start.

Centroid Containers			
100 MB	**1 GB**	**5 GB**	**Virtual Machine**
0.344 s	0.346 s	0.349 s	51.56 s

Table 3: Time it takes (in seconds) to start containers vs starting virtual machines

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.

Before a container is actually started Centroid has to do several operations. The most critical one is the farm manager, 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. Figure 6 illustrates this evaluation. Observations show that the response time does not increase with the number of simultaneous requests. Herein we can conclude that it is scalable.

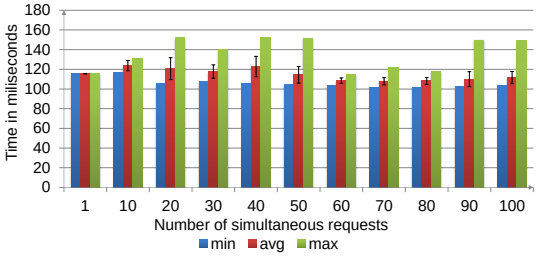

Figure 6: Farm manager stress test. Represents the time it takes to respond to simultaneous requests at the same time.

9.3. Remote Display

The remote display is implemented in three parts: encoder; stream; decoder. 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. A frame rate of 60 FPS is excellent while a frame rate bellow 30 FPS is unusable for most applications.

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 definition chosen were "ultra" so we could simulate a very high-end game. The resolution for all applications is 1280x720. Figure 7 illustrates the three applications according to the maximum speed of three different remote display operations: encoding; streaming; decoding.

The fastest operation is encoding the images as it is done with the help of the Nvidia GRID GPU. The game is the slowest to be encoded because it is much more dynamic and GPU intensive. The decoding is done on the client device and, like the encoding operation, the more graphically intensive

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

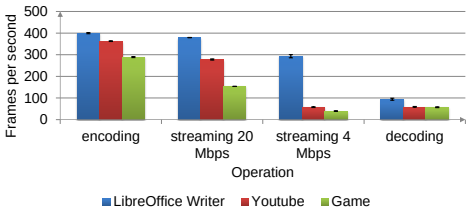

Figure 7: Encoding, streaming and decoding maximum performance for three different types of applications.

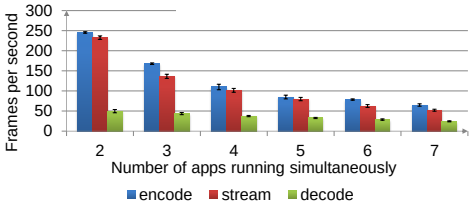

Figure 8: Scalability test for up to 7 Firefox Applications running a youtube clip in full screen.

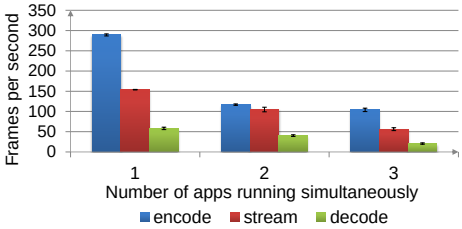

Figure 9: Scalability test for up to 3 GPU intensive Gaming Applications.

the application is, the slower it is to decode. This decoding is made on the browser. The challenge is that JavaScript, which is the browser script language, is inherently slower than, for example c or java. Nonetheless, we were able to decode a minimum of 60 FPS (which corresponds to the game). Given this value, the quality is defined as excellent.

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. The encoding is included in the streaming. 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. So keep in mind that streaming stands for "encoding + transmission" of the frames. The slowest application are the youtube clip and the game. 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. 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 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.

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. These applications are running containerized on the same virtual machine. Figure 8 and Figure 9 the scenarios described for each application.

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. 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. 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 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 demanding applications.

9.4. 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.

We used the situation described above to calculate Centroid's costs. We are using Amazon Web Services (AWS) to deploy Centroid so we will calculate costs based on its prices. We also calculated the costs if we would have taken the approaches of related works such as: providing directly virtual

machines to users so they can run applications; no discrimination of applications' types meaning that all applications are streamed even if they run on a terminal that we defined as Textual. Table 9.4 describes these total costs per user per month.

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 4: 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 10 illustrates this cost reduction from 100 to 1 000 000 (one million) users.

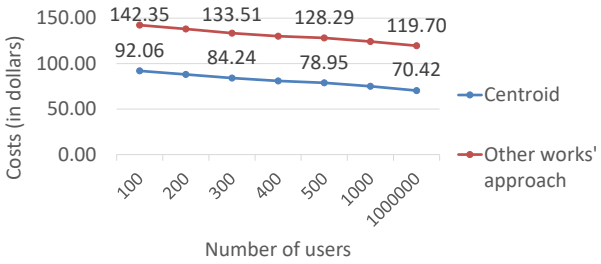

Figure 10: Costs reduction when number of users increase.

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.

10. Conclusions

This work describes Centroid, a remote desktop system with a focus on scalability, remote display quality and costs. It also features a novel interoperability component that enables communications between apps running on different OSes. An evaluation of the prototype shows that application communication is efficient, streams at almost double the frame rate as related works needing only 2/3 of the bitrate, and is able to reduce costs by over 40%.

References

[1] A. (aws). Amazon ec2 (elastic compute cloud). `https://aws.amazon.com/ec2/`, 2016. [Online; accessed 30-Sep-2016].

[2] 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*, 16(2):480–495, Feb 2014.

[3] 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*, pages 301–314. ACM, 2011.

[4] K. T. Claypool and M. Claypool. On frame rate and player performance in first person shooter games. *Multimedia Syst.*, 13(1):3–17, Sept. 2007.

[5] 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*, pages 1–6, Nov 2012.

[6] CloudUnion. CloudUnion. `http://www.cloudunion.cn/`, 2016. [Online; accessed 28-Sep-2016].

[7] 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*, pages 49–62. ACM, 2010.

[8] 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*, pages 80–84, Aug 2014.

[9] 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*, MMSys '13, pages 36–47, New York, NY, USA, 2013. ACM.

[10] 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*, pages 621–626, Sept 2012.

[11] S. Kim, H. Rim, and H. Han. Distributed execution for resource-constrained mobile consumer devices. *Consumer Electronics, IEEE Transactions on*, 55(2):376–384, May 2009.

[12] 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*, pages 1–6, Aug 2008.

[13] 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*, 24(4):2128–2139, Aug 2016.

[14] Onlive. Onlive. `http://onlive.com/`, 2016. [Online; accessed 28-Sep-2016].

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

[16] 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*, pages 1–5, July 2015.

[17] 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)*, pages 151–154, Aug 2015.

[18] 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*, 25(12):2013–2025, Dec 2015.