Luís Miguel Teixeira da Silva

Bachelor of Science

Replication and Caching Systems for the support of VMs stored in File Systems with Snapshots

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Adviser: Paulo Lopes, Auxiliar Professor,
NOVA University of Lisbon

Co-adviser: Pedro Medeiros, Associate Professor,
NOVA University of Lisbon

Examination Committee

Chairperson: Prof. Carmen Pires Morgado
Raporteur: Prof. Carlos Jorge de Sousa Gonçalves
Member: Prof. Paulo Orlando Reis Afonso Lopes

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

November, 2018

Replication and Caching Systems for the support of VMs stored in File Systems with Snapshots

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Acknowledgements

The work presented in this document would never see the light of day if not for the collaboration of several people to whom I wish to manifest my profound gratitude and recognition.

First I would like to thank my advisors and members of the iCBD Project, Professors Paulo Lopes, Pedro Medeiros and Nuno Preguiça, for there advice, always supporting this work and the countless hours spent trying to find the better course of action especially when I would find myself lost and overwhelmed.

It is necessary to give a special thanks to Engineer Miguel Martins from Reditus S.A, to whom I had the pleasure of work closely for more than a year. Who taught me a lot, not only from his vast knowledge of computer systems, backed by countless years of experience in the IT world and without who this work would not be possible but also for the many conversations we held in the realms of Physics, Economics and History. I will always admire how a person can hold such a large body of knowledge and a passion for sharing it.

I also like to extend my recognition to Dr Henrique Mamede and Engineer Sérgio Rita, also from Reditus S.A., for the opportunity given, constant support and the warm-hearted welcome in my year-long stay with SolidNetworks. And to Luís Anjos from the *Divisão de Infraestruturas Informáticas da FCT NOVA* (Div-I FCT NOVA) for all his help in resolving issues related to FCT NOVA's internal network.

Finally, my very heartfelt gratitude to my family and friend for their unconditional support, for putting up with my grumbles when work problems went home with me, being always there with a kind word and some crazy plan to make me forget work for a couple of hours and enjoy their friendship and time spent together.

I also would like to acknowledge the following institutions for their hosting and financial support: *Departamento de Informática and Faculdade de Ciências e Tecnologia of the Universidade NOVA de Lisboa* (DI-FCT NOVA); the *NOVA Laboratory of Computer Science and Informatics* (NOVA LINCS) in particular the Computer Systems group; *SolidNetworks – Business Consulting, Lda* of the *Reditus S.A. Group*; and the funding provided through the *COMPETE2020 / PORTUGAL2020* program for the *iCBD* project (POCI-01-0247-FEDER-011467).

Abstract

Recently, in a relatively short timeframe, there were fundamental changes in the way computing power is used. Virtualisation technology has changed both the model of a data centre's infrastructure and the way physical computers are now managed. This shift is a consequence of today's fast deployment rate of Virtual Machines (VM) in a high consolidation environment with minimal need for human management.

New approaches to virtualisation techniques are being developed at a surprisingly fast rate, leading to a new exciting and vibrating ecosystem of platforms and services. We see the big industry players tackling problems such as *Desktop Virtualisation* with moderate success, but completely ignoring the computation power already present in their clients' infrastructures and, instead, opting for a costly solution based on powerful new machines. There's still room for improvement in VDI and development of new architectures that take advantage of the computation power available at the user's desk, with a minimum effort on the management side; iCBD is one of these projects.

This thesis focuses on the development of mechanisms for the replication and caching of *VM* images stored in a local filesystem, albeit one with the ability to perform snapshots. In this work, there are some challenges to address: the proposed architecture must be entirely distributed and completely integrated with the already existing client-based Virtual Desktop Infrastructure (VDI) platform; and it must be able to efficiently cope with very large, read-only files, (some of them snapshots) and handle their multiple versions. This work will also explore the challenges and advantages of deploying such a system in a high throughput network, with both high availability and scalability while efficiently supporting a large number of users (and their workstations).

Keywords: VDI, BTRFS, Snapshots, Replication Middleware, Cache Servers.

Resumo

Nos últimos anos tem-se assistido a mudanças fundamentais na forma como a capacidade computacional é utilizada - com o grande aumento da utilização da virtualização, tanto a forma como são geridas as máquinas físicas como os modelos de infraestruturas num centro de dados sofreram grandes alterações. Estas mudanças são o resultado da procura por uma forma de disponibilizar rapidamente *VMs* num ambiente altamente consolidado e com necessidades mínimas de intervenção humana na sua gestão.

Estão a ser desenvolvidas novas abordagens às técnicas de virtualização a um ritmo nunca visto, o que leva à existência de um ecossistema altamente volátil com novas plataformas e serviços a serem criados a todo o momento. É possível apreciar o esforço de grandes empresas da indústria das tecnologias de informação relativamente a problemas como a virtualização de desktops - com algum sucesso, mas ignorando completamente o poder de computação que está presente nos PCs instalados, optando por uma via de custo elevado, baseada em máquinas poderosas. Existe ainda espaço para melhores soluções e para o desenvolvimento de tecnologias que façam uso das capacidades de computação que já se encontrem presentes nas organizações, mantendo a simplicidade da sua configuração.

Esta tese foca-se no desenvolvimento de mecanismos de replicação e *caching* para imagens de máquinas virtuais armazenadas num sistema de ficheiros local que tem a funcionalidade (pouco habitual) de suportar *snapshots*. A arquitectura da solução proposta tem de ser distribuída e integrar-se na solução *client-based VDI* já desenvolvida no projecto iCBD; tem de suportar eficientemente ficheiros com vários GB (alguns deles resultantes da criação de *snapshots*) acedidos em leitura, e manter destes múltiplas versões. A solução desenvolvida tem ainda de oferecer desempenho, alta disponibilidade, e escalabilidade na presença de elevado número de clientes geograficamente distribuídos.

Palavras-chave: VDI, BTRFS, Snapshots, Replication Middleware, Cache Servers.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

API Application Programming Interface.

BIOS Basic Input/Output System.

CLI Command Line Interface.

CoW Copy-on-Write.

CPU Central Processing Unit.

DaaS Desktop as a Service.

DC Data Centre.

DHCP Dynamic Host Configuration Protocol.

FAT File Allocation Table.

FS File System.

HTTP Hypertext Transfer Protocol.

IaaS Infrastructure as a Service.

iCBD Infrastructure for Client-Based Desktop.

iMI iCBD Machine Image.

IP Internet Protocol.

iSCSI Internet Small Computer Systems Interface.

IT Information Technology.

KVM Kernel-based Virtual Machine.

LAN Local Area Network.

NFS Network File System.

NTFS New Techology File System.

NVRAM Non-Volatile Random-Access Memory.

OS Operating System.

PC Personal Computer.

PXE Preboot Execution Environment.

RAID Redundant Array of Independent Disks.

RAM Random-Access Memory.

RCS Replication and Caching Service.

RDP Remote Desktop Protocol.

RDS Remote Desktop Services.

REST Representational State Transfer.

RFB Remote Framebuffer Protocol.

RPC Remote Procedure Call.

SSH Secure Shell.

TCP Transmission Control Protocol.

TFTP Trivial File Transfer Protocol.

UDP User Datagram Protocol.

VDI Virtual Desktop Infrastructure.

VLAN Virtual Local Area Network.

VM Virtual Machine.

VMM Virtual Machine Monitor.

Chapter 1

Introduction

1.1 Context

The concept of virtualisation, despite all the recent discussion and hype, isn't new: the technology has been around since the 1960s [11], but it was not until it was available for the x86 architecture [2] and, later on, further developed with the introduction of Intel VT [48] and AMD SVM [24] in the 2000s, that virtualisation has entered the mainstream as a must-have technology for server deployment across any production environments.

With efficient techniques that take advantage of all available resources, and continuously lowering price points on hardware, an opportunity for the advance in application architecture and a revamp in the supporting infrastructure was at hand.

However, companies realised that the cost to run an in-house, fully fledged data centre (DC) may be, in some cases, unreasonable, and that building and managing it is also a cumbersome task. And, cost is high - not as a consequence of hardware costs, but when factoring in all other parcels: cooling systems that extract the heat generated by servers, storage and networking, physical security to protect the DC, fire suppressing systems and, the largest factor, cost of manpower to maintain the infrastructure; when all these parcels are added together, the result is a high value for OPEX (operational expenditures).

To further aggravate costs, demand for instantaneous access to information coupled with an ever-increasing amount of data to handle also demands a level of resources that continues to grow every day.

This created an opportunity for the Infrastructure as a Service (IaaS) [39] cloud model that, in the case of public clouds, outsources all the resource provisioning to third parties, which are (must be) experts in maintaining very large data centres, geographically dispersed across the globe (for high availability), and can benefit from high discounts from suppliers (hardware, energy, etc.).

With major industry players building their own, very large DCs, and offering both public and hybrid cloud services, supporting more and more types of services with an increasing number of customers, new ways to store data have emerged, and distributed object storage is the platform that supports other storage paradigms – files, databases, key-value stores, etc. – that exhibit high degrees of reliability, consistency, performance, scalability, all essential to a broad range of applications with different workloads.

But, as always, there's no one size fits all solution, and clouds are no exception: some environments have peculiarities – such as low latency, and the cost to keep it under control – that dictates that computing resources must be close to the user; one of these environments is Desktop Virtualisation. VDIs, i.e., Virtual Desktop Infrastructures, are computing infrastructures that provide their users with a virtual desktop: when the user pics hisher workstation (usually PC or laptop) and interacts with it, the desktop application (e.g., Windows Desktop, CentOS Gnome, Ubuntu Unity) displayed at the device's screen is not running natively on the user's workstation. It is running in a Virtual Machine (VM).

This is the umbrella (context) of our thesisproject: an infrastructure that runs VMs that act as the user's "personal computer" and that heshe can access anywhere where a network is available (subject, naturally, to security and performance constraints), using another computer for that task. The remainder of this chapter will provide more details but, for now, we want to add that previous work was researched and developed a VDI, named iCBD, that is running and functionally "complete". Our task is to provide features that will bring fault-tolerance, high-availability and scalability to the solution (with an eye on cost).

1.2 Motivation

Virtualisation is the pillar technology that enabled the widespread availability of IaaS cloud providers, benefiting of economies of scale. These cloud providers, such as Amazon AWS [7], Microsoft Azure [40] and Google Cloud Platform [20], manage thousands of physical machines all over the globe, with the vast majority of their infrastructures being multitenant oriented.

The sheer magnitude of those numbers leads to an obvious problem, i.e., how to store all this data efficiently. And, not only there is the need to store customer's generated data but also to manage all the demands of the infrastructure and the multitude of services offered. One approach taken by these companies was the development of their proprietary storage solutions. For instance, Google uses BigTable distributed storage system [12], to store application-specific data, and then serve it to users. This system relies on the Google File System underneath to provide a robust solution to store logs and data files and was designed to be reliable, scalable and fault tolerant.

One characteristic in particular that stands out and is present in many of today's very large storage infrastructures is the use of snapshots with copy-on-write (CoW) techniques.

The adoption of such mechanisms allows for quick copy operations (a.k.a. cloning) of large data sets while saving resources and, at the same time, providing transparent concurrent operations as read-only versions of the data always are always ready for use – e.g., to perform backups - while applications simultaneously execute write operations on their snapshots. But large storage infrastructures are distributed; hence, the other important mechanisms are replication, and data distribution: the duplication of records across multiple machines, serves not only as a "security net", in case of a fault (as duplication removes the single point-of-failure characteristic), but can also be used to take advantage of network bandwidth (as one can spread a single access across multiple servers, i.e., accessing several "duplicate segments").

While the above discussion was focused on distributed storage systems, some of these features, namely snapshots, are also available in non-distributed file storage implementations. One of these recently developed systems that has a significant adoption in the Linux community is Btrfs [49].

Btrfs, whose name derives from B-tree data structures, was designed with two goals: support snapshots and maintain excellent performance in a comprehensive set of conditions. We, bounded by the iCBD's project goals, claim that the combination of Btrfs with replication and partitioning techniques opens up the way to a more advanced solution that serves the needs of an up-to-date storage system, and be easily integrated into an existing platform, serving a vast number of clients and presenting very good performance.

1.3 Project Presentation

This dissertation work was performed in the context of a more comprehensive project, named Infrastructure for Client-Based Desktops (iCBD) [37]. The iCBD project is being developed in collaboration between the *NOVA LINCS Research Center*, hosted at *DI - FCT/NOVA*, and *SolidNetworks – Business Consulting, Lda.*, a subsidiary of *Reditus S.A.*.

The project's primary objective is to develop a winning VDI product that separates itself from the current solutions (that we have dubbed server-based VDI), which execute the virtual desktops in large, resource-heavy, servers. iCBD's approach, which we dubbed client-based VDI, supports the execution of both native (Linux only) and virtual desktops (any "virtualizable" OS) on the user's physical computer, in a nonintrusive way.

1.3.1 iCBD Project

There are some leading-edge aspects of the Infrastructure for Client-Based Desktop project which sets it apart from other existing ones such as the adoption of a diskless paradigm with a remote boot, the way the platform stores Virtual Machine images, and the support for virtualised or native execution on the target workstation [1], depending on

[1] In this document workstation is a user's desktop PC, laptop, etc., any PC compatible computing device with a recent enough architecture that support modern hypervisors.

the user's choice. [35]

The remote boot of the user's workstation requires the cooperation of HTTP, TFTP, DHCP and the image repository servers - the ones that store both read-only VM templates and writable space for running instances started from those templates. Communication between workstations and the platform is over the HTTP protocol, providing both flexibility and efficiency. [3, 35, 38]

In short, the primary objectives of the project are:

- Offering a work environment and experience of use so close to the traditional one that there is no disruption for the users when they begin to use this platform.
- Enabling centralised management of the entire infrastructure, including servers, in their multiple roles, storage and network devices, all from a single point.
- Completely decouple users and workstations in order to promote mobility.
- Support disconnected operation of mobile workstations.

When all these objectives are taken into account, there is a clear distinction from other solutions currently (and previously) available. As far as we know, no other solution is so comprehensive in the use of the resources offered by the physical workstations whether they are PCs, laptops or similar devices.

1.3.2 Previous Work

The iCBD project started about two years ago, and there has been a lot of work in that period, namely by DI-FCT/NOVA students. Previous work was focused on the creation of the instances of virtual machines, using either the storage system's snapshot mechanisms (for those where native snapshots were available) or the hypervisor's own mechanism. These studies resulted in two MSc dissertations, one carrying out a more in-depth study of Btrfs (and other file systems) while the other focused itself in object-based storage, particularly on the Ceph (RADOS) product.

These two avenues, file system and object-storage system, proved themselves well suited for the iCBD project, and resulted in the next step, namely on how to support a multi-node, scalable, and failure-tolerant iCBD infrastructure, one where nodes may be geographically dispersed.

1.4 Problem Stating and Main Contributions

This dissertation aims to build upon the previous contributions, deeply study the core of the iCBD platform, and tackle the next set of questions, mainly:

- *In a geographically dispersed, multi-server iCBD infrastructure, how do we keep the VM templates consistent and available even on the presence of network faults?*

- *How do we keep a simple management interface in a distributed platform?*
- *How to scale the platform in order to handle a large number of clients and maintain or even enhance its performance?*

1.4.1 Replication and Caching - The Problem

To address the previous questions, we start with a known fact: providing high availability for data requires redundancy, and the simplest way to provide redundancy is to have replicas. Therefore, building a replication mechanism is clearly our starting point. With that in mind, we tackle a second part of the problem: if we want to provide to the iCBD clients (workstations) fast access to the data, that data must be moved, or cached, near those clients. Now, if our cache stores the full data objects that the client needs, and not just parts of those objects, caching becomes just a side-effect from replication.

Establishing Goals Providing a mechanism that ensures the correct replication of data between nodes of the platform is of paramount importance; but other objectives, such as achieving the best performance possible on data transfer operations – that is, transfer speed must be maximised, and the amount of data moved around should be minimised, and the process should not be computationally intensive – are also important. Another goal is ease of integration of the replication process with multiple technologies: the iCBD storage layer is open to different storage backends – we already mentioned Btrfs and Ceph. So, the design of the replication module should strive for an API that eases the integration with different backend technologies, as long as these do provide a snapshotting mechanism.

As far as caching is concerned, two problems must be addressed: how to plan the number and location of cache (i.e., replica) servers, and which platform services are fundamental for running images on the user's workstations and how can they be integrated with the replication and caching infrastructure in order to deliver the best possible experience to the highest number of clients.

To summarise, the following list expresses the key requirements that must drive the architecture, design, planning and execution of (the rest of) our work:

- The iCBD platform needs to be always available and maintain top-notch performance in multiple locations, while serving a considerable number of clients.
- There are two main reasons for data replication: fault-tolerance (replicas represent backups) and moving data closer to the clients.
- Taking for granted that data is close to the consumer, devise ways to deliver that data (to the client) as efficiently as possible.
- Booting a client should require the smallest number of platform functionalities possible, simplifying the boot process near the consumer.

1.4.2 Main Expected Contributions

The main expected contributions are:

- Perform a throughout analysis of the iCBD platform layers and modules that already implemented, in order to understand its internals, to document it and allow an efficient planning of the remaining of our work.
- Study, develop, and evaluate an implementation of a distributed and replicated storage platform to support VMs, built on top of Btrfs.
- Implement a client-side caching solution in order to increase availability, improve response time, and enable better management of resources.
- Integrate the solutions described above with the previous work and the existing infrastructure.
- And, finally, carry out a series of tests that a) allow us to assert that our replication and caching system is functionally complete and stable enough for production use, b) allows us to draw meaningful conclusions about its performance, and c) provide us with hints for future enhancements of the iCBD platform.

1.5 Document Structure

The rest of the document is structured as follows:

- *Chapter 2* **Research Context** - This chapter presents existing technologies and theoretical approaches which were studied; examples include storage systems and their features, and details on virtualisation techniques and hypervisors.
- *Chapter 3* **iCBD - Infrastructure for Client-Based Desktop** - In this chapter, we present the iCBD platform: we start with an overview of the solution, describe the multiple layers and try to explain the conceptual and architectural decisions that were made. This chapter is essential to the understanding of the bigger picture and where our work fits in.
- *Chapter 4* **Implementation of the *iCBD-Replication and Cache Server*** - We start with an in-depth view of the implementation of the iCBD Replication module, detailing the architectural decisions and the implemented components. Then, a description is given on the efforts to build, on the FCT NOVA campus, a server infrastructure exclusively dedicated to the iCBD project. Then we explain how to build and deploy the iCBD Cache Server software for a node that will support a student's computer lab with 15 workstations in the Computer Science Department.

- *Chapter 5* **Evaluation** - In this chapter, we present the evaluation process employed to validate our Implementation, with an emphasis on the analysis of the results we obtained and a comparison with the baseline values.
- *Chapter 6* **Conclusions & Future Work** - This chapter concludes the dissertation; we provide the answers to the questions we raised in the Introduction, summarise the results achieved in the evaluation process, and recall some ideas for further improvements, ones that were formulated during the implementation process, and we believe are a good starting point for future work.

CHAPTER 2

Research Context

The focal point of this dissertation is the implementation of a scalable and coherent distributed data store on top of a set of local (and independent) file systems. The file systems, however, are not used for "general purpose"work: in our deployment, they store VMs - (1) their read-only base images (or templates) and / or (2) their running instances backstores and / or (3) their "support files"(VM specifications, NVRAM / BIOS images, etc). Furthermore, the target file systems are those which are able to natively provide snapshots and, for the purpose of this dissertation our choice was BTRFS.

Moreover, our work should integrate smoothly into a broader infrastructure illustrated in detail in Section 3. In this chapter, we start with a survey of core concepts directly associated with the thesis and compliment with some analysis on the state-of-art in the relevant fields.

The organisation of this chapter is as follows:

Section 2.1 overviews virtualisation as a core concept, describing significant properties and the inner works of hypervisors and finishes with a comprehensive discussion about the multiple VDI models.

Section 2.2 studies the principal challenges for a storage system in a VDI context and makes a survey of the multiple types of file systems which are currently prevalent in a data centre environment.

Section 2.3 talks about the problem of the locality of the data, and how that fact can influence the performance and scalability of a system.

Section 2.4 expands on the fact the storing data in a single location is not enough for compliance with current requirements, such as high availability, fault tolerance and performance standards in critical systems.

2.1 Virtualisation

Most of today's machines have such a level of performance that allows the simultaneous execution of multiple applications and the sharing of these resources by several users. In this sense, it is natural to have a line of thought in which all available resources are taken advantage of efficiently.

Virtualisation is a technique that allows for the abstraction of the hardware layer and provides the ability to run multiple workloads on a shared set of resources. Nowadays, virtualisation is an integral part of many *IT* sectors with applications ranging from hardware-level virtualisation, operating system-level virtualisation, and high-level language virtual machines.

A Virtual Machine, by design, is an efficient, isolated duplicate of a real machine [45], and therefore should be able to virtualise all hardware resources, including processors, memory, storage, and network connectivity.

For the effort of managing the VMs, there is a need for a software layer that has specific characteristics. One of them is the capability to provide an environment in which VMs conduct operations, acting both as a controller and a translator between the VM and the hardware for all *IO* operations. This piece of software is known as a Virtual Machine Monitor (VMM).

In today's architectures, a modern term was been coined, the *Hypervisor*. It is common to mix both concepts (*VMMs* and *Hypervisors*), as being the same, but in fact, there are some details that make them not synonymous. [2]

2.1.1 Hypervisors

The most important aspect of running a VM is that it must provide the illusion of being a real machine, allowing to boot and install any Operating System (OS) available for the real hardware. It is the VMM which has that task and should do it efficiently at the same time providing this three properties [45]:

Fidelity: a program should behave on a VM the same way or in much the same way as if it were running on a physical machine.

Performance: the vast majority of the instructions in the virtual machine should be executed directly by the real processor without any intervention by the hypervisor.

Isolation: the VMM must have complete control over the resources.

A hypervisor is, therefore, both an Operating System and a Virtual Machine Monitor. It can be deployed on top of a standard OS, such as *Linux* or *Microsoft Windows*, or in a bare metal server.

To start a VM, the hypervisor kernel spins up a VMM, which holds the responsibility of virtualising the architecture and provide the platform where the VM will lie. Thus,

since the VM executes on top of the VMM, there is a layer of separation between the VM and the hypervisor kernel, with the necessary calls and data communications taking place through the VMM. This feature confers the necessary degree of isolation to the system. With the hypervisor kernel taking care of host-centric tasks such as *CPU* and memory scheduling, and network and storage data movement, the VMM assumes responsibility to provide those resources to the VM.

An hypervisor can be classified into two different types [8], depicting two virtualisation design strategies, as shown in Figure 2.1:

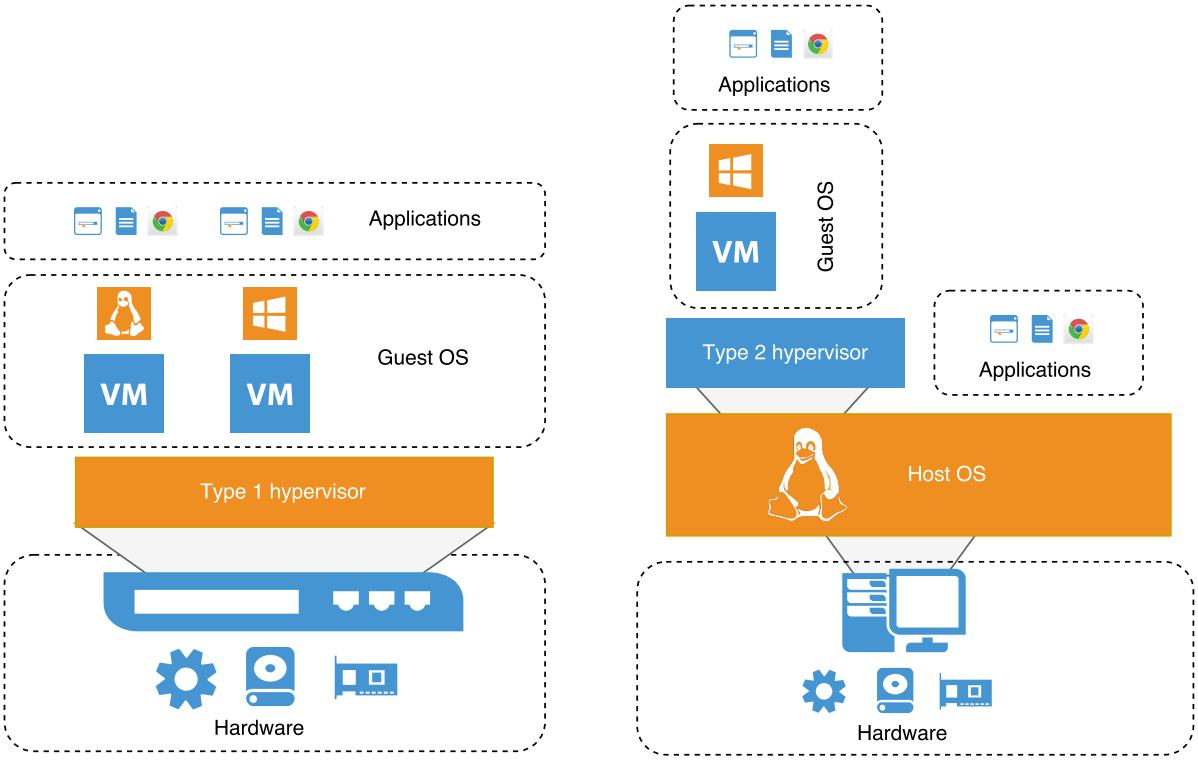

Figure 2.1: Virtualization architecture with type 1 and type 2 hypervisors

Type 1 hypervisor: Sometimes referred as a bare-metal hypervisor, since there is no need to rely on a host operating system, as it runs directly on the hardware. Moreover, it is the only program executed by the CPU in its most privileged mode. As there isn't any layer between the hypervisor and the resources, this type of hypervisor presents a more efficient solution than the Type 2.

In addition to the improved performance provided by the sharing or partitioning of devices between the several guest VMs, this architecture provides the benefit of supporting the execution of real-time OSs. The low-level nature of these hypervisors with the broad access to the hardware has proven useful for use-cases that need to deploy a multiplicity of operating systems even in mission-critical circumstances.

Recognising all the facts above, we can point that there are also some disadvantages. Any drivers needed to support different hardware platforms must be covered by the hypervisor package.

Type 2 hypervisor: This second variant of the hypervisor model relies on an already installed operating system and acts very similarly to any conventional process. Here, the hypervisor layer is a union of a host operating system with specialised virtualisation software, including extensions to that OS kernel, that will manage the guest VM. In this case, the hypervisor makes use of the services provided by the OS, which leads to a more significant memory footprint when compared to Type 1 but is integrated seamlessly with the remainder of the system. An excellent illustration of this kind of paradigm is Oracle VirtualBox and VMware Workstation/Fusion [2].

In this architecture, the host operating system retains ownership of the physical components, with each VM having access to a confined subset of those devices, and the virtual machine monitor providing an environment that emulates the actual hardware per VM.

All the above culminates in some advantages: Type 2 hypervisors are regularly deployed on desktop an laptop class of hosts, allowing: simulation of a rather complex testing virtualised systems without the expense and complexity of managing dedicated hardware; seamless integration with a graphical environment; host-guest file and print sharing.

Either way, the challenge lays in the fact that the hypervisor needs to provide to guest OS a safe execution environment and at the same time create different machine configurations to each one of them. These characteristics, such as the number and architecture of virtual CPUs (vCPU), the amount and type of memory available (vRAM), the allowed space to store files (vDisk), and so on, are user configurable but it is the job of the hypervisor to do all the resource management. The settings of these individual components reside in a VM configuration file. In the case of VMware hypervisors, the file has the `.vmx` extension,[46, 58] while in a KVM environment, that configuration is stored in a `.xml` file. [13]

With a virtualised infrastructure there is an opportunity for a substantial reduction in the number of servers which, in turn, diminishes the setup time as those VMs are, in a broad manner, created simply by cloning techniques. Software updates can be hugely simplified and made available to all VMs at once if those VMs are created on-demand from up-to-date templates at the beginning of a user session.

2.1.2 Virtual Desktop Infrastructure

It is common to find in a typical midsize corporate infrastructure hundreds of servers and thousands of workstations. All in a diverse ecosystem counting with many hardware

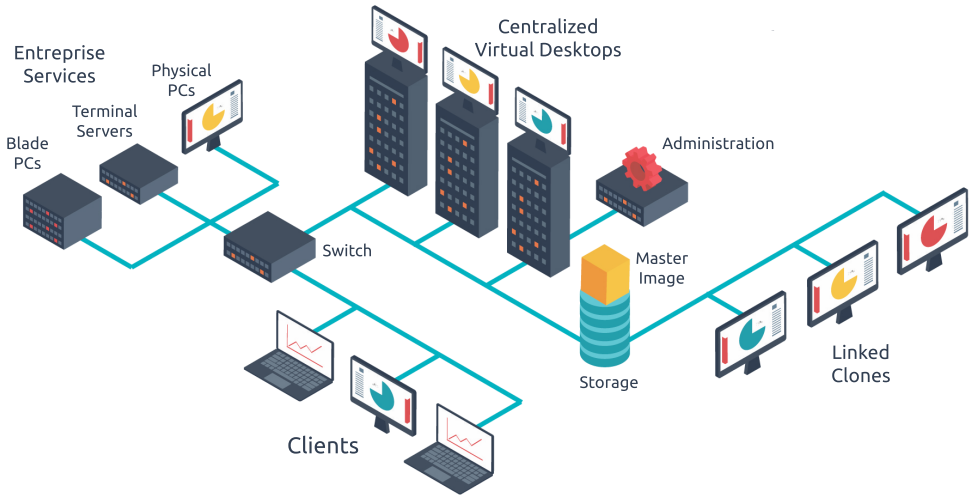

Figure 2.2: An exemple of a Virtual Desktop Infrastructure, adapted from AppDS [9]

configurations, different OSs and applications needs. Probably even supporting several versions of the same software is required for the day to day operations.

Organisations struggle daily with the traditional problem of installing software in local workstations disks one-by-one (even if employing an automated process). This task tends to be daunting as a company escalates in size and leads to some other predicaments:

- A Systems Administrator and IT Staff burden with significant infrastructure administration responsibilities and technical skills.
- A delay on the installation or reinstallation of new software and recovery from breakdowns or administration mistakes, which in large installations such intervention could take days.
- Installation processes may consume much of the available bandwidth in a network, so if this job is to be executed simultaneously on several workstations, it tends to be scheduled to off work hours to avoid disturbances.
- Periodical software updates (such Microsoft's famous Patch Tuesdays [43]) are ordinarily released in the morning's first boot of a workstation, which can bloat the traffic and render useless the workstation for the remainder of the update period.
- If an update proves to be undesirable, by introducing some unexpected behaviour, it is quite difficult to reverse this situation, which may even demand a new configuration infrastructure-wise.

One solution to the unpleasant situations outlined above is to minimise the footprint of installed software and reduce its managing needs. It is possible to conceive all the

software required to run a workstation (Operating System and applications) packaged in a single unit like a Virtual Machine. This mechanism allows for the virtualisation of a workstation that can be executed either locally on a typical PC / Laptop, or on a server. The most relevant approach and with more expression at the moment is the Virtual Desktop Infrastructure (VDI).

The concept encompasses a series of techniques, providing on-demand availability of desktops, in which, all computing is performed employing virtual machines [25]. Typically this solution offers a centralised architecture, where users desktops run in VMs, the user's environment resides on a server in a data centre, as shown in Figure 2.2. However, other components are required, such as storage for the users and VMs data and a network capable of moving large data blocks quickly, all in a perspective where from the user's viewpoint there can't be any apparent difference between a virtual desktop and a local installation.

There are two antagonistic approaches to the architecture, one focused on the server-side and the other on the client-side but both solutions are in an in-house paradigm were all configurations, management and storage needs are the responsibility of the business IT staff. A third approach emerged in recent years, with the peculiarity of being cloud-based, coined Desktop as a Service. In this section, we present a summary of the technologies above-mentioned.

Server-based VDI This is the most common approach, in which the VM runs remotely on a server through a hypervisor. In this model, the images for the virtualised desktops remain deposited in a storage system within a Data Centre. Then, when the times comes for the execution of such VM, a server that is running a hypervisor provisions the VM from storage and puts it into action. Featuring such benefit, as the fact that only a low-performance thin client with support for a protocol such as Remote Desktop Protocol (RDP) [47] or the Remote Framebuffer Protocol (RFB) [53] is required to interact with the virtual desktop.

The downside involves the costs necessary to maintain the service. Highly capable support infrastructure is needed (computing, storage, networking and power). With the additional requirement, of a need in some use cases, for adding high-end graphics processors to satisfy the workflow of customers using multimedia tools. We can still observe that the totality of the computing capacity of the hardware already present in the premises of a client prevails not harnessed. Of course, the machines already present can continue to be used, since they naturally have the resources to use the tools mentioned above, but the non-use of their full potential makes for all past investment made in hardware that pointless.

There are plenty of commercial solutions that use this principle, with the three most significant players being VMware's Horizon platform [56], XenDesktop from Citrix [60] and Microsoft with Microsoft Remote Desktop [42].

Client-based VDI In this model, the VM that contains the virtual desktop is executed directly on the client's workstation. This machine makes use of a hypervisor that will wholly handle the virtual desktop.

Since all computing work predominates on the client side, the support infrastructure (as far as servers are concerned) in this model as a much smaller footprint, having only as a general task to provide a storage environment. Alternatively, all the data could be already locally present in the hard drives of the clients, almost disowning the servers to sheer administration roles and the maintenance of other services.

The advantages remain close to the previous solution, with the added benefit of a reduced need for resources and the possibility of using some already present in the infrastructure. Although this approach presents itself as significantly more cost restrained, there isn't a notable adoption by software houses in developing products in this family. Reasons for this fact can be attributed to the implementation of such solutions that required a more complicated process, sometimes claiming the complete destruction of locally stored data on workstation hard disks. [14]. An example is a previously existing solution by Citrix, the XenClient [60]

Desktop as a Service The third, and most modern, concept incorporate the VDI architecture with the made fashionable cloud services. In some aspects shows some astonishing similarities to the server-based method, where servers drive the computation, but here, the infrastructure, the resources and the management efforts are located in the midst of a public cloud.

The points in favour are some: There is good potential for cost reduction in the field of purchase and maintenance of infrastructure since those charges are imposed on third parties. Every subject related to data security is also in the hands of the platform providers. Enables what is called zero clients, an ultrathin client, typically in a small box form factor, which the only purpose is to connect the required peripherals and rendering pixels onto the user's display. [61] With the added benefit of presenting very competitive costs per workstation when compared to other types of clients (thick and thin clients) and a reasonable saving on energetic resources.

However, in contrast, the downsides are also a few. Since the data location frequently is in a place elsewhere from its consumption, some bandwidth problems can arise, limiting the ability to handle a large number of connections. Adding to this mix is the issue of the unavoidable latency, a result of the finite propagation speed of data, which tends to escalate with the distance required to advance. Also, there is the jitter factor, caused by latency variations, which are observed when connections need to travel great lengths through multiple providers with different congestion rates. All these facts not only may lead to a cap on the numbers of clients that are able of connecting simultaneously but also can be a motive in a diminished

experience and quality of service provided, when in comparison to the previously presented solutions.

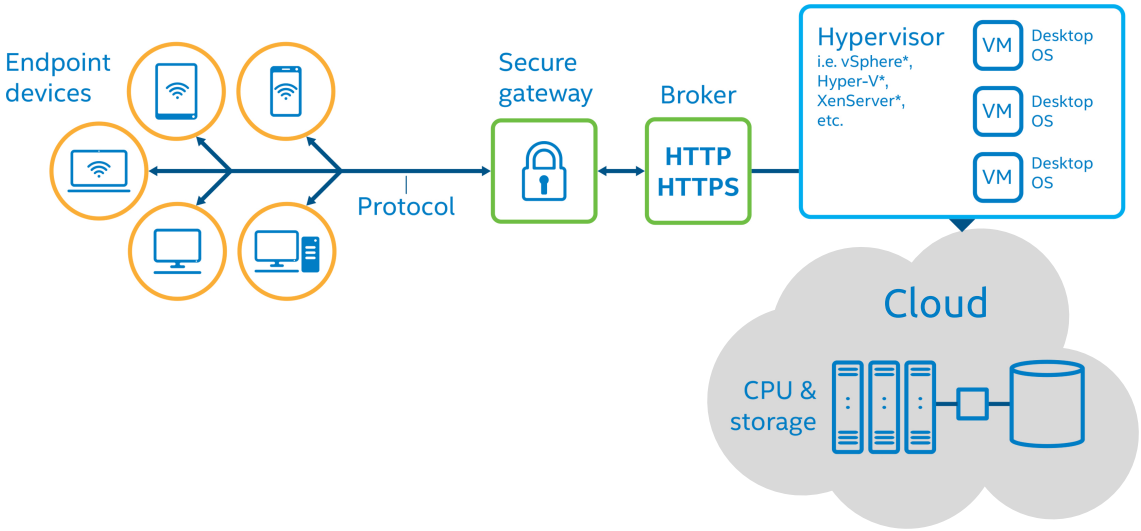

Figure 2.3: Conceptual overview of DaaS architecture, adapted from Intel [30]

In this new field, a multitude of solutions is emerging with public cloud providers leading the way. Amazon in its AWS portfolio delivers Amazon Workspaces [6], and Microsoft implements the RDS features [41] on the Azure product line. Nevertheless, there are also some smaller contenders, as an example, Workspot [59] (a company founded by ex-Citrix employees) makes use of the Microsoft Azure Cloud to provide there take on cloud-native Virtual Desktops.

2.1.3 Virtual Machine Image Storage

The data storage is one of the focal points to address in this work. Therefore, it is meaningful to understand how a virtual machine is composed and how is translated to a representation in a storage device.

The basic anatomy of a Virtual Machine encompasses a collection of files that define the VM settings, store the data (i.e. Virtual Disks) and save information about the state of its execution. All of these data and metadata need to be deposited on storage devices of whatever type.

VMware Architecture Given the architecture presented by VMware software [58], the main files required for the operation of a VM are:

- *The VM configuration file* - The `.vmx` file holds the primary configuration options, defining every aspect of the VM. Any virtual hardware assigned to a VM is present here. At the creation time of a new virtual machine, the configurations regarding the guest operating system, disk sizes, and networking are appended to the `.vmx`

file. Also, whenever an edit occurs to the settings of a virtual machine, this file is updated to reflect those modifications.

- *The virtual disk files* - Embodying multiple `.vmdk`, which stores the contents of the virtual machine's hard disk drive and a small text disk descriptor file. The descriptor file specifies the size and geometry of the virtual disk file. Also includes a pointer to the full data file as well as information regarding the virtual disks drive sectors, heads, cylinders and disk adapter type. The virtual disk actual data file is conceived while adding a virtual hard drive to a VM. The size of these files will fluctuate based on the maximum size of the disk, and the type of provisioning employed (i.e. thick or thin provisioning)

- *The file that stores the BIOS* - The `.nvram` file stores the state of the virtual machine's BIOS.

- *The suspended state file* - The `.vmss` saves contains the state of a suspended virtual machine. This file is utilised when virtual machines enter a suspended state giving the functionality of preserving the memory contents of a running VM so it can start up again where it left off. When a VM is returned from a suspend state, the contents of this file are rewritten into the physical memory of the host, being deleted in the event of the next VM Poweroff.

- *Log files* - A collection of `.log` files is created to log information about the virtual machine and often handled for troubleshooting purposes. A new log file is created either during a VM power off and back on process, or if the log file stretches to the maximum designated size limit.

- *The Swap file* - The `vswp` file warehouses the memory overflow in case the host cannot provide sufficient memory to the VM, and Ballooning technique cannot be employed to free memory [27]

In addition to the records described above, there may be some more files associated with the use of snapshots. More concretely, a `.vmsd` file and multiple `.vmsn`. The first is a file with the consolidation of storing and metadata information about snapshots. The other one, represents the snapshot itself, saving the state of the virtual machine in the moment of the creation of the snapshot.

The implementation of snapshots mentioned above applies to a specific implementation of VMware and takes form as follows: first, the state of the resource is stored in the form of an immutable and persistent object. Then, all modifications that transform the state of the resource are gathered in a different object. The diverse snapshotting techniques are addressed in a more comprehensive sense in the Section 2.2.3.

2.2 Storage

As stated in previous sections, the main problem to be addressed in this work is the storage concerning virtual machines. That could be either images, snapshots, files or data structures that are needed to support the execution of a VM.

When applied to the VDI concept some demands appear in the form of specific care needed at planning the storage system architecture, as well as the supporting infrastructure: the hardware picked, network topology, protocols used, and software implemented.

At the end of the day, the idea is to present a solution that offers an appropriate cost to performance ratio, and that with little effort can scale when the need emerges.

2.2.1 Storage Challenges

In a typical data centre application, with a well-designed infrastructure and in normal conditions, the storage system is steadily used but isn't being stressed continuously with requests I/O requests that directly affect the system performance. However, that postulate is no longer valid when talking about the storing of VM files for use in a VDI environment. In this type of context, some events can cascade in I/O storms that eventually introduce degradations in storage response time, which diminishes the performance of the overall system and in turn leads to a lower satisfaction level for the users of said system.

I/O Storms In a typical data centre application, with a well-designed infrastructure and in normal conditions, the storage system is steadily used but isn't being stressed continuously with I/O requests that directly affect the system performance. However, that postulate is no longer valid when talking about the storing of VM files for use in a VDI environment. In this type of context, some events can cascade in I/O storms that eventually introduce degradations in storage response time, which diminishes the performance of the overall system and in turn leads to a lower satisfaction level for the users of said system. From several events that influence a storage system we can point out some that have more expression in a VDI setting:

Boot Storm It may happen, on the occasion of the starting a work shift; with several users simultaneously arriving at their desk and booting their workstations. In this circumstance, all VMs are simultaneously performing multiple read and write operations on the storage system, which translates into poor response times and a long wait for the end of the boot process.

Login Storm Right after the booting an OS, the workstations are not entirely operational users have to log in to access a desktop, including applications and files. This procedure, results in a considerable number of concurrent I/O requests from multiple VMs in a short time, as the system attempts to load quite a few files related to the user's profile.

Malware and Anti-Virus Software Scanning Usually scans for unwanted files and untrusted applications, are scheduled to execute at a time when they cause the least possible impact taking into consideration the load of the machine. However, it is not uncommon to observe a behaviour where this kind of software starts a scan right after boot. Alternatively, the unfortunate case where different machines decide to start that examination at the same time, causing a negative impact on every machine.

Big Applications Needs Some applications can be very I/O intensive, like loading a project in an IDE with numerous libraries and dependencies that need to be reviewed at startup. Also, we can envision a scenario where multiple users simultaneously open the same very resource intensive application, for instance, in a classroom, the teacher asks the students to start a particular application, the I/O requests to the storage system will most likely be simultaneous.

Operating System Updates Similarly to Malware and Anti-Virus Software Scanning, the update process of an Operating System will most likely be tied to a schedule that is based on the current load of a machine. Yet, multiple systems may decide to perform an update in the same space of time thus leading to the bottleneck problem of concurrent access to the storage system.

2.2.2 File Systems

The traditional and perhaps most common way of storing files and, in turn, VMs is the use of file systems. This kind of system is used to manage the way information is stored and accessed on storage devices. A file system can be divided into three broad layers, from a top-down perspective we have:

- The **Application Layer** is responsible for mediating the interaction with user's applications, providing an API for file operations. This layer gives file and directory access matching external names adopted by the user to the internal identifiers of the files. Also, manages the metadata necessary to identify each file in the appropriate organisational format.

- Then the **Logic Layer** is engaged in creating a hardware abstraction through the creation of logical volumes resulting from the use of partitions, RAID volumes, LUNs, among others.

- The last one is the **Physical Layer**. This layer is in charge with the physical operations of the storage device, typically a disk. Handling the placement of blocks in specific locations, buffering and memory management.

There are many different types of file systems, each one boasting unique features, which can range from security aspects, a regard for scalability or even the structure followed to manage storage space.

Local file systems: A local filesystem can establish and destroy directories, files can be written and read, both can move from place to place in the hierarchy but everything contained within a single computing node. Good performance can be improved in certain ways, incorporating caching techniques, read ahead, and carefully placing the blocks of the same file close to each other, although scalability will always be reduced. There are too many file systems of this genre to be here listed. Nevertheless, some of the most renowned may be mentioned. As the industry-standard File Allocation Table (FAT), the New Technology File System (NTFS) from Microsoft, the Apple's Hierarchical File System Plus (HFS+) also called Mac OS Extended and the B-tree file system (BTRFS) initially designed by Oracle.

Distributed file system: A distributed file system enables access to remote files using the same interfaces and semantics as local files, allowing users to access files from any computer on a network. Distributed file systems are being massively employed in today's model of computing. They offer state-of-the-art implementations that are highly scalable, provide great performance across all kinds of network topologies and recover from failures. Because these file systems carry a level of complexity considerably higher than a local file system, there is a need to define various requirements such being transparent in many forms (access, location, mobility, performance, scaling). As well as, handle file replication, offer consistency and provide some sort of access-control mechanisms. All of these requirements are declared and discussed in more detail in the book "Distributed Systems: Concepts and Design" by George Coulouris et al. [15] We can give as example of file systems the well-known Network File System (NFS) [50] originally developed by Sun Microsystems, and the notable Andrew File System (AFS) [51] developed at Carnegie Mellon University.

There are numerous types of additional file systems not mentioned since they are not in the domain of this work. Still, it is important to note the existence of an architecture that is not similar to the traditional file hierarchy adopted in file systems, which is the object-based storage.

This structure, as opposed to the ones presented above, manages data into evenly sized blocks within sectors of the physical disk. It is possible to verify that it has gained traction leading to the advent of the concept of cloud storage. There are numerous implementations of this architecture, whether in small local deployments or large-scale data centres supporting hundreds of petabytes of data. This type of file system is being studied in the context of a parallel thesis but inserted in the same project already presented.

It is worthwhile to enumerate some examples such as CephFS [57], OpenStack Swift [52], and in a IaaS flavour the Amazon S3 [4] and Google Cloud Storage [18].

2.2.3 Snapshots

In this work, the snapshot functionality of the file system itself is a valuable asset. This technique is present in some of the most recently designed file systems, such as the BTRFS. As the name implies, a snapshot is an image at a given instant of the state of a resource, we are particularly interested in snapshots of volumes (logical disks), and of files (individually or grouped, for example, in a directory).

The implementation of a snapshot can be described as follows: a) the state of a resource is saved in the form of a persistent and immutable "object"; b) changes to the state of the resource forces the creation and storage of another object. Consequently, it is possible to return to any previous state, as long as, the object corresponding to that state is available. Snapshots are especially interesting in virtualised environments because the hypervisor can take snapshots of the most critical features of a VM: CPU, memory, and disk(s).

In this work, we propose to use the snapshot functionality of the file system itself, present in some of the most recently designed file systems, such as the BTRFS. This way the creation of linked-clones is handled by the file system capabilities as an alternative to linked-clones created by virtualisation software itself

In order for multiple snapshots, do not take up space unnecessarily, data compression techniques are applied when implementing snapshots. So, the new object created to register the sequence of new changes of a resource only registers the modifications made, keeping unchanged the state in the initial (parent) object. This phenomenon (i.e. the changes between the current snapshot and the previous one) is called a "delta"connecting snapshots.

2.3 Caching

A cache can be defined as a store of recently used data objects that is nearby one client or a particular set of clients than the objects themselves. The inner works of one of these systems are rather simple. When a new object is obtained from a server, it is added to the local cache, replacing some existing objects if needed. That way when an object is requested by a client, the caching service first checks the cache and supplies the object from there if an up-to-date copy is available. If not, an up-to-date copy is fetched, then served to the client and stored in the cache.

Caching often plays a crucial role in the performance and scalability of a file system and is used extensively in practice.

Caches may be found beside each client or they may be located on a server that can be shared by numerous clients.

Server-side Cache: Server side caching is when the caching data occur on the server. There is no right way to the approach of caching data; it can be cached anywhere and at any point on the server assuming it makes sense. It is common to cache

frequently used data from a DataBase to prevent connecting to the DB every time some data is requested. In a web context, it is common to cache entire pages or page fragments so that there is no need to generate a web page every single time a visitor arrives.

Client-side Cache: Maintaining the analogy to the Web environment, caches are also used on the client side. For instances, Web browsers keep a cache of lately visited web pages and other web resources in the client's local file system. Then when the time comes to serve a page that is stored in the cache, a special HTTP request is used to check, with the corresponding server, if the cached page is up-to-date. In a positive response the page is simply displayed from the cache, if not, the client just needs to make a normal request.

2.4 Replication

At the storage level, replication is focused on a block of binary data. Replication may be done either on block devices or at the file-system level. In both cases, replication is dealing with unstructured binary data. The variety of technologies for storage-level replication is very extensive, from commodity RAID arrays to network file system. File-based replication works at a logical level of the storage system rather than replicating at the storage block level. There are multiple different methods of performing this. And, unlike with storage-level replication, these solutions almost exclusively rely on software.

Replication is a key technology for providing high availability and fault tolerance in distributed systems. Nowadays, high availability is of increasing interest with the current tendency towards mobile computing and consequently the appearance of disconnected operation. Fault tolerance is an enduring concern for does who provide services in critical and other important systems.

There are several arguments for which replication techniques are widely adopted; these three are of significant importance:

Performance improvement: Performance improvement: Replication of immutable data is a trivial subject, is nothing more than a copy of data from one place to another. This increases performance, sharing the workload with more machines with little cost within the infrastructure.

Increased availability: Replication presents itself as a technique for automatically keeping the availability of data despite server failures. If data is replicated in additional servers, then clients may be able to access that data from the servers that didn't experience a failure. Another factors that must be taken into account are network partitions and disconnected operation.

Fault tolerance: There is the need o maintain the correctness guarantees of the data in the appearance of failures, which may occur at any time.

Chapter 3

iCBD - Infrastructure for Client-Based Desktop

The acronym Infrastructure for Client-Based Desktop (iCBD) stands for Infrastructure for Client-Based (Virtual) Desktop (Computing), a platform being developed by an R&D partnership between *NOVA LINCS*, the Computer Science research unit hosted at the *Departamento de Informática of Faculdade de Ciências e Tecnologia of Universidade NOVA de Lisboa* (DI-FCT NOVA) and *SolidNetworks – Business Consulting, Lda*, a subsidiary of *Reditus S.A.* group.

iCBD's primary goal is to implement a client-based VDI, a specialized form of VDI where all computing chores – from graphical display to application execution – are performed directly on the user's workstation (PC/laptop, etc.) hardware as opposed to performed on big and expensive servers, as it goes with mainstream VDI implementations such as the ones from Citrix, Microsoft or VMware, to name the most relevant ones. Furthermore, iCBD, while using the workstation's hardware, does not touch the disk – either to load software or as a temporary scratch device: it runs diskless. And, however, it does offer a simple and centralised administration of the infrastructure, even when it spans multiple sites.

This chapter will address the project's central concepts and associated technologies:

Section 3.1 overviews the project's core concepts and address note peculiarities and limitations of mainstream implementations in contrast with our approach.

Section 3.2 studies the main architectural components of the platform, with emphasis on the different layers and how they act together to serve the end-user.

3.1 The Concept

iCBD, as a project, aims to research an architecture that leads to the development of a platform that we call client-based VDI, while maintaining all the benefits of both client-based and server-based VDI. Additionally, it should be deployable as a Cloud Desktop as a Service (DaaS) without any of the drawbacks of current DaaS offerings.

In short, our aim is to preserve the convenience and simplicity of a fully centralised management platform for Linux and Windows desktops, instantiating those in the users' physical workstation from virtual machine templates (VMs) kept in repositories. We will further address this subject in Section 3.2.

To summarise the iCBD platform should be able to:

- Within the boundaries of the proposed architecture, adapt to a wide range of server configurations.
- Provide an user experience so close to the traditional one, that users should not be able to tell it from a PC standard (local) install.
- Simplify installation, maintenance and platform management tasks for the entire infrastructure, including servers in their multiple roles, storage and network devices, all from a single point of administration.
- Allow for a highly competitive per-userworkstation cost.
- Maintain an inter-site solution to support efficiently, e.g., a geographically disperse multi-site organisation.

3.2 The Architecture

The iCBD platform encompasses the use of multiple services; to achieve a better understanding of its inner workings, we can group these services in four major architectural blocks, as seen in Figure 3.1.

iCBD Machine Image (iMI) embodies the required files to run a iCBD platform client; this nomenclature was borrowed and adapted from Amazon Web Services' AMI [5]. An iMI includes a VM template (with an operating system, configurations and applications), the iCBD boot package (a collection of files needed for the network boot and tailored to the operating system) and an assortment of configurations for services like PXE and iSCSI.

Boot Services Layer responsible for providing the initial code that supports network boot of client machines, the transfer a bespoke follow-up package (OS, ramdrive, initial scripts), using services such as PXE, DHCP, TFTP and HTTP.

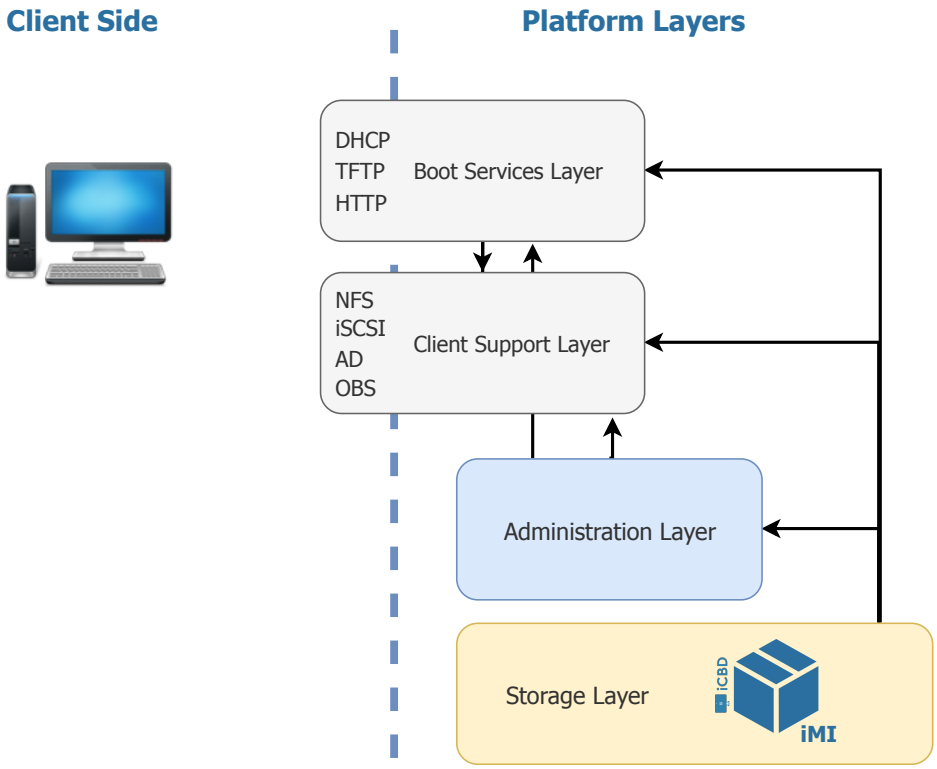

Figure 3.1: iCBD Layers View

Client Support Layer provides support for client-side operations including, e.g., authentication, read/write storage space for client instances (since iMIs run on "diskless"workstations) and access to the users' home directories.

Administration Layer maintains platform users and the full iMI life cycle, from creation to retirement. Currently, administration is based on a custom set of scripts.

Storage Layer maintains the repository of iMIs and provides essential operations such as version control of the VM image files. Our work is fundamentally focused on this layer, extending it in such a way that a single repository abstraction can be built on top of the localindividual repositories through replication and caching. These local repositories are implemented on Btrfs or Ceph and may be exported to clients using NFS, iSCSI, REST and other suitable protocols.

In the next subsections, we will provide a more detailed description of each of the above-mentioned layers.

3.2.1 iCBD Machine Image

In its essence, an iCBD Machine Image is an aggregation of everything that is needed to run an Operating System within the iCBD platform – kernel, binaries, data and configuration files. For the sake of simplicity, we categorise iMI files into three main groups:

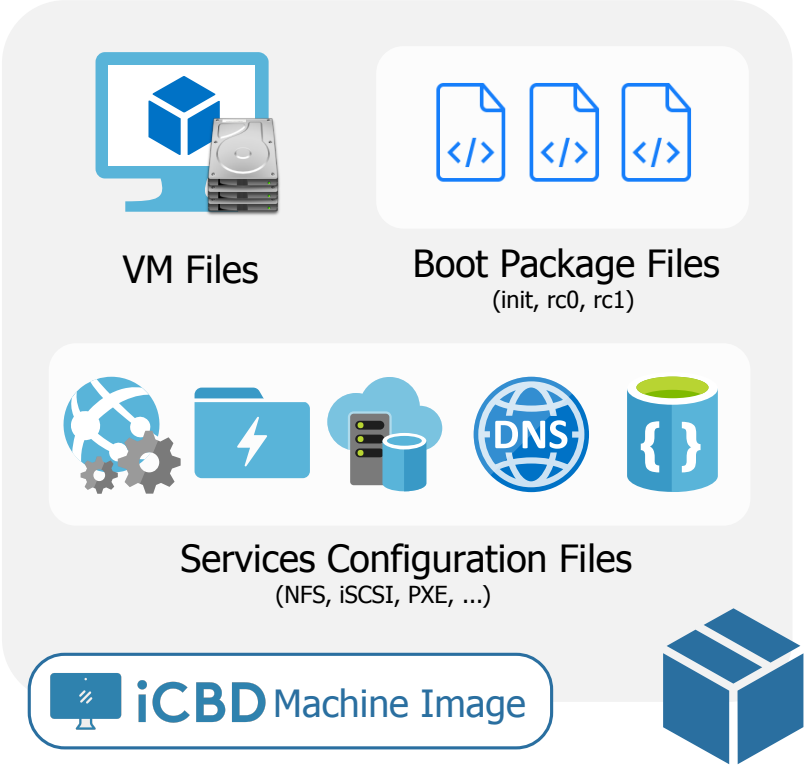

Figure 3.2: iCBD Machine Image Files

VM Template files The main component of this group is the virtual machine template in the form of a read-only image. As described in Section 2.1.3, the anatomy of a template follows the standard VMware and KVM formats either with multiple files (i.e., Virtual Disk Files like `.vmdk` or `.qcow`) or a *raw* storage format (a disk image).

iCBD Boot Package files In a network boot environment, such as the one used, there is a need to keep a set of files that manage the boot process of the user's workstation; these files can be included in the initial *ramdisk* or transferred over HTTP later on, when needed. Included in the boot package are: a *BusyBox* tool, an init file, and at least two Run Control Script files (`rc0` and `rc1`) that are responsible for starting network services, mount all file systems, and ultimately bring the system up into the single-user level. With *BusyBox* (a single executable file with a stripped-down set of tools), a basic *shell* is available during the boot process to fulfil all the required steps.

Service Configuration files The iCBD platform uses several services, and some do require particular settings in the configuration files. As an example, the "NFS exports"configuration file should reflect which file systems are exported, which networks a remote host can use, as well as a myriad of options that NFS allows; the same happens to iSCSI, where an iSCSI target needs to refer to a backing store for the storage resource where the image resides.

iMI Life Cycle The life cycle of an iMI encompasses all stages that take it throughout its course within the platform; Figure 3.3 shows the major ones, from creation, through deployment, when in use by multiple clients and, finally, its retirement and placement in to temporary or cold storage.

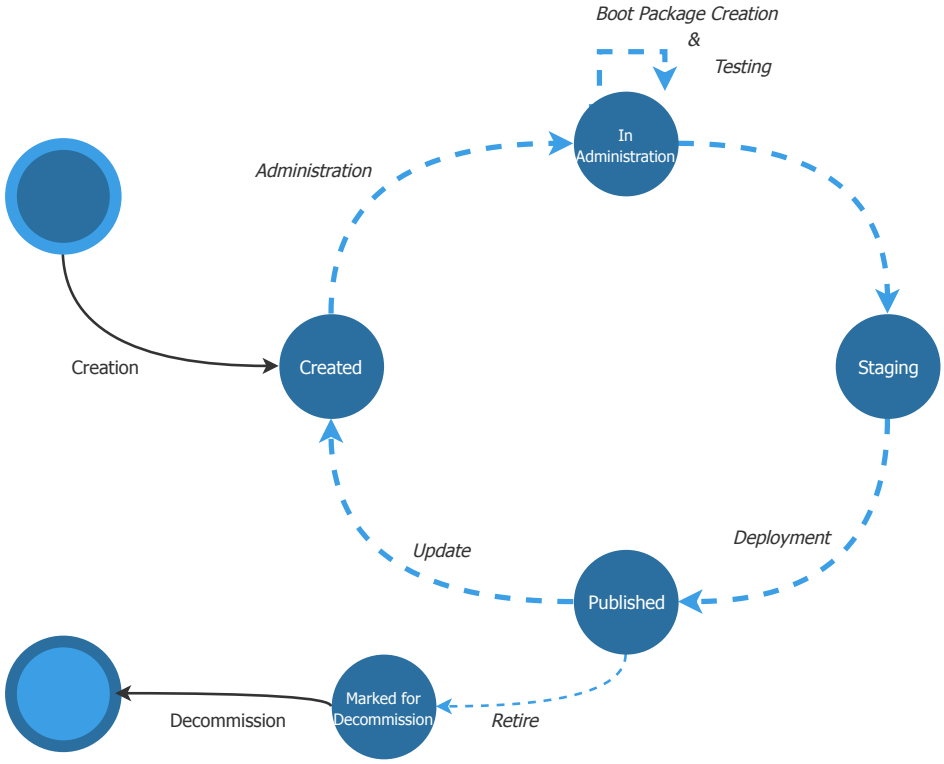

Figure 3.3: iMI Life Cycle inside the iCBD Platform

When an iMI completes a full cycle, a new version is created; so, every new update made to an iMI will spawn a new version. The creation of a new version is a rather straightforward and computationally light operation, thanks to the snapshotting features available at the storage layer.

During its life in the platform, an iMI can be in one of the following four main states:

Created After being inserted into the platform, an image is not instantaneously ready to be served to clients and booted in a workstation; it must pass through a number of administration steps for the generation of the appropriate boot package.

In Administration An iMI goes through this phase in two moments: the first one, described above, when an image has just been injected (created) into the platform; and the second, and most frequent case, when an image needs to be updated or, in any way, modified. The iMI will stay in this state as long as it is being managed, which can take from a few minutes to hours; at the end this process the boot package is automatically created.

Not Published This is the status of an image that is ready but isn't yet published, and is therefore not visible to platform users. This phase is of particular interest for testing it for correctness of the boot process, and to ensure that the modifications were, in fact, applied. Only after testing should an iMI made available for general use.

Published This state corresponds to the deployment of an iMI into production and is the one where the iMI is expected to spend most of its time. iMIs in this stage have their entries displayed at the user's workstation screen at boot time, and all the necessary support for their execution is available at the Boot Services Layer. So, if the user chooses one of those iMIs, no matter what device (e.g., PC or laptop) heshe is using, the image is expected to boot to completion, and the user should be able to login and work just as if the boot was from a local disk.

When an iMI completes a cycle and undergoes an update process, the old version is retired and set to a new state, named **Marked for Decommission**, which is comparable to a stay in limbo. First, because when the administration process was initiated clients could be using that same image, so it must be available for them (or else a disruption would happen). And, when the administration process finishes, clients may still be using the old version. Therefore, **Decommission** is only triggered when the last client "closes" its session. At that moment, the old version can be removed entirely from the platform or, more wisely, stored as a backup e.g., to allow the administrator to retrieve an older state of the image (for example, a newly installed update breaks some application).

3.2.2 Boot Services Layer

From an end-user perspective, the only layer that is visible and, for "power-users", requires interaction, is the boot layer – one that displays the set of images the user is allowed to boot, and waits for his/her choice. However not every single aspect is noticeable: while this happens in the workstation's screen, in the background multiple services cooperate to run the chosen iCBD Machine Image.

The iCBD platform provides two distinct ways to remote boot an iMI: one instantiates, from an iMI, a native Operating System that runs on the workstation's "bare metal" just like a standard diskless network boot of, say, Linux does; the other uses the above mechanism to start a minimal OS with an embedded hypervisor installed, then runs the hypervisor, and finally launches another iMI, one chosen from those available in the platform. Both approaches are entirely transparent to the user and, users who are not knowledgeable about virtualisation technologies will be completely unaware of whether they are running a native or virtualised OS.

The first part of the boot process runs like any other network boot: a series of DHCP requests are used to provide suitable network parameters - particularly the location (IP address) of the TFTP server and, then, a small network boot manager program, is

transferred, loaded and executed. Using the standard PXE boot environment, a friendly looking, tailored graphical menu, displays an assortment of choices that announces the different iMIs ready to boot.

Booting an iMI in a Workstation After the selection, in the PXE [22] boot menu, of one of the available iMIs , the second-stage boot kicks in, using *PXELINUX* as a bootloader. That provides us with the capability of transferring a compressed Linux Kernel (`vmlinuz`) and an initial ramdisk (`initramfs`) [23] using either TFTP or HTTP. In this step, a number of parameters needed for proper operation are set with their appropriate values based on the requirements of the chosen image. After loading everything into memory (RAM), the second-stage boot loader runs the kernel image which, after initialisation, starts the first user-space application – usually, the init program.

In the iCBD platform init starts a chain of execution of two custom files, `rc0` and `rc1`; these Run Control scripts configure every single aspect in the OS according to the hardware characteristics of physical machine that is booting. The first step is to reconfigure the network interface card and obtain IP connectivity. Then, a check is performed to see if there is a need of getting more files to complete the boot process and, if necessary, those files are transferred. The next script, deals with data volumes and mounting operations – R/W space, user home directories, etc.. If the image's OS is just the platform to run another iMI in virtualisation mode, more configuration steps are performed, e.g., a check is made to determine if the base OS file system happens to be Btrfs: if true, seeding must be used in order to create a RW instance from an R/O iMI. Alternatively, or if the iMI OS is not Btrfs, an overlay filesystem is used to create the RW instance.

After these configuration steps, the `switch root` command is executed moving the (already mounted) filesystems `/proc`, `/dev`, `/sys`, `/tmp` and `/run` to a new root and turning it into the new root filesystem, performing some housekeeping, namely erasing all files not in use in the `initramfs` root, releasing any unused memory.

Finally, the remaining configuration steps include setting of the current time with the NTP service and logging some statistics such as the elapsed time and the bandwidth consumed by the boot process as a whole.

3.2.3 Administration Layer

One of the most important features provided by the platform is the ability to perform administration operations on an iMI, a task accomplished with the help of an administration tool which enables an iCBD administrator (or architect, if we draw a parallel with a role commonly found in private cloud infrastructures) in an organisation to make the changes heshe deems necessary (e.g., update the OS and applications, add or remove software, and modify configurations) and then publish the new image (version) for widespread use.

The Administration Process The administration tool consists of a main script, `adm`, which then calls a series of others, depending on the task at hand. Calling `adm` with the name of one iMI as an argument starts an administration appliance - a VM with a custom-tailored base image that will support the administration process. Usually, the VM guest OS will be Linux, with several distributions supported: openSUSE Leap 42.2, Fedora 27, CentOS 7 and Ubuntu 16.04 LTS. The whole administration process makes extensive use of the snapshotting capabilities of the Storage Layer (whether using Btrfs or Ceph), with no (predictable) performance degradation on the other iCBD platform services.

For each iMI, there is a snapshot with an index number that relates to its version and age (i.e., higher numbers represent more recent versions); that index is used as the name of a directory, and the snapshot (and related files) are kept inside that directory. So, creating a new linked clone from the latest version of a VM becomes a very simple process.

When booted, the administration VM will start a hypervisor (VMware Workstation, VMware Player or KVM) and the hypervisor will be instructed to boot a linked clone created on-the-fly from the VM version (i.e., the iMI) that the administrator wishes to update. Thus, this process will, if executed in the iCBD administration server itself, use nested virtualisation [16] to achieve its goal, which may result in some performance degradation (even considering the use of a Type 1 hypervisor, such as KVM). However, in theory, nothing prevents the administrator from using hisher workstation in native mode or a server with a bare-metal hypervisor and run the administration VM using only one level of virtualisation.

In this step, the (newly created) snapshot being managed is in a temporary directory, one whose lifetime is the duration of the procedure. This method serves two purposes: first, all clients using the iMI version that is undergoing changes can keep using it; and, last, one may quickly discard all changes made in the working directory version if one wishes so.

When all modifications have been made to the temporary image, the administrator is given the option whether to commit or discarding them; if one commits, the temporary (working) directory is used to create a new linked clone (of the temporary snapshot), but one that follows the naming rules, and may, therefore, be used as the name of the new directory.

Creating the boot package After completion of the above-described steps, the iMI is not yet ready to be published, as there are no files to support the boot process; the next step is, therefore, the creation of a boot package. This is the responsibility of the `mki` script, one that, depending on the type of the iMI, may be called immediately when the Administration Process is over by the `adm` tool.

The procedure is different for Windows or Linux iMIs: while sharing a common set of steps, Linux iMIs require an additional number of customisation steps because iMIs for Linux have two sub-types – one for iMIs intended to run natively on a workstation, and

the other for IMIs that serve as hosts for other images. The first type requires the creation of custom `initramfs` and `vmlinuz` files, the addition of a subset of the image's Kernel drivers and firmware (namely for all available network interfaces and the filesystems used during the boot process), and the customisation of Run Control scripts (`rc0` and `rc1`) that start the network services with a suitable configuration, compatible with the iCBD platform, and mount the correct remote file systems. At the end of this process, a script called `runvm` is also added; that script is instrumental in starting a virtualised iMI, as well as in configuring the hypervisor parameters in order to take advantage of the client's hardware.

However, the job of the `mki` script is not yet complete: for the two types of iMIs, the configuration of both the *iSCSI targets* (to reflect the new iMI version) and the *pxelinux* (to reflect the new paths to the files that will be served to the clients) must be updated.

3.2.4 Client Support Layer

The Client Support Layer is the most fluid of all layers, containing an aggregation of services (most of them originated outside the platform) working together to provide the environment that is required for a client workstation to operate correctly. Another essential service provided by this layer is its relation with storage layer, one which, using protocols such as NFS and iSCSI, allows the client to provide the necessary data for the boot process and obtain readwrite space to support the changes made to the client's storage instance (derived from the iMI). Moreover, these protocols are the ones that provide access to the user home directory and other volumes, if supported.

It is important to note that there are other services that are essential to the correct operation of clients in the iCBD platform, such as DNS and NTP. And there are also services that, while not directly related to the iCBD platform, are nevertheless need to support clients; as an example, in a medium/large organisation Microsoft's Active Directory plays a paramount role – therefore it is necessary for the iCBD platform to coexist with this type of service, and Windows iMIs can be prepared in a way that, when an instance is booted, it will "automatically" join the organisation's AD.

3.2.5 Storage Layer

The most significant part of our work will focus on this layer, which is responsible for storing all platform data, whether they are iMIs, virtual disks (`.vmdk`) for VMs such as the administration VM and others that support platform services, databases that preserve information and configuration data for those services, code repositories, etc.. In its essence, the Storage layer consists of a set of file systems (or, if OBS-based, of object stores) each with its own purpose.

To keep this document short, we will only focus on file systems that provide storage for iMIs. Given the uniqueness of this type of data, the file system must have the right set of features, and of particular importance is the support for snapshots; as we have

previously referred, our choice is Btrfs. It is important to mention that this is not the only solution that supports snapshots – object stores such as RADOS (from Ceph) are particularly well suited to store very large objects, such as virtual disks, but that avenue is being researched in another project.

The Btrfs features that are heavily used in the iCBD platform are: sub-volumes; snapshots; cloning of both sub-volumes and snapshots; Btrfs seeding; and incremental backups, just to name a few. These features are used achieve several goals: multiple sub-volumes are used to store different parts of the platform data, snapshots are widely used to create (a kind of) version control for iMIs (allowing us to access any version at any moment) and to backup the entire iCBD platform. Btrfs seeding is another important feature, as it provides a mechanism that allows multiple read-only mounts of the same Btrfs file system, thus enabling multiple clients to use the same iMI.

One must remember that, while this work is focused on Btrfs, from the point of view of the remaining layers the type of store - filesystem or object-based - should be entirely transparent to the other layers, which will interface iCBD "data objects" through standard, widespread protocols such as NFS or iSCSI.

Therefore, when dealing with the problem of data replication across multiple file systems transparency, as an attribute, is fundamental. We will discuss in detail the replication and caching topics in the next chapter, and we will provide, then, an explanation of the decision-making process and a discussion on the implementation issues.

Chapter 4

Implementation of the *iCBD-Replication and Cache Server*

This chapter is about the core of this thesis: the iCBD-Replication and Cache Server, a middleware system that provides replication features in an integrated way to the iCBD platform. Naturally, the first subsection provides a detailed description of the motivation, architecture and design aspects while the second subsection concentrates on the implementation. Finally, the last subsection deals with the process of building the Replication and Cache Server: that is, how the modules are installed and deployed in an iCBD infrastructure.

Section 4.1 begins by explaining the motivation for the implementation of a replication module within the iCBD platform, as well as explaining the need to include a new role in the platform with the creation and deployment of a cache server near the clients. Concluding with the overall architecture of the system and how both topics complement each other.

Section 4.2 overviews the implementation of the middleware dubbed iCBD-Replication. Beginning the journey through the initial architectural process and then showing the implemented components and their interaction with the several layers of the platform.

Section 4.3 shows how the complete iCBD platform was installed in the NOVA University cluster. Then, a description of the work performed to include a client-side caching server directly connected to the final clients.

4.1 Motivation and System Architecture

One of the key objectives of iCBD is to provide a platform that can stretch from a fully hosted, on-site architecture to one where an important part of its services is cloud-based. Naturally, in-the-middle solutions are also interesting, such as one where, e.g., both the administration and storage of *"golden"* iMIs are cloud-based, while the rest is on-premises. Therefore, it becomes evident that data locality is an important subject, and thus one must study how data flows between the components of the iCBD platform – a complex, data-intensive platform, boasting multiple storage devices and many different networks, accessed by multiple consumers demanding data at any given time.

All these factors, allied to the desire of having a distributed and scalable platform architecture, result in the need to design a new service whose mission is to ensure that data is correctly replicated in the appropriate places, and consistency of the various versions of the iCBD Machine Images stored is maintained. This led to the high-level architecture of Figure 4.1

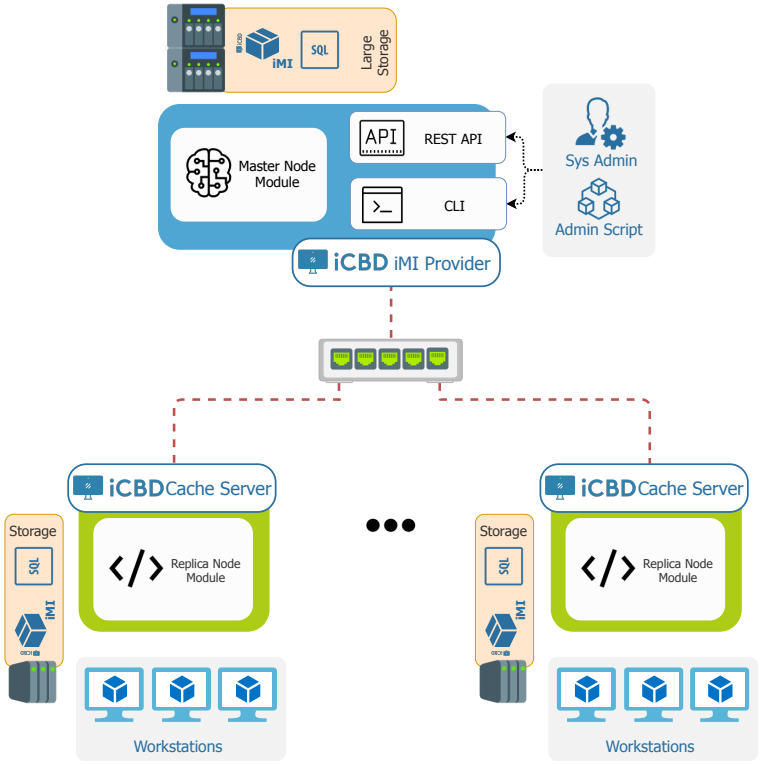

Figure 4.1: iCBD Replication and Caching Architecture (high-level)

4.2 Implementation of a Replication Module

When our project started, iCBD was already available; therefore, our Replication and Caching Service (from now on, RCS) had to be integrated with the existing architecture and support (and enhance) the previous work. This influences the choices available for the

RCS implementation and could, in a worst-case scenario, severely limit them. Fortunately, that was not the case.

iCBD was designed from the start based on the assumption that the storage layer where iMIs live was built on top of a storage system which supports snapshots. In the first prototype, Btrfs was chosen for the storage layer but, in theory, any storage system that supports snapshots can be used in the platform. That is, in fact, the case with a parallel work being developed in another dissertation, where the focus is the use of an object-based storage system, Ceph.

4.2.1 Requirements

Btrfs is a modern file system based on the concept of copy-on-write (COW): it is capable of creating lightweight copies of filesystem structures – files, directories, volumes. We already detailed the importance of this trait in Section 3.2.5. Therefore, a mandatory step was to investigate Btrfs-provided tools that could help us to achieve our primary goal: being able to move information ("base" iMIs and their snapshots) around, from "master" to "secondary" nodes, while consuming the minimum bandwidth – one must not forget that, while a typical Linux iMI is less than 10 GB, a Windows 10 iMI is circa 40 GB. We found that Btrfs has incremental backup capabilities, and therefore we set out to explore those.

So, Btrfs's incremental backup capabilities are a good starting point; however, their purpose is to help in data transfer; but that is not enough. Preservation of consistency of the iMIs is also a concern, as one has to assure that iMIs cached in the RCSs are up-to-date, and when a new iMI is created its distribution will start "immediately", to ensure high availability in case of faults. Moreover, the locality of the data should be taken into account, since data transfer endpoints may be located within the same data centre, or at, e.g., geographically disperse regions in the world. Bandwidth use and data encryption become essential, requiring the study of compression algorithms and encryption techniques.

Btrfs Incremental Backup feature A first step is trying to understand the most efficient way to transfer this unique kind of data (i.e. an iMI). Given the fact that we are working with a file system with snapshots capabilities, we want to take advantage of this functionality and minimise the amount of data roaming the network.

Btrfs has a set of userspace utilities to manage Btrfs filesystems, called `btrfs-progs`; these include a pair of commands, `btrfs send` [34], and `btrfs receive` [33], that provide the capability to transport data via a stream and apply differences from/to a remote filesystem. The send command simplifies the process of generating a stream of instructions that describe changes between two subvolume snapshots. Also available is the ability to use an incremental mode, where given a parent snapshot that is available in both the send and receive sides, only the small delta between snapshots (e.x. V2 and

V2-1 in Figure 4.2) is going to integrate the stream. This feature considerably reduces the amount of information that needs to be transferred to reconstruct a snapshot in the receiving end. The send-side operations occur in-kernel, beginning by determining differences within subvolumes and, based on those differences, the kernel generates a set of instructions in a custom-formulated stream.

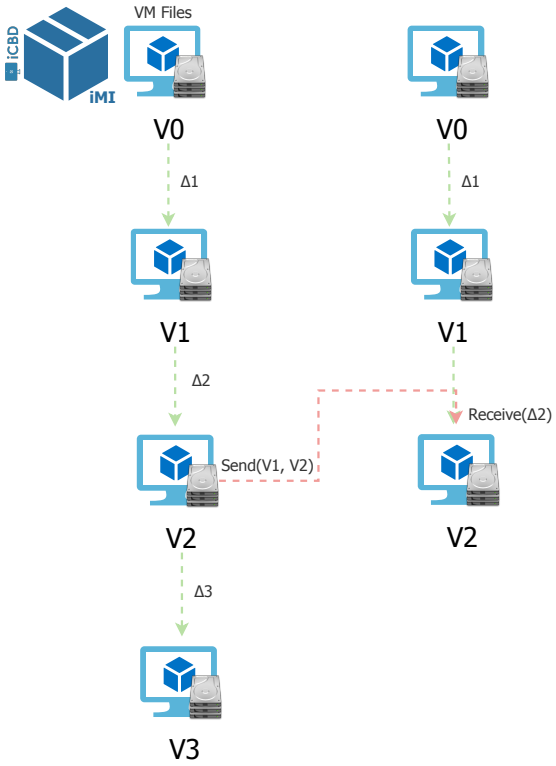

Figure 4.2: iCBD iMI Snapshots Structure

On the remote end, the receiving command accepts the stream generated by the send and uses that data to recreate one or more snapshots. Contrary to the send command, receive executes in user space, replaying the instructions in the stream one by one; the set of instructions includes the most relevant calls found in a Virtual File System, such as `create()`, `mkdir()`, `link()`, `symlink()`, `rename()`, `unlink()`, and `write()`, along with others [32].

4.2.2 System Overview

Since the replication module interacts with different tools and utilities, e.g., bash scripts and command line tools and some OS calls, we think that the most suitable approach was to create a Python distributed middleware layer using a master-replica paradigm.

Python, as a programming language, enjoys some idiosyncrasies, functioning as an object-oriented language, possessing an extensive standard library and enjoying a big community delivering packages with a wide range of functionality; all these facts contribute

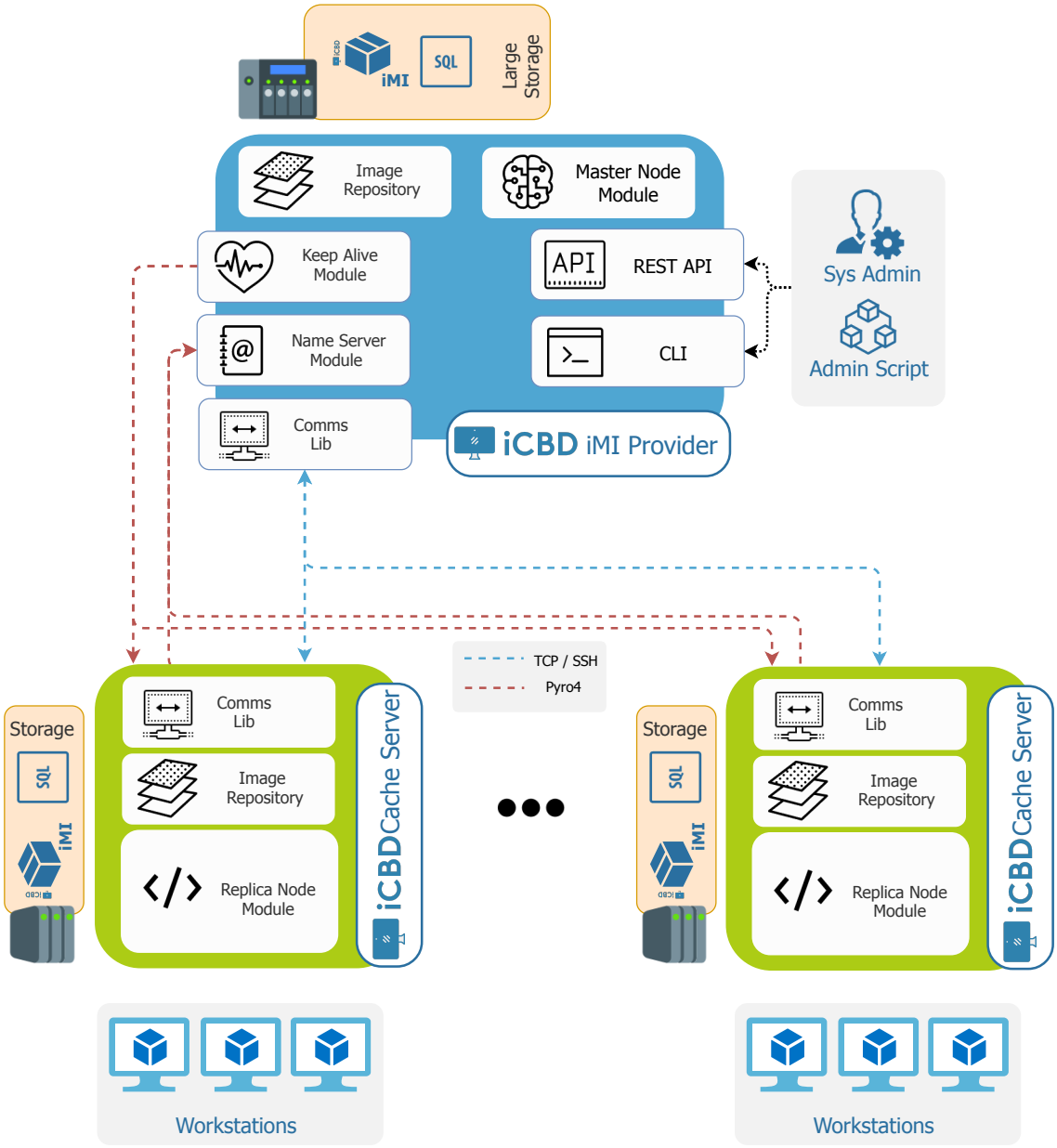

Figure 4.3: iCBD Replication Modules and Communications

to make it the most appropriate programming language to bind everything together in our iCBD platform.

Our middleware is composed of several modules and includes all the necessary functionalities that allow a node to behave as a Master or as a Replica, where each maintains its individual Image Repository. A number of libraries were also developed: to interface with some tools, such as Btrfs send / receive and with SSH, to build wrappers for compression algorithms, and even to provide a REST API.

An overview with a small description of the functionality of each component of the RCS system follows:

Master Node This node acts both as a controller to the replication system and an interface to interact with a client whether through a CLI or REST API. This node shall reside on an iCBD Administration machine since it must deliver changes made to iMIs to all replicas. It is also responsible for communicating with the Name Server to gather information about which nodes are present on the platform, or what is their status; even instruct the replicate nodes to execute certain operations.

Replica Node Its main task is to maintain the list of subscribed iMIs that are not up-to-date, receive the updates as soon as the Master Node sends them, and store them locally. Upon request, the Replica Node can deliver the list of iMIs that are locally stored and their version numbers, and subscribe (or unsubscribe) for new IMIs.

Name Server This service lives in the same machine as the Master Node, holds records about nodes in the replication system, in a phone book type simple database. Nodes register themselves in the name server during the startup process (and leave on shutdown) and can, at any moment, query about the location of other nodes.

Keep Alive It's a Master Node thread that periodically checks if Replica Nodes are still operating correctly. When it identifies that a replica node is no longer responding, it immediately sends a request to the for the removal of the dead node from the Name Server's directory.

Image Repository This module is a custom-made data structure that holds a (large) set of iCBD Snapshot objects in a `key:value` store. In addition to storing these objects, it has an interface that provides quick answers to queries that ask which iMIs are stored in the node's repository. Every node in the platform (i.e. Master or Replica) must necessarily hold one repository.

All the previously described components interact with a number of "objects", so it is not a surprise that a sizeable amount of code that we have created has been packed into a number of libraries:

Btrfs Lib The Btrfs library holds two classes: the first one, called `Btrfs Tool`, is a Python wrapper for `btrfs-progs`, described in section 4.2.1.; the other class, designated

`BtrfsFsCheck` implements functions that validate if a given path transverses a Btrfs filesystem and, if it does, verifies if in that path a subvolume also exists. Additionally, a method is provided to discover all snapshots in a directory.

Compression Lib Since multiple compression algorithms are used, it makes sense to create a library that encapsulates multiple wrappers, one for each algorithm; these wrappers only contain code that provides compression and decompression of data streams, no other operations are present.

Serialiser Lib Some communication operations between nodes require the transmission of objects; therefore a library containing serialisation and deserialisation methods for those objects has been implemented.

SSH Lib This library implements a wrapper for the SSH command, allowing the creation of tunnels so that data can be funnelled through a secure connection between nodes.

REST API Lib To comply with one of the objectives of the replication module, a REST API should be provided. That is precisely what this library does, providing the endpoints to interact with the system, and enabling an easy way to expand that interaction to other platform modules.

The libraries we created to implemented the common set of functionalities required for the replication module, do require more libraries, implemented by the Python community. These are necessary to handle communication between nodes, algorithms for data compression, and secure data transmission; the most relevant ones are described through the remaining text in this section.

4.2.3 Communications between nodes

To coordinate the multiple modules and their activities, an abstraction of a network-shared communication channel must be supported. The remote procedure call (RPC) was chosen to support the inter-process communication, allowing a procedure running on a system to invoke an operation in a process running in a different location, most likely on a remote system.

As seen in figure 4.3, multiple processes are running in different machines any given time, and these processes need to continually send and receive information: operations that must be executed, metadata updates, monitoring if a process is compliant with its expected behaviour or is in a faulty state, etc. Managing nodes is just the right case for the Pyro 4 library, one that gives an holistic view of the system and allows triggering operations in a node.

Pyro4 Library Pyro4 [29] is a library that enables the development of Python applications where objects can talk to each other over the network through the use of RPCs. It is

```
[[source]]
url = "https://pypi.python.org/simple"
verify_ssl = true
name = "pypi"

[packages]
"pyro4" = "==4.62"
serpent = "==1.23"
"py-lz4framed" = "==0.10.0"
python-snappy = "==0.5.1"
Sphinx = "==1.6.6"
sphinx_rtd_theme = "==0.2.4"
flask = "==0.12.2"
Flask-Jsonpify = "==1.5.0"
Flask-RESTful = "==0.3.6"

[dev-packages]

[requires]
python_version = "3.6"
```

Listing 1: Dependences of the iCBD-Replication module bundled in a Pipfile

designed to be very easy to use and integrate in projects while having a considerable flexibility. This library can be imagined as a glue that easily integrates various components of a heterogeneous system.

Some Pyro4 core features are used in the iCBD replication module including (but not limited to):

Proxy This object acts as a substitute for the real one, intercepting the calls to a given method of that object. Then, through the Pyro library, the call is sent to the actual object - one that will probably reside in a remote machine – and will return the resulting data (very useful, considering that the function that performs the call does not need to know if it is dealing with a local or remote object).

Pyro object A Pyro object is a regular Python object that goes through a registration process with Pyro in order to facilitate remote access to it. Objects are written just as any other piece of code, but the fact that Pyro knows their existence allows other programs to place calls.

Pyro daemon This component listens for remote method calls done to a proxy, dispatches them to the real object, collects the result of the call, and returns it to the caller.

Name Server It keeps track of the object's actual locations in the network so that they can move around freely and transparently. Similarly to a yellow-pages book, provides a way to lookup objects based on metadata tags.

Automatic reconnecting Provides an automatic retry mechanism to handle the fault that arises when a client (in our case a Replica Node) becomes disconnected from the server (Master Node) as a result of a server crash or communications error.

Secure communication Pyro, in itself, does not encrypt by default the data it sends over the network. Still, Pyro RPCs communications can travel over a secure network (VPN, SSL/SSH tunnel) where the encryption is performed outside the library. Alternatively, it is also possible to enable SSL/TLS in the Pyro library itself, securing all communications with custom cert files, private keys, and passwords.

Serialisation Pyro supports the transformation of objects into streams of bytes that flow over the network to a receiver that reconstructs them back into the original format. This process is essential to ensure the delivery of all data that remote method calls receive as arguments, as well as the corresponding responses they return.

TCP Sockets and Secure Shell Protocol (SSH) Coordination of system activities is only a part of the work that "loads" the network; the other, and more important part, is carrying large volumes of data (as a result of transferring GB-sized iMIs) to perform the replication tasks that keep VM images consistent across RCS nodes. The Pyro4 library allows secure communications, but only for method calls (arguments and responses) within replication nodes.

The delivery process of iMIs throughout nodes follows one out of two principles: in the first scenario, we consider the case where the iMI does not leave the same trusted local network (i.e. communications within the walls of one organisation); the second covers the transport of data across external networks, including the Internet.

When discussing intranet environments, it's safe to assume that there are some security measures already in place (e.g. VLANs) so, in this regard, we assume that data security is already catered for. That allows us to use a simple Stream Socket [21], which does provide a connection-oriented flow of data with error detection, in our case implemented on top of TCP. This option, when paired with non-encrypted communications, delivers the best performance possible for the transfer of an iMI.

In the second case, which involves data travelling through external networks, an extension of the previous solution is employed: we used the same type of socket, but transport data through an SSH tunnel deployed between nodes.

This, in addition to solving the issue of ensuring data security in the transferal process, has the benefit of being a modular solution that allows future changes in the way data is encrypted without needing significant modifications to the code base. But, while we do not believe that this is a perfect security model, and there is room for improvement, we will not pursue that avenue because this is not the focal point of this work. However, the current solution does provide the level of security we deemed enough to conduct functional tests linking geographically separated data centres.

Data Compression As noted in the section where requirements were discussed, our implementation should aim for a reduction in the bandwidth used by operations in the iCBD platform, and that includes the replication of iMIs. Part of it is already addressed in the Storage Layer since the images, by default, are transparently stored in Btrfs compressed with zlib [31]. However, the replication process using the Btrfs send and receive, as explained in section 4.2.1, does not send an iMI itself, but an "instruction" stream (which includes the data to be transferred) which, in itself, is an excellent candidate to compression.

Given the design of the send/receive features, is unthinkable to hold in memory all the stream waiting to be compressed, or store that information in a file, compress it, and then send it, without creating a huge bottleneck and introducing a delay on the replication process. To expedite the process of transmitting a compressed stream immediately, only compression algorithms that provide a framing format (i.e. allow decompressing without having to hold the entire stream in memory) were chosen.

In this work, we included three algorithms that had the desired the framing format feature and were found to be widely used, maintaining a modular code base in order to support the future inclusion of other algorithms.

LZ4 is a lossless data compression algorithm that belongs to the LZ77 family [62] of byte-oriented compression schemes and is focused on maximizing compression and decompression speeds. Its reference implementation is in C and was initially implemented by Yann Collet. There are ports and bindings in various languages such as Java, C#, Python, and Go, among others. In this work, we use a multithreaded Python version developed by Iotic Labs called *py-lz4framed* [28].

zlib is a widely used, and kind of a *de facto* standard, library of lossless data compression that uses an abstraction of the DEFLATE compression algorithm (also a variation of LZ77). The algorithm version written by Jean-loup Gailly and Mark Adler provides good compression on a wide variety of data sets and environments with the minimal use of system resources [1]. Written in C, it can be found in a wide diversity of platforms: in the Linux Kernel in multiple modules, and in multimedia libraries, databases, version control systems, and others. In the replication module, we use the zlib library [17] included in Python, which then provides an interface to the zlib C library.

Snappy is a compression decompression library, created by Google [19], that contrary to other algorithms, strives for very high speeds at reasonable compression rates, not maximum compression. The library is written in C++ but has several bindings for the most popular languages. In order to interface with Python, we used the Python binding [44] for the snappy C++ compression library provided by Andrés Moreira.

4.2.4 Name Server

In a distributed systems environment, nodes need to know how to communicate with each other, uniquely identify themselves and be able to refer to their locations. The mechanism that addresses this problem is commonly referred to as Naming [54].

The iCBD Replication module implements a Naming Server, where each node is identified by a tuple with three elements: Node Name, Node URI, and Tag. Tuples are registered in the name server and operation include locating a node by its name, or retrieving a set of nodes that are marked with the same tag.

The name server is a module that consists of a continuously running thread and a local SQLite database. It uses the aforementioned Pyro4 Name Server, but instead of being launched from a console, it leverages the "launch on your code"feature provided by the library to seamlessly integrate with the remaining modules and starts up together with other modules of the Master Node.

Consider the scenario where a node wants to contact another node and does not have its location: a request (which includes the name of the target node) must be made to the name server, expecting in return a URI for the target node. If the requesting node already knows the location (IP and Port) of the name server, it directly addresses it; however, if the node does not know how to contact the name server, it resorts to a simple UDP network broadcast to locate it.

Methods in the Name Server API The Pyro Name Server presents an extensive API but, for the purpose of our work, only the subset presented below is used:

`locateNS()` Get a proxy for a name server somewhere in the network.

`register()` Enrol an object in the name server, associating the URI with a logical name.

`list()` List all objects registered in the name server; the results will be filtered if a prefix is provided.

`lookup()` Looks for a single name registration and returns the corresponding URI.

`remove()` Removes an entry, created by registering an object with the exact given name, from the name server.

4.2.5 Image Repository

Each Replica Node in the platform can subscribe to an independent set of iMIs that will be replicated to its local storage, with the Master Node holding the entire collection. To represent this relation between nodes and to facilitate the process of knowing which image is present in each node, we implemented an Image Repository.

The Image Repository sub-module is present in every node (Master and Replicas) and plays a central role in the replication process, not only because it acts as a backbone for

```
class NameServer(Thread):

        def __init__(self, config):
                # Thread configs
        Thread.__init__(self, name="Thread-NameServer")
        self.ns_ip = config["ip"]
        self.ns_port = config["port"]
        self.ns_db = config["dbFile"]

        # Pyro name server
        self.nameserver_uri,
        self.nameserver_daemon,
        self.broadcast_server = Pyro4.naming.startNS(host=self.ns_ip,
                                                     port=self.ns_port,
                                                     storage=self.ns_db)

        self.nameserver_daemon.combine(self.broadcast_server)


        def run(self):
        # This is triggered in the thread.start() call
        try:
            self.nameserver_daemon.requestLoop()
        finally:
            # clean up
            self.broadcast_server.close()
            self.nameserver_daemon.close()
```

Listing 2: Starting procedure of a Name Server

the subscription of images, but also because it tracks all iMI versions available in the platform in a way similar to that of a versioning system. As described before, the iCBD platform stores multiple versions of one iMI as snapshots, that materialise as directories in the local filesystem. The information stored by the Image Repository is backed in persistent storage in the form of an SQLite database, similarly to what happens in the Name Server.

The interface that the Image Repository provides is very simple, and consists of a handful of mutator methods (get and set functions) that populate one main data structure, based on the Python built-in Dictionary, thus providing an unordered set of `key:value` pairs, where the `key` is the name of the iMI and the `value` is a `List` of several *icbdSnapshot* objects, one for each version.

iCBD Snapshot Object Structure (iMI) Inside the replication module the iMI, as presented in Section 3.2.1, is treated as a first-class citizen, being represented by the class

icbdSnapshot. This object stores the relevant metadata and properties of an iMI that are essential to unequivocally identify the multiple images present in the system unequivocally; but it does not hold actual data.

In that sense, from this lightweight object, we can get hold of: the name of the iMI, its version number, the full path to the VM files in the filesystem, the location of the boot package associated with that particular version, and the configuration file for the iSCSI target.

Since the data stored in this object is appended at creation and is immutable, the object only provides get functions to retrieve its values. Given that all the relevant data is stored locally in a node (i.e. the actual VM file, the boot package, and the iSCSI configuration files) the *icbdSnapshot* object only needs to maintain paths to that data with regard to the local filesystem, leading to a clean and straightforward interface that can be seen in Listing 3.

```
class icbdSnapshot(object):
    def __init__(self, mount_point: str,
                 image_name: str,
                 snapshot_number: str,
                 creation_time: float,
                 icbd_boot_package_path: str,
                 iscsi_target_folder: str):

        # Path to the FS where the VM Files are stored
        self.mount_point = mount_point
        # Name of the iMI
        self.image_name = image_name
        # Version number
        self.snapshot_number = snapshot_number
        # Moment the iMI was added to the platform
        self.creation_time = creation_time

        # iMI Boot Package
        self.icbd_boot_package_path = icbd_boot_package_path

        # iMI iSCSI target
        self.iscsi_target_folder = iscsi_target_folder
        self.iscsi_target_name = "{}-{}_flat.conf".format(self.image_name,
                self.snapshot_number)
```

Listing 3: Example of the information stored in the *icbdSnapshot* object.

4.2.6 Master Node

The architecture and design of the replication process dictate that managing the subscription of iMIs, disseminating new versions and, finally, providing an interface to interact

with these services is fundamental.

Given those requirements, the Master Node resides in an iCBD Administration Machine allowing it to hold a global view of all components of the system and a holistic view of the state of the platform (i.e., identify all the nodes present in the platform, list all iMIs in catalogue, maintain a list of all relations between iMIs and nodes - the subscriptions).

And, furthermore, the Master Node also acts as a gateway, providing two methods to interface with the platform: a Command Line Interface (CLI) and an REST API.

Besides having a central role in managing and overseeing the different aspects of the replication platform architecture, the Master Node holds the responsibility of sending new versions of an iMI as soon as they are created, using the Btrfs Incremental Backup feature (in this case, the send operation).

Sending a Snapshot to a Replica Node (Cache Server) The overall process follows these steps: when the task of administrating an iMI is finished, the Master Node is notified that a new version (of that iMI) is available; a new entry for that version is then created and stored in the local Image Repository; afterwards, the main replication procedure starts.

The replication procedure proceeds as follows: first, the list of nodes that have subscribed to that iMI is gathered; then one obtains the last version available in the Replica's Image Repository and that "value" will be used to determine if the new version can be sent immediately, or if it is necessary to transfer some intermediate versions. In any case, only the differences between versions are sent, not the whole iMI. Assuming that only the last delta (the most recent changes) will be shipped, it is now necessary to decide whether the transfer will occur using a plain or encrypted communication channel and/or whether some compression algorithm will be applied to the data before being sent. This information is also conveyed to the receiving side so that it is ready to accept the data.

Only after completion of the above-described process, will the data transfer be carried out, using the btrfsLib, compressionLib and sshLib libraries. The process of receiving an IMI can be found in Section 4.2.7.

Keep Alive When the Pyro4 platform was presented, we mentioned that this library allows the automatic reconnection of clients to the Master Node. However, the opposite is not provided, i.e., in case of failure of a Replica Node, the Name Server is not automatically informed of that failure; in fact, this event would only be noticed when a connection with that node was attempted and failed. Since we want the Master Node view to be always consistent with the status of the platform, it is essential that the Name Server is always up-to-date; so, we implemented this module.

The Keep Alive module is started along with the Master Node and runs its own thread, making the execution flow independent of any other sub-module (as with the Name Server). Its main task is to verify the activity status of all Replica Nodes indexed by the Name Server, in a quite simple manner: it gathers the list of all Replica Nodes and, with

a period of about ten seconds, each node is sent a message and must deliver a response. If there is no response, two further attempts to contact that node are performed, and if those fail too, it is assumed that something wrong happened and that node is declared inactive. Then, the Name Server is notified that a node entered a failed state, and should be excluded from the "database".

4.2.6.1 CLI Interface

One of the methods to interact with the iCBD replication platform is through a Command Line Interface provided by the Master Node; the CLI is nothing more than a program that is started on a terminal window and recognises a simple vocabulary that includes a few verbs (command words), arguments and options, executes them, and returns its result.

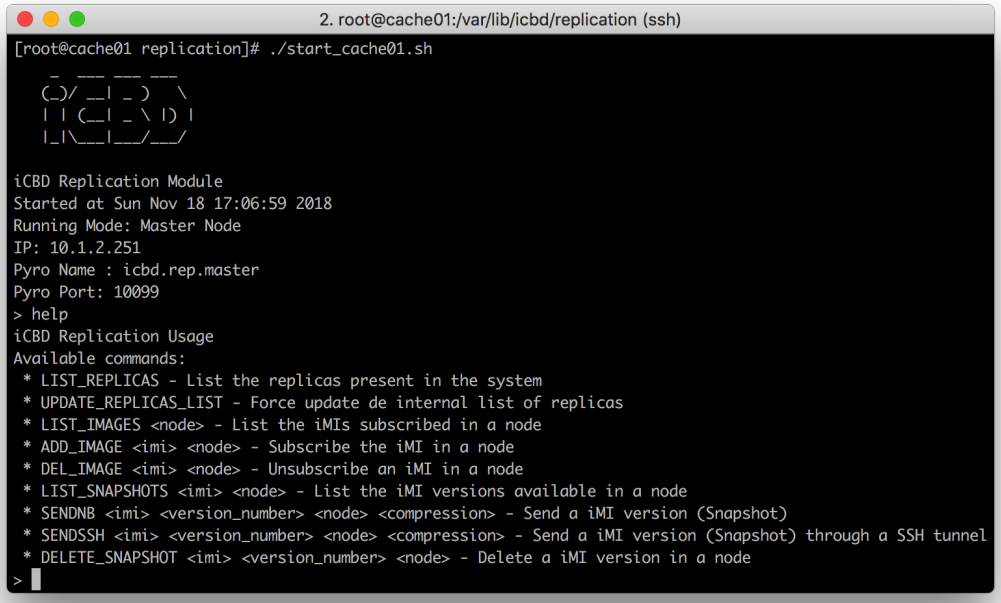

Figure 4.4: iCBD Replication Module help output

Following we demonstrate the functions provided by this interface and the effects produced on the platform:

`List Replicas` - At any time it is possible to consult which nodes are registered in the platform, this command allows to list the replicas registered in the Name Server and indicates which URI is used to make a connection.

`Force Update Name Server` - As previously explained, when a replica node stops responding, that node is deleted from the records held by the Name Server. However, this process may not be immediate because of the timeout implemented. So this

feature forces an update to the Name Server list by contacting all the nodes and determining if they are working correctly.

`List Subscribed iMIs` - This command receives as a parameter a replica node and shows a list of which iMIs that node is interested in receiving. It should be noted that a replica node may not subscribe to an iMI but still make it available to its client (was previously subscribed and the data was not deleted), it just will not be receiving new versions as they are being produced.

`Subscribe iMI` - Like the previous operation, this receives some arguments (a replica node and an iMI), then, registers the interest of the replica node in a given iMI. After this procedure, this node will be able to receive versions of this iMI.

`Unsubscribe iMI` - When a replica node ceases to be interested in a given image present on the platform, this operation marks that new versions of the iMI should not be transferred to that node. However, all versions that have already been sent persist on the node and in order to be deleted them an appropriated operation must be used.

`List iMI available versions` - It is usual for a given node to contain multiple versions of an iMI (for example, the Master Node contains all versions of all iMIs that can be distributed). Thus, given an iMI, this operation allows the listing of which versions a node stores.

`Send iMI Snapshot` - Possibly the most significant operation in this module. It is responsible for sending the respective versions of an iMI to the intended node. Always verifying which versions are present on the target node since only the differences between versions should be sent. This command also supports the application of a compression algorithm from those provided by the platform, since it may be the case of transferring a version for the first time with a very significant data volume, or the changes between versions possess a high compression rate thus making the compression of these data advantageous.

`Send iMI Snapshot Secure Connection (SSH)` - This functionality is similar to the one presented above. In particular case, they share a large portion of the code, because they carry out the same operations. They only differ in the method of sending the data, which in this instance are transmitted in an encrypted fashion through an SSH tunnel. This functionality is sure to add some overhead to the transfer process, but the encryption of the data is essential in situations where the nodes are in separate networks, where there is no control wheresoever to the data security.

`Delete iMI version` - Finally, it is necessary to provide a way to delete versions of a given IMI in a node. Either because no longer is desired to make an IMI available or for reasons of proper management the storage space on a replica node. It is important

to note that because of the way different versions of iMIs are stored in the platform, deleting older versions may not result in a space release equal to the size of the full iMI, since newer versions probably will still need this data.

4.2.6.2 REST API

In order to complement the Comand-Line Interface previously presented and creating a more straightforward, more ubiquitous way of interacting with the replication platform, a Rest API has been introduced. Aiming to provide the same functionalities as the CLI but trying to create the roots of a component that deals well with platform scaling and the introduction of new features or components.

In order to integrate this component with the remaining replication platform, we employ one of the most used frameworks for creating web platforms in Python, the Flask micro-framework.

`GET /api/replicas` - list all Replicas in the system

`GET /api/replicas/{replica}/imis` - list of the iMis present in a Replica

`GET /api/replicas/{replica}/imis/subscribe/{imi}` - Replica subscribe to iMI

`GET /api/replicas/{replica}/imis/unsubscribe/{imi}` - Replica unsubscribe to iMI

`GET /api/replicas/{replica}/imis/{imi}/versions` - list the versions of iMI present in Replica

`GET /api/replicas/{replica}/imis/{imi}/versions/{version}/delete` - delete a version of iMI in Replica

`GET /api/master/imis/` - list the iMIs present in Master

`GET /api/master/imis/{imi}/versions` - list versions of iMIs present in Master

`GET /api/master/send?imi={imi}&version={version}&replica={replica}` - send version of iMI to Replica

Listing 4: iCBD-Replication REST API Route Mapping

Flask Started as an April's Fool's Day joke to become the second most popular web development frameworks. Flask is a Python micro web framework designed with simplicity in mind, enabling quick deployment of applications and at the same time providing the ability to scale for complex environments. Such library enables the development of web applications without having to worry with more low-level aspects like network protocols and thread management. This framework began its development in 2010 by Armin Ronacher as a wrapper of two of his libraries: Werkzeug and Jinja.

Our use of this framework has focused only on its ability to quickly provide an environment for creating a REST API and connecting it with the rest of the replication

platform. Next, we present the endpoints through which it is possible to interact with a master node:

4.2.7 Replica Node

The replica node introduces a lightweight module that is responsible for facilitating the process of transferring and updating the iMIs that are closest to the clients. This module responds to the question of how to simplify the process of making available iMIs closer to the client. Complementing the remaining modules of the iCBD platform, here the focus is on implementing the logic of receiving snapshots, as well as providing a way to manage which iMIs are subscribed with their different versions. With these features, we have all the components to build a cache server, something that will be explained in detail in the section 4.3.

Contrasting with the Master Node, it is not possible to interact directly with a replica; all operations will always be conducted through a Master who then is in charge of communicating with the Replica, asking him to perform those actions. This fact is more than an architectural design choice, this way we meet one of the requirements that define that a cache server should be as light as possible leaving the platform management element to a centralised location not needing to know anything concerning the overall platform state. However, there are parts in common. Similarly to the Master Node one of the components present is an Image Repository, which in this case, manages locally the iMIs and respective versions.

Receiving a Snapshot By far, this functionality is the reason for the existence of this module. The process of receiving a version of an iMI is complementary to the send process explained in Section 4.2.6 even though much more straightforward. As explained, the `send()` operation needs some computation resources since it has to calculate all the differences between versions to transfer, and after that step create a stream of operations that when executed recreate precisely the differences between versions. So, on the receiving side of this stream (in this case the Replica Node), the `receive()` operation only is only required to receive the stream, decode the operations to administer to the file system and execute them.

Documentation for all this project was generated using a tool called Sphinx [10] and can be consulted in Annex I.

4.3 Deploying an iCBD Platform with a Cache Server

Once the process of creating a mechanism to support the replication of iMIs has been completed, we have reached the stage where we tackle the problems of supporting a large

number of clients, such a: limited bandwidth between a central repository and the workstations; high latency and jitter derived from network congestion or configuration errors in network equipment. So we believe that bringing the iMIs closer to the workstations is the way to solve all these problems.

At the beginning of this work, there already existed in the DI infrastructure an iCBD installation, but it was limited to only one Virtual Machine very limited regarding its resources and a couple of clients (also VMs), not to mention that the version of the platform as badly outdated and missing important features since implemented. Those limitations originated from the fact that the platform was sharing the same already scarce infrastructure as the remaining research projects and services of the department

All of this changed with the acquisition of new equipment for the exclusive use of this project, which became the perfect circumstance to think about the general architecture of the services and carry out a fresh installation of the whole platform.

4.3.1 The infrastructure

At our disposal is a modern infrastructure, including a two-node cluster - HPE ProLiant DL380 Gen9, interconnected by an HPE Flexfabric 5700 managed switch with 10Gbps links and storage provided by an HPE MSA 2040 SAN Storage Disk Array. According to project requirements, the cache server should be supported by a low-cost machine, for this purpose, a desktop PC used in another research project was reconditioned, making it only a few modifications, such as, upgrading the amount of RAM and adding a second hard disk. We can also count on using two teaching laboratories of the Department of Computer Science (Labs. 110 and 112) where each one contains 15 modern workstations (CPU - Intel Core i3-7100 @ 3.90GHz; RAM - 8 GB; Gigabit Lan) able to support virtualisation, to serve as clients of the platform.

The general specifications of all these components can be found in the tables 4.1, 4.2 and 4.5.

CPU	2x Intel Xeon E5-2670 v3 @ 2.30GHz
Memory	128 GB
Controller Type	12Gb/s SAS
Ethernet	2 ports at 10 Gbps plus 4 ports at 1 Gbps
Hypervisor	VMware ESXi, 6.5.0, 4564106

Table 4.1: Specifications of one HP ProLiant DL380 Gen9 host

Controllers	2 MSA 2040 SAS
Total Capacity	7.2 TB
Disks	12 HP SAS 600GB 10k Rpms
Host interface	8 12Gb/sec SAS ports (4 per controller)
Ethernet	2 ports at 1 Gbps (1 per controller)

Table 4.2: Specifications of the HPE MSA 2040 SAN Storage

Network Since it is intended that the iCBD platform coexist with other services and integrate into existing networks, we categorised the types of traffic that will be produced. We have internal traffic restricted to the platform - resulting from the execution of the various iCBD services and communication between them; and external traffic - which is an effect of accesses to the Internet or the act sending iMIs to workstations in the laboratories.

In the cluster, we have a virtualisation architecture based on the VMware solution, which makes available, as part of the VSphere suite, a centralised interface for virtual machine networking called Distributed Switch, from which one can configure, VM connections to switches for the entire cluster.

DI Distributed VSwitch - This switch holds five port groups, two corresponding to the networks of the laboratories, and the rest are networks used by various services of the department with differentiating levels of access (students, teachers, public).

iCBD Distributed VSwitch - The networks present in this switch are only visible within the cluster, having as a role to differentiate the types of traffic produced: a management network; another for the administration of iMIs, and the remaining ones for traffic generated in tests of the platform and the Replication and Caching System.

Taking advantage of that two Distributed Switches were configured, each corresponding to one of the two traffic types listed above, in order to compartmentalise and not have platform traffic running through networks used daily by the department. Each PortGroup referenced in table 4.3 corresponds to a different network.

DI Distributed VSwitch		**iCBD Distributed VSwitch**	
Port Group	*VLAN*	*Port Group*	*VLAN*
DMZ-PRIV-DI	DMZ-PRIV-DI	iCBD-Adm-Net	VMWARE_VMOTION_DI
DMZ-PUB-DI	DMZ-PUB-DI	iCBD-Ceph	VMWARE_VMOTION_DI
LAB-DI-110	LAB-DI-110	iCBD-Net	VMWARE_VMOTION_DI
LAB-DI-112	LAB-DI-112	iCBD-Rep	VMWARE_FT_DI
R-ENSINO-PRIV-DI	R-ENSINO-PRIV-DI		

Table 4.3: Specifications of all Networks

4.3.2 Roles in the Platform

From the beginning of the project, the entire platform was supported by a single VM that centralised all the services. That was the case of the early mentioned deployment on the Nova University site and is still the case in the SolidNetworks site. But, taking advantage of the restructuring and the new installation, it was considered that several machines (physical and virtual) should be employed, each with the purpose of supporting part of the platform. Thus were created the different roles that we describe below:

iCBD-imgs One can consider the role of this machine as the core of the whole platform given its three main functions: it is in this machine that all the versions of all the iMIs available by the platform are stored; it is responsible for managing the iMI administration process described in the Section 3.2.3; and can make accessible the iMIs that has in storage directly to a workstation that is connected in one of the networks for that purpose.

With the creation of the replication system, one more responsibility is added to this machine, since it is here that will reside the Master Node (described in Section 4.2.6) that manages the entire replication service of iMIs. For the deployment of this role on the Nova University cluster we created a VM with 4 vCPUs, 32GB vRAM and 2 Hard Disks (OS and Data).

iCBD-rw This machine as the key role of making available temporary read/write space (in form of a directory by client) that the clients set up during the boot process. This way any changes that users make while using an IMI are saved but only while the workstation is in operation, once a session is closed the data is deleted.

One of the reasons for separating this functionality from the *iCBD-imgs* machine is to provide a way to visualise the weight of having multiple clients booting iMIs at the same time, not polluting the results with reads and writes for regular usage.

As the functionality of this machine is only the provision of iSCSI and NFS servers for accessing r/w space its specifications may be a bit more conservative with a VM boasting 4 vCPUs, 8GB vRAM and 2 Hard Disks (OS and Data).

iCBD-home We can consider this role as optional, even so, it makes perfect sense in an infrastructure where there is a high percentage of users of Linux iMIs since it is this machine that hosts the home directories of each specific user.

This role has a straightforward process: a user have a home dir under the `/home` directory of this machine, then this path is made available by an NFS server. During a workstation boot process of a Linux iMIs, that directory which is exported by NFS is mounted in the client `/home`, allowing the login of any user and the access to his personal files.

The specifications chosen for this machine are very similar to *iCBD-rw*, but with the particularity of the size of the disk responsible for the data storage is smaller.

	vCPU	vRam	Hard Disks	Interfaces	OS
iCBD-imgs	4	32 GB	16 GB + 600 GB	5	CentOS 7
iCBD-rw	4	8 GB	16 GB + 300 GB	4	CentOS 7
iCBD-home	4	8 GB	16 GB + 100 GB	4	CentOS 7
iCBD-cache	4	32 GB	16 GB + 600 GB	2	CentOS 7
iCBD-client	4	8 GB	Diskless	1	Network Boot

Table 4.4: Specifications of the virtual hardware of the iCBD machines

iCBD-cache This role draws on one of the features of *iCBD-imgs*, which is the provision of iMIs to workstations making its primary reason for existence. Like other roles, this machine can exist on physical or virtualised hardware, and aims to make the iMIs available as close to the workstations as possible, in order to overcome the challenges stated at the beginning of the chapter, such as the high latency between image server and workstations, and also the low transmission speeds that directly affect the users experience.

Like the case of *iCBD-imgs*, the introduction of RCS brings more features, so each Cache Server deployed in the platform will run one of the Replica nodes introduced in Section 4.2.7.

On the Nova University site, two types of cache servers were set up, the virtual type given its greater convenience in deploying and to confirm executability of this role in the platform; and later a physical one that was connected directly to one of the laboratories and was responsible for delivering the iMIs to its workstations. Both configurations can be seen in the tables 4.4 and 4.5, respectively.

iCBD-admin_iMI This is a special role created specifically for the administration of iMIs. The process of administering an iMI is triggered and managed by scripts within *iCBD-imgs*, but since administration always involves the need to run a VM with the iMI in question, this process can be triggered in two ways.

One entails the administration VM to operate on top of the machine responsible for the *iCBD-imgs* role, which can lead to the occurrence of nested virtualisation leading to a degrading of the performance. The second option is running that VM directly on the hypervisor of one of the cluster hosts not burdening any other role.

This VM is created dynamically by the administration process and works in a similar way to a client. With the difference that the changes made to the IMI will be maintained and used to create a new version of that iMI.

iCBD-client The platform client is the simplest of roles. Again, there are two cases, whether the client is a physical workstation or a VM. We can create a diskless virtual machine, and connect it to a network where an image server is located (*iCBD-imgs* or *iCBD-cache*) using a network boot process an iMI is made available to that VM. The same process can be used for a workstation, which when attached to a network

with similar characteristics, using PXE is possible to get an iMI. This boot process is largely described in section 3.2.2.

Still, there are other ways to boot an iMI on a workstation, such as, generate a bootable USB drives that connect the workstation to an image server, without using DHCP or PXE.

CPU	Intel Core i5-650 @ 3.20GHz
Memory	12 GB
Storage	2x SATA 80 GB + SATA 160 GB
Ethernet	2 ports at 1 Gbps
OS	CentOS 7 3.10.0-514

Table 4.5: Specifications of the Physical Cache Server

4.3.3 Installing iCBD Core Services

As can be perceived from the number of roles described above and the complexity of their interactions within the platform, the creation of all these virtual machines and the installation of all services is not a trivial task.

In order to accomplish this task, the collaboration of the Division of Computer Infrastructures from FCT NOVA was essential, since various configurations, especially involving networking issues, are they responsibility at the level of the entire faculty. For example, we want the platform to be available not only within the internal network to the cluster but also in the laboratories mentioned. For this, several devices were configured by this division, including several switches, as well as, restructuring the already in place network boot configurations conjugating with those of the project enabling the boot of iMIs in laboratories without jeopardising their normal operation.

The installation process has therefore taken three phases. In the first instance, it was a priority to create the conditions for the platform to be in full operation. Beginning by the creation of VMs for the roles - *iCBD-imgs*, *iCBD-rw*, and *iCBD-home*. Then proceeding to the installation of an OS, multiple services and making the necessary configurations for the iCBD platform to work in the internal network. In the end, some iMIs already created in SolidNetworks site were installed and prepared to operate in this new site. At this stage, only virtualised clients within the cluster were able to run.

The second stage entailed the creation of multiple virtualised Cache Servers. This procedure was expedited by the fact that *iCBD-imgs* was used as a base, performing a full clone and then removing the unnecessary functionalities. This step was crucial to create a test base for the replication modules, which up to this point had not been tested with production iMIs or in this type of environment.

The last step involved the installation of the physical Cache Server the closest possible to one of the laboratories, and for this, it was joined directly to the switch where the

workstations of this laboratory are also connected. From this point on, we have two laboratories with access to the iCBD platform, one connected to the physical Cache Server and the other connected to the *iCBD-imgs* virtual machine hosted in the cluster, so they both have the possibility of loading iMIs on the workstations present in those laboratories.

Of course, the process of creating a single VM that does platform support is a lengthy one and with some significant details to be addressed. In this sense, the Annex II provides a document that was produced at the same time as the installation process of the platform on the Nova University site proceeded, that document details with much clarity all the steps needed to create a VM, the installation and configuration process of the respective services required to execute the iCBD.

Btrfs bug found in CentOS 7 Kernel As a curiosity during the process of building the *iCBD-imgs* VM, we found a bug in a core component of the *coreutils* tool introduced in the kernel version `3.10.0-693.5.2` of CentOS 7. The `cp` command when used with option `--reflink=always` failed with the indication "failed to clone 'someFile': Operation not supported". This behaviour was reported to both CentOS and Red Hat and was eventually resolved. A more in-depth description of this process can be found in Annex III.

Chapter 5

Evaluation

The following chapter reports the experimental work performed in order to study both the executability and performance of the Replication and Caching Service. We describe the tests defined and performed following with some analysis of the results obtained, always trying to co-relate to the effects observed throughout the platform.

The chapter is divided into the following sections:

Section 5.1 .

Section 5.2 ..

Section 5.3 ..

Section 5.4 ..

5.1 Experimental Setup

To ensure the correct execution of all the tests we intended to carry out some adjustments to the iCBD platforms were necessary. To the infrastructure, we deployed two more virtual cache servers (adding to the physical cache server already in operation) with the entire iCBD solution including the RCS.

Also as discussed in the previous chapter, the Computer Science Department provided two laboratories (Lab 110 and Lab. 112) fully equipped with fifteen machines each (the general specifications of these machines can be seen in the table 5.1), with the objective of performing validation testing of the caching solution at a functional level and then the execution of performance tests.

There was also a need to make some changes to VMs that were already deployed, in order to make the laboratories fully functional. First, the VMs iCBD-rw and iCBD-home were connected to the networks of both laboratories configuring the interfaces with

CPU	Intel Core i3-7100 @ 3.90GHz
Memory	8 GB
Storage	275GB SSD
Ethernet	1 Gbps

Table 5.1: Specifications of the Laboratories Workstations

fixed IPs. Then, one interface of the iCBD-imgs VM was set up to be connected to the Lab. 110 network and also assigned a fixed IP, then some changes were performed in the iCBD configuration files to allow workstations connected to this network access to iMIs. Also, in the Physical Cache Server, one of the interfaces was configured in the network of Lab. 112, and similar configurations were necessary for this server to provide iMIs to the workstations of this Laboratory.

It is still important to note two aspects, first the Physical Cache Server and the Lab. 112 workstations are attached to the same managed switch, so the communications between them only cross this device. Whereas all communications between workstations of both laboratories and VMs deploy in the cluster travel through various equipment in the FCT NOVA network and therefore may suffer from adverse network conditions entirely out of our control. Second, all of these connections described above are ensured by Gigabit links, either at the level of the network equipment or by the interfaces (virtual or physical) of the servers. A simplistic schematic of all these connections can be found in Figure 5.1.

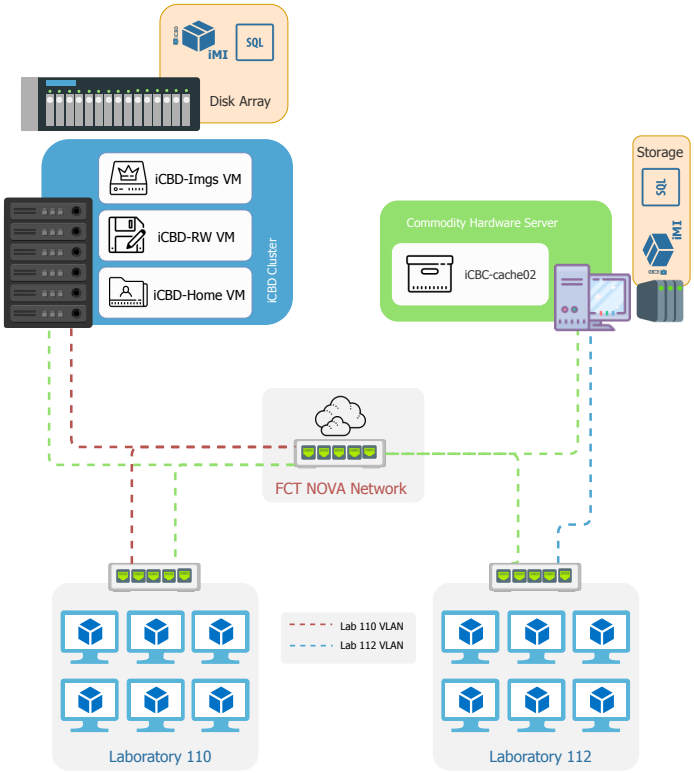

Figure 5.1: iCBD Nodes and Networking Setup

	FCT NOVA	SolidNetworks (Development)
Servers	2x HPE ProLiant DL380 Gen9	2 x HPE ProLiant DL380 Gen9
Switch	HPE Flexfabric 5700 jg898a	HPE Flexfabric 5700 jg898a
Disk Array	HPE MSA 2040 SAN Storage	N/A - (Storage on the Server)
Networking	10 Gbps (between servers)	10 Gbps (between servers)

Table 5.2: Physical infrastructure of the FCT NOVA and SolidNetworks sites

5.2 Metodology

The analysis of the RCS was divided into two distinct moments: a first part, which addressed the functional validation of both components of the RCS after being integrated into the iCBD platform, testing its execution correction; and a second part in which we focused on the performance of these components when faced with a production environment.

Functional validation Concerning the replication module, in order to test that this system is functional in a multi-node environment, the Master Node was started in the iCBD-imgs VM and three Replica Nodes, one in each Cache servers (virtual - iCBD-cache01 & iCBD-Cache03 and physical - iCBD-Cache02). Being observed that the Replica Node had registered on the Name Server as expected and that the communications between the Master Node and Replicas were done correctly.

Next, were carried out two types of test repeatedly. One focused on sending a complete version of an IMI that was not present in the Replicas' Image Repository, forcing the transportation of all the data that are part of this iMIs. We also took advantage of this moment to verify if after the sending process the local Image Repository reflected the addition of the new iMI. This scenario is likely to occur when its the first time that a replica subscribes to a new iMI.

The second type of test revolved about sending versions of iMIs that Replicas already possessed older ones in their repository. This is done to simulate the case where after the administration of an iMI this update is distributed by the Replicas with only the changed data being sent. At the end of each of these tests, we performed a calculation of an MD5 hash with the `md5sum` tool, in order to ensure that the received data was being reliably transferred.

In order to validate the correct functioning of Cache Servers we mainly focused on iCBD-cache02 (Physical Cache Server), mostly because it was connected directly to one of the laboratories allowing for immediate testing of a workstation iMI boot. Given the integration of the iCBD platform with the remaining network services and policies of FCT NOVA, a considerable iterative process of experimentation was required, constantly tuning some parameters of some iCBD services until we arrived at a fully functional configuration. In the end, it was confirmed that it is possible to boot the workstations with iMIs powered by the Cache Server.

Performance Benchmarking In this second phase of testing, the goal was to ascertain the performance of RCS in a production environment. In the case of the replication module, we make a comparison between multiple configurations of our solution (with or without compression and secure communications) and the `rsync` tool, measuring both the time spent on the transference process as well the amount of data transmitted between nodes.

While in the performance tests concerning the cache server, we measured the time spent on the boot process of a workstation. Comparing when the iMI was provided by the Cache Server or by the iCBD-imgs VM hosted at the cluster with more significant resources. For these test batteries, we consider the boot time the time elapsed from the beginning of a load of a kernel until the initialisation of all the services in userspace is finished, making use of the tool `systemd-analyze`.

Unless stated otherwise, all tests were executed five times removing the best and worst result. The final result is the average of the remaining values. The results of the work performed in these two fields are demonstrated in the following sections.

5.3 Replication Service Benchmark

This benchmark where we want to evaluate the performance of iMI replications within the iCBD platform we idealised two scenarios. One where the full transfer of an iMI to a Replica Node is performed, and the other where the Replica Node already possesses a version of that iMI and only wants to receive the differences (deltas) associated to the new version.

Regarding the IMI used in these tests, a Linux iMI was chosen that contains the Ubuntu 16.04 LTS distribution. We make use of two sequential versions (v1 and v2) of this iMI in which from one version to the other, during the administration process we applied all the updates proposed by the APT package handling utility. We know that the v1 of this iMI stores 38.60GB of data and with the resource to the tool `btrf-progs` we issued the command `btrfs filesystem df /path/` giving us the information that from the Btrfs point of view the differences between versions make up 4.45 GB.

For both test scenarios, we produced four sets of settings for the transfer:

Rsync Transfer of an iMI using the rsync tool [1] with the options `-r` (recurse into directories); `-t` (preserve modification times); `-p` (preserve permissions); `-l` (copy symlinks as symlinks) and `-u` (skip files that are newer on the receiver).

iCBD-Rep - I Use the iCBD-Replication platform to transfer the iMI using the standard python sockets and no compression.

[1] It should be noted that the rsync tool is not aware of a Btrfs subvolume, so the times presented only relate to data transmission, but more operations would be needed to make the iMI functional within the iCBD platform.

iCBD-Rep - II Apply the iCBD-Replication platform to transfer the iMI using the standard python sockets but compressing the data in a stream with the LZ4 algorithm.

iCBD-Rep - III Utilise the iCBD-Replication platform to transfer the iMI using an SSH connection to tunnel the data and not using any compression.

Sending a complete version of an iMI In this test that sends a complete iMI, that is all its 38.60GB of data; we can observe that the iCBD platform with its replication module performs similarly on the three sets of settings used. The results are shown in Table 5.3. There is a noticeable advantage of the rsync tool which takes less time to perform this process. We believe this happens because it detects that there is no data in the replica and so does not perform its usual block-by-block data comparison routines that are very time consuming and resource wasteful.

	Time	**Data Sent (MB)**
Rsync	12m23s	39543
iCBD-Rep - I	17m21s	38947
iCBD-Rep - II	20m35s	35572
iCBD-Rep - III	22m55s	39412

Table 5.3: Time spent and data transmitted on transferring a complete iMI from Master to Replica

It can also be observed that even without obtaining more favourable times, the use of compression indicates that the IMI is compressible around 8.6%, noting that not only the data of the IMI is compressed, as are the instructions generated by the send operation of the Btrfs. However, the is a caveat, because of the rsync's lack of understanding of the Btrfs internal structure, in the end of the process, the data replicated cannot be automatically made available to users, demanding further operations. While using our tool even though is showing that it takes more time to process the whole iMI, it is guaranteed that once the process is finished the transferred iMI is ready to be made accessible to the workstations.

Sending only the delta between version of an iMI We now present the results for the evaluation of the process of sending just the differences between the two versions of the chosen iMI. From the outset, one can see the drastic reduction of data transmitted by the network, as expected, only sending the data that was modified between v1 and v2 of the iMI. The results we obtained are shown in Table 5.4.

Our solution using Btrfs send receive operations shows as far superior to rsync's performance on the same data set. Even in the cases where the options of compression or cypher of the data were employed the results did not change dramatically.

	Time	Data Sent (MB)
Rsync	10m15s	5964
iCBD-Rep - I	1m47s	4864
iCBD-Rep - II	2m15s	4713
iCBD-Rep - III	2m55s	4902

Table 5.4: Time spent and data transmitted on transferring a delta between v1 and v2 of an iMI from Master to Replica

5.4 Cache Server Performance Benchmark

With this next benchmark we present, we want to evaluate the behaviour of introducing a Cache Server in a production environment. In order to obtain some interesting metrics, two types of tests were carried out, one scenario in which five workstations are sequentially booted, and the second experiments with the extreme case in which all the machines in a laboratory are connected at the same time, creating a situation known as boot storm. For each scenario, we measured the boot times of each workstation, and with a tool called netdata [55] monitored the state of the server at the time it is serving the iMis.

In addition to the two scenarios, there are more variables in play. All tests were performed with all links connected at Gigabit speed. Then every single one was repeated with links limited to 100 Mbps. For this to happen, on the tests referring to the Cache Server, the interface that serves the Lab. 112 was configured at this slower speed. As for the VM that serves the Lab. 110, iCBD-imgs, the type of virtual interface employed does not allow the change of its speed but using the Traffic Shaping feature of the Distributed VSwitch, we managed to limit the bandwidth of the entire PortGroup that refers to this laboratory, in the end reaching the goal.

Lastly, we need to explain the three types of boot tested in this section. The iMI chosen for these tests was the same as mentioned above, a Linux iMI with the Ubuntu 16.04 LTS distribution.

Linux Server VDI In this case, the computation is done on one of the cluster nodes in a diskless VM, simulating a traditional VDI environment, with the particularity of using the iCBD platform's network boot. This variant was born due to its versatility during the development of the platform, since we can connect this VM to any of the networks (internal or labs) and perform tests on its operation.

Linux iCBD Native When we talk about this variant, we are addressing the boot of an iMI in a physical workstation in which, in this case, the OS Linux runs natively on top of the hardware, with no kind of resource to virtualisation.

Linux iCBD VM This last variant presents an interesting feature since it makes use of two iMIs. In a first phase, it boots as described in the previous point, but with two

iMI		iCBD-imgs	iCBD-Cache02
Linux Server VDI	iSCSI	453.5 MB	454.3 MB
	NFS	703.0 MB	702.1 MB
Linux Client Native	iSCSI	456.3 MB	453.6 MB
	NFS	704.2 MB	703.8 MB
Linux Client VM	iSCSI	834.1 MB	836.8 MB
	NFS	950.5 MB	952.8 MB

Table 5.5: Total data received after booting, given each boot variant and for both iMI providers

differences. The booted iMI is of an openSUSE 42.2 [2], and this iMI has installed the VMware Workstation Player serving as a foundation to the next one. When the openSUSE iMI boot process ends, the iMI Ubuntu 16.04 is loaded by VMware Player and is then run on a virtual machine. All this without the user realising that he is using a virtual machine running on a different OS.

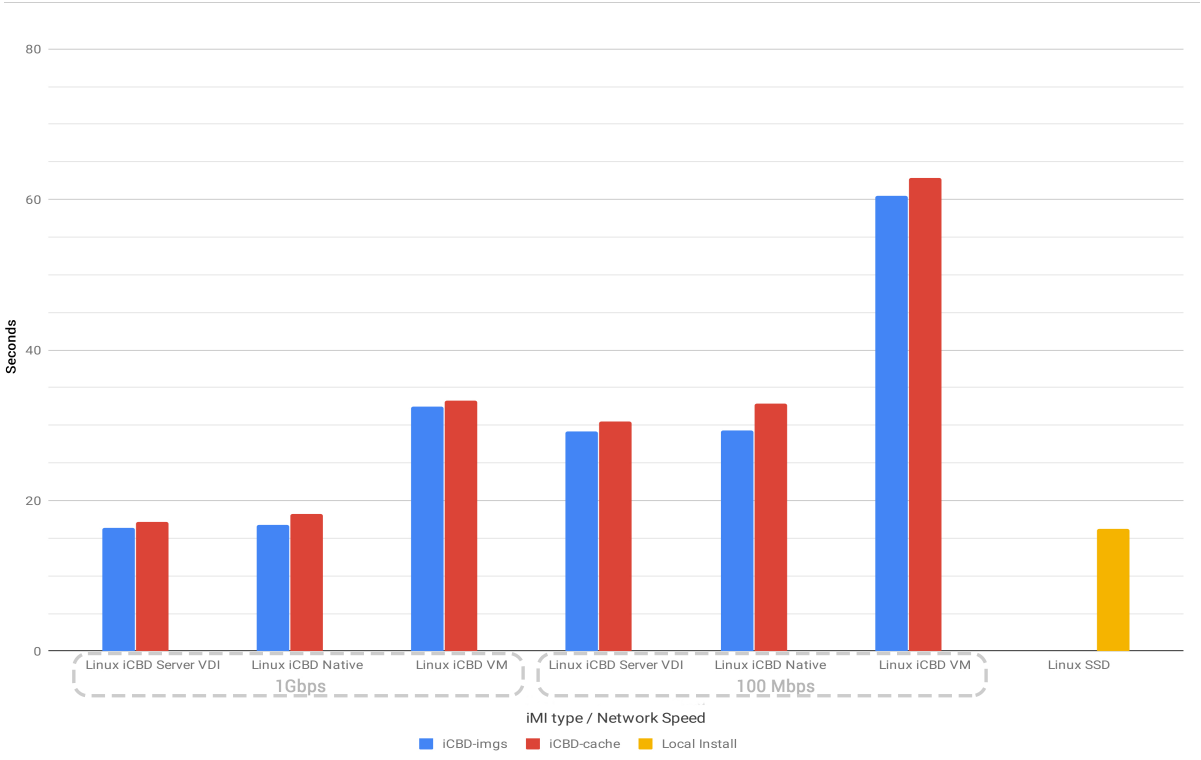

Figure 5.2: Mean Boot Time of five workstations using iSCSI (Sequential Boot Scenario), comparing iMI provider and network speed

Benchmark in a Boot Storm condition

[2] In this case, any other iMI could be used as a base (including natively run and then virtualised the same iMI), the version 42.2 of openSUSE was chosen because it was the iMI that exhibited the best stability in virtualising any other iMI, throughout the development process.

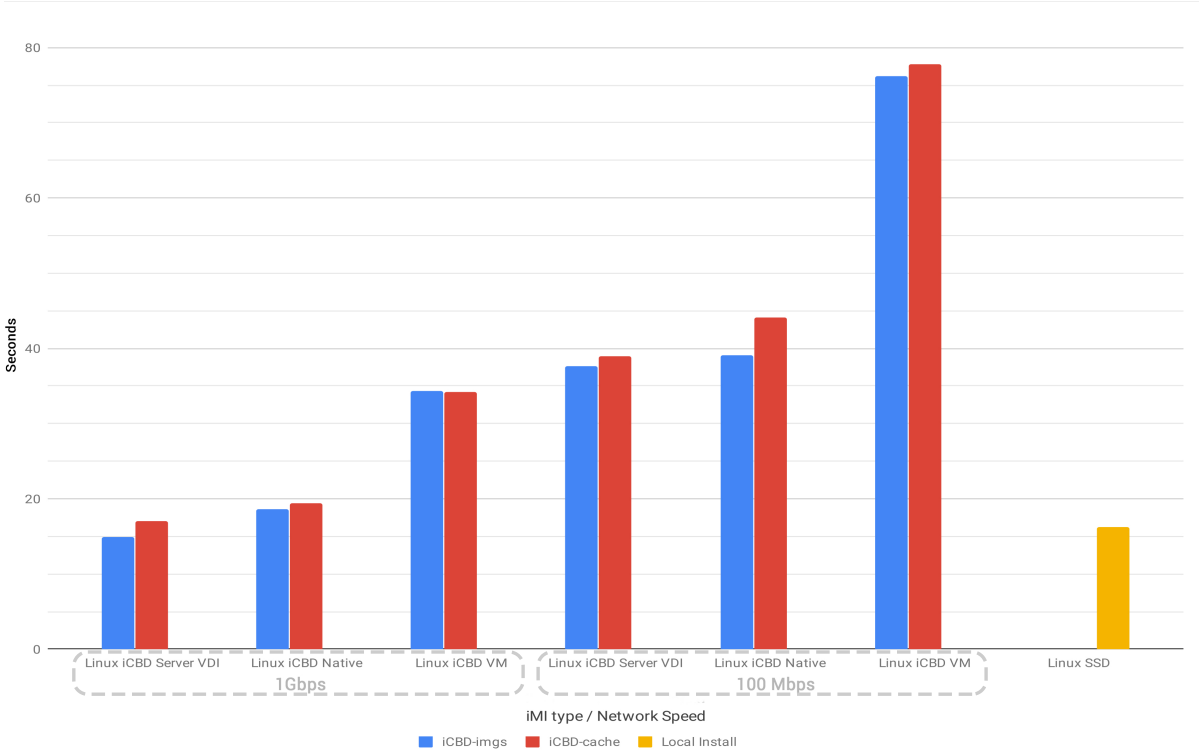

Figure 5.3: Mean Boot Time of five workstations using NFS (Sequential Boot Scenario), comparing iMI provider and network speed

iMI		**iCBD-imgs**	**iCBD-Cache02**
Linux iCBD Client Native	iSCSI	20.035 s	23.020 s
	NFS	23.248 s	28.156 s
Linux iCBD Client VM	iSCSI	42.627 s	52.952 s
	NFS	44.734 s	54.840 s

Table 5.6: Comparison of boot times in a boot storm situation in both providers (iCBD-imgs and iCBD-cache02)

iMI provider system load

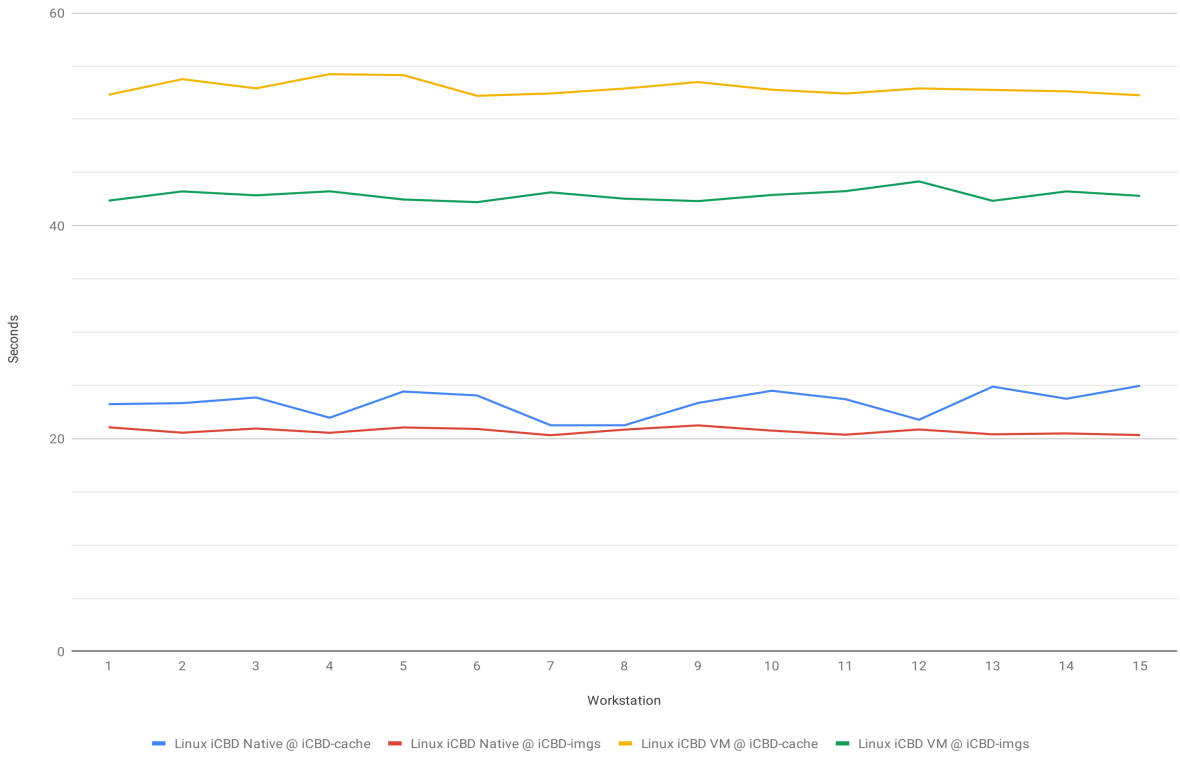

Figure 5.4: Boot Time of fifteen workstations simultaneously (Boot Storm Scenario) comparing iMI provider

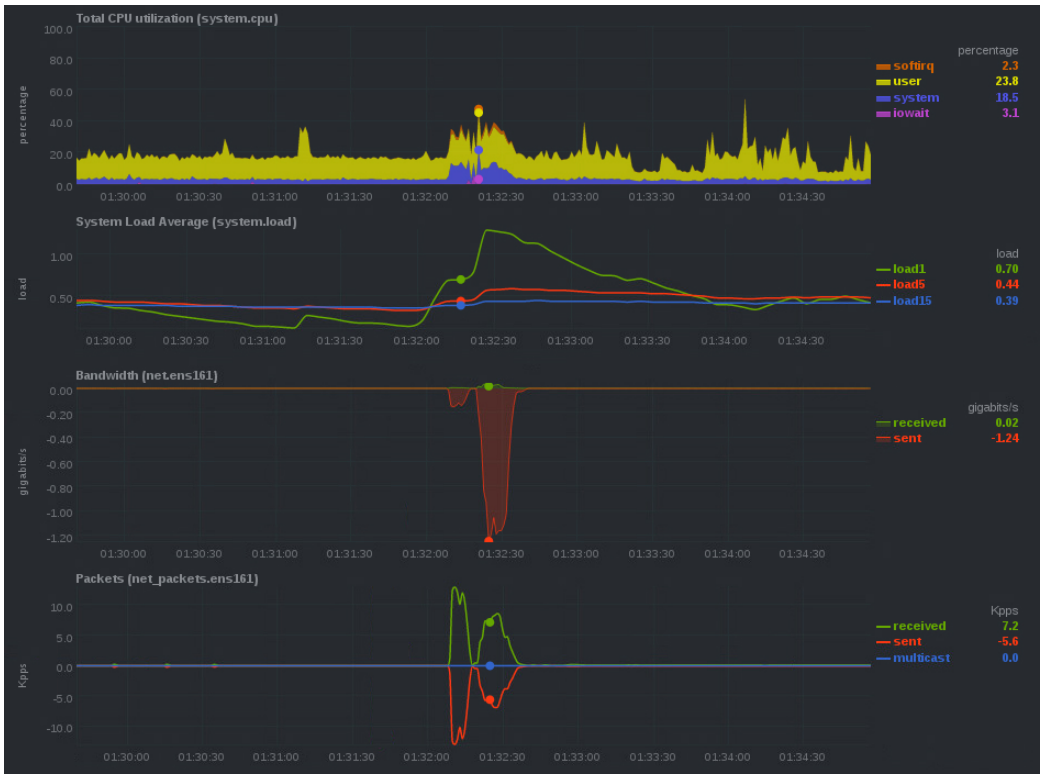

Figure 5.5: System metrics for iCBD-imgs on one run of the five workstations sequential boot scenario test

Figure 5.6: System metrics for iCBD-Cache02 on one run of the five workstations sequential boot scenario test

Figure 5.7: System metrics for one run on the iCBD-Cache02 in a boot storm scenario

Chapter 6

Conclusions & Future Work

6.1 Conclusions

The work developed by this thesis introduced two new components to the iCBD platform, a replication mechanism following a subscription model that allows the propagation of iMI by several replicas, and the concept of Cache Server that paves a way to support a large number of clients.

We show that the implementation of the replication system of iMIs in a Master - Replica model not only provides a process for sending iMIs in a geographically dispersed and multi-server environment, delivering network fault tolerance properties, but also a control layer that simplifies the entire process related to the government of the state of each node. And still being modular to the point of allowing the possibility of being easily extendable to other file systems other than Btrfs.

We discussed the complete installation process of the iCBD platform, from root, in the infrastructure of the Department of Computer Science of FCT NOVA, detailing some of the challenges encountered. All this work culminated in the elaboration of an installation manual and demonstrating the feasibility of integrating a Cache Server into the iCBD platform.

Finally, we evaluated both components of this work by performing a comprehensive functional validation, and by designing a benchmark to the system performance of both replication and caching systems. Being that the results demonstrated total feasibility of the solutions proposed, having been achieved the integration with the work already developed in the iCBD platform.

In the replication system, we saw that the solution presented meets the proposed requirements and obtains performance levels in line with what was expected. Regarding the introduction of a Cache Server on the platform, the results consistently show a more

considerable boot time when comparing to clients that boot from a machine with more significant resources. Which is mainly due to the cache server possessing hardware not much better than a workstation, we can discern that when compared to a traditional server-based VDI solution [36], the load on this component, even in a boot storm scenario, does not exceed 20%. This fact leads us to conclude that the platform can be scaled horizontally.

6.2 Future Work

During the process of development of this work, several ideas for future developments in this project came up.

Taking advantage of the extensibility of the assembled system for iMI replication, it could be extended in such a way that the replicas may communicate with each other, allowing the transfer of versions of iMIs directly between replicas. This scenario would be especially advantageous in a circumstance where the Master Node is located in a geographic location far way from the Replicas or even hosted in a public cloud. With several Replica Nodes installed on the same network, only one of the nodes needed to receive a new version of the iMI then he would send it directly to the other Replicas.

Another idea that started to be implemented in the modularity of the installation of the iCBD platform on the FCT NOVA site is the segmentation of the multiple components into microservices, where each one is able to operate autonomously using a kind of inter-process communication mechanism or merely the network itself. With this change, it is believed that it would be possible to make better use of the resources allocated to the platform, and probably even more important is the fact that it provides the support for launching services independently of their location, as well as facilitating the development in parallel of different features of the platform without affecting the platform as a whole.

Bibliography

[1] M. Adler and J. loup Gailly. *zlib Home Site*. 2018. URL: https://zlib.net/ (visited on 09/01/2018).

[2] O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam. "The Evolution of an x86 Virtual Machine Monitor." In: *SIGOPS Oper. Syst. Rev.* 44.4 (Dec. 2010), pp. 3–18.

[3] N. Alves. "Linked clones baseados em funcionalidades de snapshot do sistema de ficheiros." Master's thesis. Universidade NOVA de Lisboa, 2016.

[4] Amazon Web Services. *Amazon Simple Storage Service (S3)*. 2017. URL: https://aws.amazon.com/s3/ (visited on 02/10/2017).

[5] Amazon Web Services. *Amazon Machine Images (AMI)*. 2018. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html (visited on 07/02/2018).

[6] Amazon Web Services. *Amazon WorkSpaces - Virtual Desktops in the Cloud*. 2018. URL: https://aws.amazon.com/workspaces (visited on 02/05/2018).

[7] *Amazon Web Services (AWS) - Cloud Computing Services*. 2017. URL: https://aws.amazon.com/ (visited on 02/05/2017).

[8] A. Aneja. *Designing Embedded Virtualized Intel ® Architecture Platforms with the right Embedded Hypervisor*. Tech. rep. 2011, pp. 1–14. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-embedded-virtualized-hypervisor-paper.pdf.

[9] *AppDelivery Solutions - Desktop Virtualization*. 2017. URL: https://appds.eu/Home/DesktopVirt (visited on 02/05/2017).

[10] G. Brandl. *Sphinx - Python Documentation Generator*. 2018. URL: http://www.sphinx-doc.org/en/master/ (visited on 09/01/2018).

[11] J. P. Buzen and U. O. Gagliardi. "The Evolution of Virtual Machine Architecture." In: *Proceedings of the June 4-8, 1973, National Computer Conference and Exposition* (1973), pp. 291–299.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: A distributed storage system for structured data." In: *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA* (2006), pp. 205–218.

[13] H. Chirammal, P. Mukhedkar, and A. Vettathu. *Mastering KVM Virtualization*. Packt Publishing, 2016. ISBN: 9781784396916.

[14] *Citrix Bids Adieu to XenClient*. 2015. URL: http://vmblog.com/archive/2015/09/24/citrix-bids-adieu-to-xenclient.aspx (visited on 02/07/2017).

[15] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. 5th. USA: Addison-Wesley Publishing Company, 2011. ISBN: 0132143011, 9780132143011.

[16] B. Das, Y. Z. Zhang, and J. Kiszka. *Nested Virtualization - State of the art and future directions*. Tech. rep. 2013, pp. 1–29. URL: https://www.linux-kvm.org/images/3/33/02x03-NestedVirtualization.pdf.

[17] T. P. S. Foundation. *zlib — Compression compatible with gzip*. 2018. URL: https://docs.python.org/3/library/zlib.html (visited on 09/01/2018).

[18] Google. *Google Cloud Platform - Cloud Storage*. 2017. URL: https://cloud.google.com/storage/ (visited on 02/10/2017).

[19] Google. *snappy*. 2018. URL: https://github.com/google/snappy (visited on 09/01/2018).

[20] *Google Cloud Platform*. 2017. URL: https://cloud.google.com/ (visited on 02/05/2017).

[21] Gordon McMillan. *Socket Programming HOWTO*. 2018. URL: https://docs.python.org/3/howto/sockets.html (visited on 09/01/2018).

[22] IBM Corporation. *Inside the Linux boot process*. 2018. URL: https://www.ibm.com/developerworks/library/l-linuxboot/index.html (visited on 07/04/2018).

[23] IBM Corporation. *Linux initial RAM disk (initrd) overview*. 2018. URL: https://www.ibm.com/developerworks/library/l-initrd/index.html (visited on 07/04/2018).

[24] A. M. D. Inc. *AMD-V Nested Paging*. Tech. rep. 2008, pp. 1–19. URL: http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf.

[25] V. Inc. *VDI : A New Desktop Strategy*. Tech. rep. 2006, pp. 1–19. URL: https://www.vmware.com/pdf/vdi_strategy.pdf.

[26] V. Inc. *Virtualization overview*. Tech. rep. 2006, pp. 1–11. URL: http://www.vmware.com/pdf/virtualization.pdf.

[27] V. Inc. *Understanding Memory Resource Management in VMware ESX Server*. Tech. rep. 2009, pp. 1–20. URL: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/perf-vsphere-memory_management.pdf.

[28] Iotic-Labs. *py-lz4framed*. 2018. URL: https://github.com/Iotic-Labs/py-lz4framed (visited on 09/01/2018).

[29] Irmen de Jong. *Pyro 4.x - Python remote objects*. 2018. URL: https://github.com/irmen/Pyro4 (visited on 09/01/2018).

[30] S. Jain. *Considerations for implementing a desktop virtualization strategy*. Tech. rep. 2014, pp. 1–8. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/practical-considerations-desktop-virtualization-paper.pdf.

[31] Kernel.org - Linux Kernel Organization, Inc. *Compression - btrfs Wiki*. 2018. URL: https://btrfs.wiki.kernel.org/index.php/Compression (visited on 09/01/2018).

[32] Kernel.org - Linux Kernel Organization, Inc. *Design notes on Send/Receive - btrfs Wiki*. 2018. URL: https://btrfs.wiki.kernel.org/index.php/Design_notes_on_Send/Receive (visited on 09/01/2018).

[33] Kernel.org - Linux Kernel Organization, Inc. *Manpage/btrfs-receive - btrfs Wiki*. 2018. URL: https://btrfs.wiki.kernel.org/index.php/Manpage/btrfs-receive (visited on 09/01/2018).

[34] Kernel.org - Linux Kernel Organization, Inc. *Manpage/btrfs-send - btrfs Wiki*. 2018. URL: https://btrfs.wiki.kernel.org/index.php/Manpage/btrfs-send (visited on 09/01/2018).

[35] P. Lopes. *Proposta de Candidatura ao programa P2020*. Tech. rep. DI-FCT/NOVA, Reditus S.A, 2015, pp. 1–26.

[36] P. Lopes, M. Martins, N. Preguiça, and P. Medeiros. *CLME2017/VCEM - Presentation : iCBD*.

[37] P. Lopes, N. Preguiça, P. Medeiros, and M. Martins. "iCBD: Uma Infraestrutura Baseada nos Clientes para Execução de Desktops Virtuais." In: *Proceedings CLME2017/VCEM 8º Congresso Luso-Moçambicano de Engenharia / V Congresso de Engenharia de Moçambique* (2017), pp. 13–18.

[38] E. Martins. "Object-Base Storage for the support of Linked-Clone Virtual Machines." Master's thesis. Universidade NOVA de Lisboa, 2016.

[39] P. Mell and T. Grance. "The NIST definition of Cloud Computing." In: *NIST Special Publication* 145 (2011), p. 7.

[40] *Microsoft Cloud Computing Platform and Services*. 2017. URL: https://azure.microsoft.com/ (visited on 02/05/2017).

[41] Microsoft Cloud Platform. *Desktop virtualization and Virtual Desktop Infrastructure*. 2018. URL: https://www.microsoft.com/en-us/cloud-platform/desktop-virtualization (visited on 02/05/2018).

[42] *Microsoft Remote Desktop Services (RDS) Explained*. 2010. URL: https://technet.microsoft.com/en-us/video/remote-desktop-services-rds-explained.aspx (visited on 02/07/2017).

[43] *Microsoft Security TechCenter - Microsoft Security Updates*. 2017. URL: https://technet.microsoft.com/en-us/security/bulletins.aspx (visited on 01/29/2018).

[44] A. Moreira. *python-snappy*. 2018. URL: https://github.com/andrix/python-snappy (visited on 09/01/2018).

[45] G. J. Popek and R. P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures." In: *Communications of the ACM* 17.7 (1974), pp. 412–421.

[46] M. Portnoy. *Virtualization Essentials*. 1st. Alameda, CA, USA: SYBEX Inc., 2012. ISBN: 1118176715, 9781118176719.

[47] *Remote Desktop Protocol*. 2017. URL: https://msdn.microsoft.com/en-us/library/aa383015(v=vs.85).aspx (visited on 02/07/2017).

[48] M. Righini. *Enabling Intel Virtualization Technology Features and Benefits*. Tech. rep. 2010, pp. 1–9. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/virtualization-enabling-intel-virtualization-technology-features-and-benefits-paper.pdf.

[49] O. Rodeh, J. Bacik, and C. Mason. "BTRFS: The Linux B-Tree Filesystem." In: *ACM Transactions on Storage* 9.3 (2013), pp. 1–32.

[50] D. N. S. Shepler M. Eisler. *Network File System (NFS) Version 4 Minor Version 1 Protocol*. RFC 5661. Internet Engineering Task Force (IETF), 2010, pp. 1–617. URL: https://tools.ietf.org/html/rfc6143.

[51] M Satyanarayanan. "A Survey of Distributed File Systems." In: *Annu. Rev. Comput. Sci.* 4.4976 (1990), pp. 73–104.

[52] SwiftStack. *OpenStack Swift*. 2017. URL: https://www.swiftstack.com/product/openstack-swift (visited on 02/10/2017).

[53] J. L. T. Richardson. *The Remote Framebuffer Protocol*. RFC 6143. Internet Engineering Task Force (IETF), 2011, pp. 1–39. URL: https://tools.ietf.org/html/rfc6143.

[54] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN: 0132392275.

[55] C. Tsaousis. *netdata*. 2018. URL: https://github.com/netdata/netdata (visited on 09/01/2018).

[56] *VMware Horizon*. 2017. URL: http://www.vmware.com/products/horizon.html (visited on 02/07/2017).

[57] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C Maltzahn. "Ceph: A Scalable, High-Performance Distributed File System." In: *Proceedings of USENIX Symposium on Operating Systems Design and Implementation* (2006), pp. 307–320.

[58] *What Files Make Up a Virtual Machine?* 2006. URL: https://www.vmware.com/support/ws55/doc/ws_learning_files_in_a_vm.html (visited on 02/05/2017).

[59] Workspot. *The Workspot Desktop Cloud*. 2018. URL: https://www.workspot.com/daas-2-0/ (visited on 02/05/2018).

[60] *XenApp & XenDesktop*. 2017. URL: https://www.citrix.co.uk/products/xenapp-xendesktop/ (visited on 02/07/2017).

[61] C. Zikmund. *Key Considerations in Choosing a Zero Client Environment for View Virtual Desktops in VMware Horizon*. Tech. rep. 2014, pp. 1–12. URL: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vmware-top-five-considerations-for-choosing-a-zero-client-environment.pdf.

[62] J. Ziv and A. Lempel. "A universal algorithm for sequential data compression." In: *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343. ISSN: 0018-9448. DOI: 10.1109/TIT.1977.1055714.

ANNEX I

iCBD-Replication Documentation

iCBD-Replication Documentation

Release 1.0.0

Luis Silva

Nov 19, 2018

ICBD REPLICATION MODULE

This site covers iCBD-Replication usage & API documentation. For basic info on what iCBD-rep is, including its public changelog & how the project is maintained, please see the git repo.

CHAPTER

ONE

API DOCUMENTATION

We maintain a set of API documentation, autogenerated from the python source code's docstrings (which are typically very thorough.) and for the RESTfull API (TODO: FUTURE)

1.1 icbdrep.ImageRepo module

class icbdrep.ImageRepo.**ImageRepo**(*config*)

Bases: object

addImage(*image_name: str*)

Add an image name to the repository And checks if in that directory are already present some snapshots

Args: image_name: name of the image to be added

Returns: None

Raises: DirNotFoundException, BTRFSPathNotFoundException, ImageAlreadyExistsException

addSnapshot(*image_name: str*, *snap_number: str*) → None

Add a snapshot to a image

Args: image_name: the name of the image to receive a snapshot snap_number: the snapshot

Returns: None

Raises: BTRFSSubvolumeNotFoundException, SnapshotAlreadyExistsException

deleteImage(*image_name: str*) → None

Deletes a given image from the repository

Args: image_name: the name of the image to be deleted

Returns: None

Raises: ImageNotFoundException

deleteSnapshot(*image_name: str*, *snap_number: str*) → lib.icbdSnapshot.icbdSnapshot

Deletes a given snapshot of an image

Args: image_name: the image to which the snapshot refers to snap_number: the snapshot number

Returns: None

Raises: SnapshotNotFoundException

getImagelist() → typing.List[str]

Get the list of the VM images present in the repo

Returns: a list of strings with the images names

getImagepath(*image_name: str*) → str
Returns the path to the given image.

Args: image_name: the name of the image

Returns: a string with the path to the image

Raises: ImageNotFoundException

getLastSnapshot(*image_name: str*) → lib.icbdSnapshot.icbdSnapshot
Get the last snapshot from the given image.

Args: image_name: name of the image

Returns: an obj icbdSnapshot

Raises: ImageNotFoundException

getSnapshot(*image_name: str*, *snap_number: str*) → lib.icbdSnapshot.icbdSnapshot
Gets a specific snapshot given its number and the image name

Args: image_name: the name of the image snap_number: the number of the snapshot

Returns: an icbdSnapshot object

Raises: SnapshotNotFoundException

getSnapshotlist(*image_name: str*) → typing.List[lib.icbdSnapshot.icbdSnapshot]
Get the list of snapshots present in the repo for the given image. If there are no snapshots it returns a empty list.

Args: image_name: The image name that contains the snapshots

Returns: a list with the snapshots present in the repo

Raises: ImageNotFoundException

hasImage(*image_name: str*) → bool
Check if a given image name is present in the repository

Args: image_name: the image name to be checked

Returns: True if present, otherwise False

hasSnapshot(*image_name: str*, *snap_number: str*) → bool
Check if a snapshot is present in the given image

Args: image_name: the name of the image that should contain the snapshot snap_number: the snapshot

Returns: True if the snapshot is present, otherwise False

1.2 icbdrep.KeepAlive module

class icbdrep.KeepAlive.**KeepAlive**(*interval=10*, *tries_num=3*)
Bases: threading.Thread

keepAlive(*pyro_bind: bool*) → None
Check a replica state and updates NS if needed.

Args: pyro_bind: boolean True to use of the _pyroBind or False to use the ping method

Returns: None

run()
The main method of the class. This is triggered in the thread.start() call

Returns: None

stopKeepAlive() → None
Stop the execution of the keep alive thread. This should be part of the shutdown process.

Returns: None

1.3 icbdrep.MasterNode module

class icbdrep.MasterNode.**MasterNode**(*node_config*, *ns_config*, *interactive_mode_flag: bool*)
Bases: threading.Thread

addImage(*image_name: str*, *node: int*) → None
Add an image to the node repository

Args: image_name: the name of the image to be added node: the node where the image will be added

Returns: Node

delete_snapshot(*image_name: str*, *snap_number: str*, *node: int*) → None
Deletes a snapshot from a given image in a node.

Args: image_name: the image name snap_number: the snapshot number node: the node to do the deletion

Returns: None

exeCommand(*line: str*) → None
Receives a command line and interprets the content. Separating the various fields of the string into arguments, and calls the appropriated function.

Args: line: a line with the command to execute

Returns: None

getReplicasFromNS() -> (<class 'int'>, typing.Dict[int, Pyro4.core.Proxy])
Get a list of the replicas present in the system (Name Server) and saves them to the replicas proxy list

Returns: the number of found replicas

interactiveMode() → None
When in interactive mode, the server runs with a prompt, so that individual commands can be typed in

Returns: None

listImages(*node: int*) → None
List the collection of images available in a node.

Args: node: The node to list. (Master or one of the Replicas)

Returns: None

listReplicas() → None
List the replicas present in the system and prints to the console.

Returns: None

listSnapshots(*node: int*, *image_name: str*) → None
List the colection of snapshots of a given image in a node.

Args: node: The node to list (Master or one of the replicas) image_name: The image the snapshots refer to

Returns: None

registerInNS() → Pyro4.core.Daemon
Register the server in the Name Server

Returns: the registered daemon

run()
The main method of the class. This is triggered in the thread.start() call

Returns: None

send(*node: int*, *image_name: str*, *snapshot_number: str*, *blocking: bool*, *ssh: bool = False*, *compression: str = None*) → None
Send Command - Instructs the replica to listen for a transfer, and sends the snapshot in the btrfs path

Args: node: the number of the node image_name: the name of the image snapshot_number: the number of the image blocking: if the function should block

Returns: None

stopMaster() → None
WARNING!! Don't use this! Only for testing and should be deprecated!

Returns: None

1.4 icbdrep.NameServer module

class icbdrep.NameServer.**NameServer**(*config*)
Bases: threading.Thread

run()
The main method of the class. This is triggered in the thread.start() call

Returns: None

stopNS() → None
This function closes both the broadcast and name servers. This is called in the shutdown procedure.

Returns: None

1.5 icbdrep.ReplicaNode module

class icbdrep.ReplicaNode.**ReplicaNode**(*rep_id: int*, *node_config*, *ns_config*)
Bases: object

addImage(*image_name: str*) → bool
Add an image to the node's repository

Args: image_name: the name of the image to be added.

Returns: a boolean with the sucess of the operation

deleteSnapshot(*image_name: str*, *snap_number: str*) → lib.icbdSnapshot.icbdSnapshot
Delete a snapshot from the repo and FS

Args: image_name: the name of the image snap_number: the number of the snapshot

Returns: the snapshot which as deleted

getImagesList() → typing.List[str]
Get the list of images present in the replica

Returns: a list of strings

getLastSnapshot(*image_name: str*) → lib.icbdSnapshot.icbdSnapshot
Return the last snapshot of the given image.

Args: image_name: the name of the image

Returns: an obj icbdSnapshot

getName() → str
Get the replica name

Returns: a string with the name

getReplicaBtrfsAddress() → typing.Tuple[str, int]
Return the IP and PORT address for the btrfs transfer.

Returns: A tuple with an IP and PORT

getReplicaID() → int
Get the replica ID number. This should be a integer that originates from the

Returns: the replica ID

getSnapshotList(*image_name: str*) → typing.List[lib.icbdSnapshot.icbdSnapshot]
Return the list of snapshots stored in the repo for the given image name. Case there are no snapshots the list returned is empty. Case the image in args ins't in the repo return None.

Args: image_name: Image name to get the snapshot list.

Returns: a list with the snapshots.

ping() → str
Responds to a ping request with "pong"

Returns: "pong"

poisonPill() → None
Shutdown message to the replica

Returns: None

prepareReceive(*image_name: str*, *snap_number: str*) → bool
This function should precede the receive() call. Checks if the node wants the image in question or if the snapshot is already present.

Args: image_name: the name of the image snap_number: the name of the snap

Returns: a bool that indicates if the replica will accept the receive

receive(*image_name: str*, *snap_number: str*, *compression: str = None*)
Receives a snapshot

Returns: None

1.6 icbdrep.icbdrepd module

1.7 lib.benchmarkinglib module

class lib.benchmarkinglib.**Benchmark**(*name*)
Bases: object

addRun(*run: lib.benchmarkinglib.Run*)

get_name()

mean()

median()

stdev()

class lib.benchmarkinglib.**Run**(*interfaceName*, *runNumber=-1*, *imageName='default'*)
Bases: object

getBtrfsTransferBytes()
Returns:

getBtrfsTransferPackets()
Returns:

getBtrfsTransferRuntime()
Returns:

getGlobalTransferRuntime()
Returns:

getIcbdBootTransferBytes()
Returns:

getIcbdBootTransferPackets()
Returns:

getIcbdBootTransferRuntime()
Returns:

getIscsiTargetTransferBytes()
Returns:

getIscsiTargetTransferPackets()
Returns:

getIscsiTargetTransferRuntime()
Returns:

startTimmer(*transferType*)
Start a timmer for one of the transfer counters.

Args: transferType: the type of the transfer to start counting time

Returns: call the appropriated function

stopTimmer(*transferType*)
Stop a timmer for one of the transfer counters.

Args: transferType: the type of the transfer to start counting time

Returns: call the appropriated function

class lib.benchmarkinglib.**linuxNetworkTraffic**
Bases: object

static getInterfaceStats(*interfaceName*)

Args: interfaceName:

Returns:

1.8 lib.btrfslib module

class lib.btrfslib.**BtrfsFsCheck**
Bases: object

static isBtrfsPath(*path: str*)
Check if a given path is in fact present in a BTRFS tree

!!Caution!! : This function does not takes into account the fact that the path might not be a valid one.

Args: path: the path to be checked

Returns: true if present, otherwise falses

static isBtrfsSubvolume(*path: str*)
Check if the given path is a BTRFS subvolume / snapshot.

Args: path: the path to be checked

Returns: True if a subvolume, otherwise false

static searchForSnapshots(*path: str*) → typing.List[str]
Search the directory , and gets the snapshots that are already present

Args: path: the directory to be searched

Returns: a List with the name of the snapshot

class lib.btrfslib.**BtrfsTool**
Bases: object

static delete(*path: str*) → None
Wrapper for the BTRFS Tools subvolume delete command.

The method receives a path and calls the btrfs subvolume delete for that path.

Args: path: the path to the subvolume to delete

Returns: None

static receive(*dst_path: str*, *src_port: int*, *compression: str = None*)
Wrapper for the BTRFS Tools receive() command.

This method opens a socket and listens for a connection Then receives a snapshot and redirect it to the stdin of the BTRFS receive

Args: dst_path: the path of the image to place the snapshot src_port: the port to listening for the transfer

Returns: None

static send(*src_path: str*, *dst_ip: str*, *dst_port: int*, *parent: str = None*, *compression: str = None*)
Wrapper for the BTRFS Tools send() command.

This method is BLOCKING, it will wait for the conclusion of the send command. It uses regular sockets to send to an endpoint the data from the snapshot.

Args: src_path: the path of the snapshot to be send dst_ip: the IP of the destiny socket dst_port: the Port the destiny is listening

Returns: None

static sendNonBlock(*src_path: str*, *dst_ip: str*, *dst_port: int*, *parent: str = None*, *compression: str = None*)
Wrapper for the BTRFS Tools send() command.

This method is NON BLOCKING, it will NOT wait for the conclusion of the send command. It uses regular sockets to send to an endpoint the data from the snapshot.

Args: src_path: the path of the snapshot to be send dst_ip: the IP of the destiny socket dst_port: the Port the destiny is listening

Returns: None

static sendSSH(*src_path: str*, *dst_ip: str*, *dst_port: int*, *parent: str = None*, *compression: str = None*)
Wrapper for the BTRFS Tools send() command.

This method is BLOCKING, it will wait for the conclusion of the send command. It uses regular sockets to send to an endpoint the data from the snapshot.

Args: src_path: the path of the snapshot to be send dst_ip: the IP of the destiny socket dst_port: the Port the destiny is listening

Returns: None

static setReadOnly(*path: str*, *state: bool*) → None
Wrapper for the BTRFS Tools property set read only command.

This method sets the the read only property for the given subvolume in the path.

Args: path: the path to the subvolume state: a boolean of the state of the read only

Returns: None

1.9 lib.compressionlib module

class lib.compressionlib.**compressionLib**
Bases: object

static checkCompression(*compression: str*) → bool
Check if the given compression algorithm is available to use in the lib.

Args: compression: A string with the algorithm to check

Returns: A bool representing the availability of the chosen algo.

class lib.compressionlib.**g_snappy**
Bases: object

static compressStream(*in_stream*, *out_stream*, *blocksize=65536*) → None
Uses the Google snappy compress function to compress a stream of bytes.

Takes an incoming file-like object and an outgoing file-like object, reads data from "in_stream", compresses it, and writes it to "out_stream". "in_stream" should support the read method, and "out_stream" should support the write method.

Args: in_stream: a stream of bytes out_stream: a compressed stream blocksize: [optional] the size used for the buffer in bytes

Returns: None

static compress_native(*in_stream*, *out_stream*, *blocksize=65536*) → None
Wrapper for the snappy native stream compression

Args: in_stream: a stream of bytes out_stream: a compressed stream blocksize: [optional] the size used for the buffer in bytes

Returns:

static decompressStream(*in_stream*, *out_stream*, *blocksize=65536*) → None
Uses the Google snappy decompress function to handle a compressed stream.

Takes an incoming file-like object and an outgoing file-like object, reads data from "in_stream", decompresses it, and writes it to "out_stream". "in_stream" should support the read method, and "out_stream" should support the write method.

Args: in_stream: a compressed stream out_stream: the original stream of bytes blocksize: [optional] the size used for the buffer in bytes

Returns:None

static decompress_native(*in_stream*, *out_stream*, *blocksize=65536*) → None
Wrapper for the snappy native stream decompression

Args: in_stream: a compressed stream out_stream: the original stream of bytes blocksize: [optional] the size used for the buffer in bytes

Returns:

class lib.compressionlib.**lz4**
Bases: object

static compressStream(*in_stream*, *out_stream*) → None
Uses the lz4 compress function to compress a stream of bytes

Takes an incoming file-like object and an outgoing file-like object, reads data from "in_stream", compresses it, and writes it to "out_stream". "in_stream" should support the read method, and "out_stream" should support the write method.

Args: in_stream: a bytes input stream to be compressed out_stream: the compressed stream

Returns: None

static decompressStream(*in_stream*, *out_stream*) → None
Uses the lz4 decompress function to decompress a stream of bytes

Takes an incoming file-like object and an outgoing file-like object, reads data from "in_stream", decompresses it, and writes it to "out_stream". "in_stream" should support the read method, and "out_stream" should support the write method.

Args: in_stream: a compressed stream out_stream: the original bytes

Returns: None

class lib.compressionlib.**z_lib**
Bases: object

static compress2(*in_stream*, *out_stream*)
!!IN TESTING!! !!DONT USE THIS!!

Args: in_stream: out_stream:

Returns:

static compressStream(*in_stream*, *out_stream*, *blocksize=32768*) → None
Uses the zlib compress function to compress a stream of bytes.

Takes an incoming file-like object and an outgoing file-like object, reads data from "in_stream", compresses it, and writes it to "out_stream". "in_stream" should support the read method, and "out_stream" should support the write method.

Args: in_stream: a stream of bytes out_stream: a compressed stream blocksize: [optional] the size used for the buffer in bytes

Returns: None

static decompress2(*in_stream*, *out_stream*)
!!IN TESTING!! !!DONT USE THIS!!

Args: in_stream: out_stream:

Returns:

static decompressStream(*in_stream*, *out_stream*, *blocksize=32768*) → None
Uses the zlib decompress function to handle a compressed stream.

Takes an incoming file-like object and an outgoing file-like object, reads data from "in_stream", decompresses it, and writes it to "out_stream". "in_stream" should support the read method, and "out_stream" should support the write method.

Args: in_stream: a compressed stream out_stream: the original stream of bytes blocksize: [optional] the size used for the buffer in bytes

Returns: None

1.10 lib.icbdSnapshot module

class lib.icbdSnapshot.**icbdSnapshot**(*mount_point: str*, *image_name: str*, *snapshot_number: str*, *icbd_boot_package_path: str*, *iscsi_target_folder: str*)
Bases: object

getICBDBootPackagePath()
Get a string with the full path to the iCBD Boot Package of the Image.

Returns: a string with the path

getISCSITarget()
Get a string with the path to the ISCSI target for this snapshot.

Returns: a string with the path

getImagePath() → str
Get a string with the formatted path, but without the snapshot number. This should be used as a destiny path

Returns: a string with the path in the format {/mountpoint/imagename}

getMountpointPath() → str
Get a string with only the mount point of the snapshot

Returns: the mountpoint

getPath() → str
Get a string with the full path of the snapshot, including the mountpoint and image name. Format: {mountpoint/imagename/snapshotnumber}

Returns: a string with the path

1.11 lib.restapilib module

class lib.restapilib.**RestAPI**(*port: int = 5009*)

Bases: object

iCBD-Replication Rest API Class

This instantiate the micro-framework Flack to provide a simple HTTP API for interacting with the system.

Note that every communication with this API uses JSON files. Responses are in JSON and an example can be found in the documentation of each method.

api = <flask_restful.Api object>

app = <Flask 'lib.restapilib'>

deleteImageVersion(*replica*, *imi*, *version*)

Delete a version of an iMI in a Replica Json response example:

Endpoint path : <IP>:<Port>/api/replicas/<replica>/imis/<imi>/versions/<version>/delete/

Returns:

listImageVersionsByReplica(*replica*, *imi*)

List the version of an iMI that are present in a Replica Json response example:

Endpoint path : <IP>:<Port>/api/replicas/<replica>/imis/<imi>/versions

Returns:

listImagesByReplica(*replica*)

List all iMIS present in a replica. Json response example:

Endpoint path : <IP>:<Port>/api/replicas/<replica>/imis

Returns:

listReplicas()

List all the replicas registered in the system. Json response example:

Endpoint path : <IP>:<Port>/api/replicas

Returns:

listSystemImages()

List all the iMIs present in the Master Node This will list all iMIs available to be transfered to any replica.

Json response example:

Endpoint path : <IP>:<Port>/api/master/imis

Returns:

listSystemImagesVersions(*imi*)

List all the versions of an iMIs present in the Master Node

Json response example:

Endpoint path : <IP>:<Port>/api/master/imis/<imi>/versions

Returns:

root()
Default root route endpoint. Mainly for testing

Endpoint path : <IP>:<Port>/api

Returns: a simple test string

sendImageVersionToReplica()
List all the versions of an iMIs present in the Master Node

Json response example:

Endpoint path : <IP>:<Port>/api/master/send?imi={imi}&version={version}&replica={replica}

Returns:

subscribeImage(*replica*, *imi*)
Replica subscribe to a iMI Json response example:

Endpoint path : <IP>:<Port>/api/replicas/<replica>/imis/subscribe/<imi>

Returns:

unsubscribeImage(*replica*, *imi*)
Replica unsubscribe to a iMI Json response example:

Endpoint path : <IP>:<Port>/api/replicas/<replica>/imis/unsubscribe/<imi>

Returns:

1.12 lib.serializerslib module

class lib.serializerslib.**icbdSnapshotSerializer**
Bases: object

static icbdSnapshot_class_to_dict(*obj: lib.icbdSnapshot.icbdSnapshot*)

static icbdSnapshot_dict_to_class(*class_name*, *dict*)

1.13 lib.sshlib module

class lib.sshlib.**sshTunnel**(*host*, *local_port*, *remote_port*)
Bases: object

createTunnel(*host*, *local_port*, *remote_port*)

1.14 lib.utillib module

class lib.utillib.**icbdUtil**
Bases: object

logHeading(*string*)
Big header for logger –[“string”]————————

Args: string: a string to be placed inside the big header

Returns: the string encapsulated in the header

prettify(*obj*)
Return pretty representation of obj. Useful for debugging.

Args: obj: the object to prettify

Returns: a pretty representation of obj

1.15 exceptions.ImageRepoException module

exception exceptions.ImageRepoException.**BTRFSPathNotFoundException**(*message*)
Bases: Exception

Raise when a BTRFS Path is not in the File System

exception exceptions.ImageRepoException.**BTRFSSubvolumeNotFoundException**(*message*)
Bases: Exception

Raise when a BTRFS Subvolume is not in the File System

exception exceptions.ImageRepoException.**DirNotFoundException**(*message*)
Bases: Exception

Raise when a Directory is not in the File System

exception exceptions.ImageRepoException.**ImageAlreadyExistsException**(*message*)
Bases: Exception

Raise when a Images already is present in the repo

exception exceptions.ImageRepoException.**ImageNotFoundException**(*message*)
Bases: Exception

Raise when a Images is not found

exception exceptions.ImageRepoException.**SnapshotAlreadyExistsException**(*message*)
Bases: Exception

Raise when a Snapshot already is present in the repo

exception exceptions.ImageRepoException.**SnapshotNotFoundException**(*message*)
Bases: Exception

Raise when a Snapshot is not found

1.16 exceptions.ReplicasException module

exception exceptions.ReplicasException.**ReplicaNotFoundException**(*message*)
Bases: Exception

Raise when a replica is not found

1.17 tests.benchLibTests module

tests.benchLibTests.**dummyFunc**()

tests.benchLibTests.**main**()

tests.benchLibTests.**startCompleteRun**()

1.18 tests.pyroNSTests module

class tests.pyroNSTests.**NamingTrasher**(*nsuri*, *number*)
Bases: threading.Thread

list()

listprefix()

listregex()

lookup()

register()

remove()

run()

tests.pyroNSTests.**main**()

tests.pyroNSTests.**randomname**()

1.19 tests.utilTests module

class tests.utilTests.**TestMount**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Our basic test class

isBTRFS(*path*, *assertVal*)

isSubvolume(*path*, *assertVal*)

test_isBtrfsSet()

test_isSubvolumeSet()

1.20 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

i

l

t

INDEX

H

I

K

L

ANNEX II

iCBD Installation Guide

iCBD Installation Protocol

Version 1.0.1 - Last Updated 9 Oct 2018

Luis Silva - `lmt.silva (at) campus.fct.unl.pt`

In this document, we will detail all the steps needed to entirely install from scratch and start the iCBD Management Platform.

Pre-Requisites

What is needed:

- 3 x *CentOS 7* Minimum Install VM
  - 2 Hard Drives (the extra for *BTRFS*)
  - 1 or more NICs (Depending on the VM)
- iCBD install files for each VM
- Some iCBD VM images

Attention - CentOS 7 Kernel Version! The Kernel `3.10.0-693.5.2.el7.x86_64` on CentOS 7 has manifested a problem with a core component of the *coreutils* tool command, the `cp` when used with option `--reflink=always`. To circumvent the issue is advised to use an older Kernel, such as `3.10.0-514.2.2.el7.x86_64` as we confirmed is working. This until Red Hat releases a new kernel with the bug fix.

Introduction

This tutorial assumes a fresh minimal install of a CentOS 7 Operating System. The installation procedure will cover all configurations needed for the implementation of VMs that will take a role in the platform. Some of the settings are specific for one of the roles, in this case, there will be a note in the step description.

iCBD Roles

The iCBD Management Platform consists of a minimum of three VM's, but for a more complex typology, we can mix in some cache servers and some clients. So we can have the following roles:

- *iCBD-imgs* - Primary repository of VM images and facilitator of the administration process
- *iCBD-rw* - Provides read/write space to the iCBD clients
- *iCBD-home* - Hosting of Home accounts to be used by iCBD clients
- *iCBD-cache* - Hosting of VM images closest to the clients
- *iCBD-Client* - A VM shell that don't have a hard disk and will boot from network

iCBD Networks

Also, there is the need to define multiple networks. Here, as we are using the VMware platform, there is the ability to design a Distributed VSwitch with various Port Groups, each one symbolising an individual network. The networks are:

On the iCBD-DSwitch (This distributed virtual switch only works inside the cluster)

- iCBD-Net
- iCBD-Adm-Net
- iCBD-Rep
- iCBD-CacheXX-Net

On the DI-DSwitch (Outside access to DI networks and Internet)

- DMZ-PRIV-DI
- DMZ-PUB-DI
- R-ENSINO-PRIV-DI

In the next table is showed the characteristics of each VM given its role. These properties mirror what is implemented in the Cluster at *DI - FCT NOVA*. Then we present two tables: one with the sizes used for the hard drives, and the other including the networks for the NICs of each VM.

VM Hardware by Role

	iCBD-imgs	iCBD-rw	iCBD-home	iCBD-CacheXX
CPUs (cores)	8	4	4	4
RAM (GB)	32	8	8	32
Hard Drives	2	2	2	2
NICs	3	1	1	2

Hard Drives by Role

	iCBD-imgs	iCBD-rw	iCBD-home	iCBD-CacheXX
Hard Drive 1 (Root FS)	16 GB	16 GB	16 GB	16 GB
Hard Drive 2 (BTRFS)	600 GB	300 GB	100 GB	600 GB

NICs by Role

	iCBD-imgs	iCBD-rw	iCBD-home	iCBD-CacheXX
NIC 1	DMZ-PRIV-DI (Internet)	iCBD-Net	iCBD-Net	iCBD-Net
NIC 2	iCBD-Net	X	X	iCBD-CacheXX-Net
NIC 3	iCBD-Adm-Net	X	X	X

First Step

Let's start:

The first thing we need is a vanilla VM with *CentOS 7* minimal install. This VM will be our basis. Many of the procedures that we will need to implement are more conveniently executed from a terminal in your machine, so probably is a good idea to configure an *SSH* access to the VM. Anyway, you will need to *SSH* to the VM in the future, so it's better to start this way.

Setup a static IP and configure SHH

Setup a static IP address.

Depending on the machine it may be that there is more than one network card installed. In the case of the `iCBD-imgs` this is true. So, I leave here the configuration prepared in this machine.

The VM `iCBD-imgs` as 3 NICs :

- NIC1
  - Port Group: DMZ-PRIV-DI
  - DVSwitch: DSwitch1 (DI-FCT Networks)
  - Used: Outside access
  - Config File - `vi /etc/sysconfig/network-scripts/INTERFACE_NAME`

```
HWADDR=00:50:56:96:A3:52 # Interface MAC Address
TYPE=Ethernet
BOOTPROTO=none
DEFROUTE=yes
IPV4_FAILURE_FATAL=no
IPV6INIT=no
IPV6_FAILURE_FATAL=no
```

```
NAME=ens192
ONBOOT=yes
IPADDR=10.170.137.98    # External IP
NETMASK=255.255.255.0
NM_CONTROLLED=no        # Doesn't let the Network Manager change the
config
PREFIX=24
GATEWAY=10.170.137.254  # Gateway for the .137 network
DNS1=10.130.10.25       # FCT DNS1
DNS2=10.130.10.26       # FCT DNS1
DOMAIN=ensino.priv.di.fct.unl.pt
```

- NIC2
  - Port Group: iCBD-Net
  - DVSwitch: iCBD-DSwitch
  - Used: Main internal network. Platform clients connect were.
  - Config File - `vi /etc/sysconfig/network-scripts/INTERFACE_NAME`

  This NIC will be connected to a bridge, so this is the config for the interface, and then is shown the config for the bridge.

```
HWADDR=00:50:56:96:2E:9C
TYPE=Ethernet
#BOOTPROTO=none
#DEFROUTE=yes
#IPV4_FAILURE_FATAL=yes
#IPV6INIT=no
#IPV6_FAILURE_FATAL=no
NAME=ens224
ONBOOT=yes
#IPADDR=10.0.2.251
#PREFIX=24
BRIDGE=br0
#NETMASK=255.255.255.0
#NM_CONTROLLED=no
ZONE=internal
```

  The Bridge config:

```
DEVICE=br0
STP=yes
TYPE=Brige
BOOTPROTO=none
DEFROUTE=yes
IPV4_FAILURE_FATAL=yes
IPV6INIT=no
NAME="Brige br0"
ONBOOT=yes
BRIDGIN_OPTS=priority=32768
IPADDR=10.0.2.251
PREFIX=24
ZONE=internal
```

- NIC3
  - Port Group: iCBD-Adm-Net
  - DVSwitch: Standard Switch
  - Used: Internal network for the administration machines
  - Config File - `vi /etc/sysconfig/network-scripts/INTERFACE_NAME`

```
HWADDR=00:50:56:96:74:85
TYPE=Ethernet
BOOTPROTO=none
DEFROUTE=yes
IPV4_FAILURE_FATAL=no
IPV6INIT=no
IPV6_FAILURE_FATAL=no
NAME=ens161
ONBOOT=yes
IPADDR=10.0.3.1
NETMASK=255.255.255.128
NM_CONTROLLED=no
PREFIX=24
```

***SSH* access without password**

A configuration with password-less *SSH* access it's highly recommended since you will be connecting to the different servers a lot. A lot!

Still, the next step for your own machine is optional. But since in a later moment, it will be necessary to configure this between the servers and the physical machines the instructions are already here.

For some reference take a look at the next table. Each row represents a particular VM, and the columns indicate the VM keys that should be present in the `~/.ssh/authorized_keys`.

	iCBD-imgs	iCBD-rw	iCBD-home	iCBD-CacheXX	Your Machine
iCBD-imgs		√	√	√	√
iCBD-rw	√		√	√	√
iCBD-home	√	√		√	√
iCBD-CacheXX	√	√	√	√, other caches	√

To generate an *RSA key* pair to work with version 2 of the *SSH* protocol, type the following command at a shell prompt: `ssh-keygen -t rsa`

Transfer your public key to `~/.ssh/authorized_keys`

Need the command ? `cat ~/.ssh/id_rsa.pub | ssh user@server "mkdir -p ~/.ssh && cat >> ~/.ssh/authorized_keys"`

Note: If you are cloning the main VM as a template for the other services, don't forget to create a new *RSA key* and add it to the remaining servers.

Install packages

Now we need to start building the environment with all the necessary tools to run iCBD.

So first run `yum update`, to make sure that all already installed packages are up to date.

Next we need to install all of these packages:

```
yum install net-tools
yum install hdparm
yum install Xorg
yum install gdm
yum install qemu-kvm
yum install virt-manager
yum install gcc
yum install kernel-headers
yum install kernel-devel
yum install epel-release
yum install htop
yum install httpd
yum install ntp
yum install firefox
yum install open-vm-tools
yum install open-vm-tools-desktop
yum install exportfs
yum install vnc
yum install xinetd
yum install tigervnc-server-applet
```

```
yum groupinstall fonts
yum groupinstall "X window system"

yum install kde-workspace
yum install ksysguard
yum install tftp
yum install tftp-server
yum install target-cli **
yum install iscsi-initiator-utils
yum install scsi-target-utils
yum install firewall-config
yum install tcpdump
yum install libvirt
yum install qemu
yum install rsync
yum install php
yum install wget
yum install bind-utils
yum install spice-protocol
yum install spice-server
yum install iotop
yum install iftop
yum install libguestfs
yum install libguestfs-tools
yum install traceroute
yum install strace
yum install nmap
yum install whois
yum install ed
yum install sysstat
yum install rsh
yum install pure-ftpd
```

Setup a RSA key for the apache user

In the iCBD-imgs and iCBD-Cache roles the apache user will need to execute some ssh connections. For that the password-less login is paramount.

Since the we generated a RSA pair for the root user we will use them also for the apache user.

Simply execute the following:

```
mkdir /usr/share/httpd/.ssh
cp /root/.ssh/id_rsa /usr/share/httpd/.ssh/
chown -R apache:apache /usr/share/httpd/.ssh/
chmod 0700 /usr/share/httpd/.ssh/
chmod 0600 /usr/share/httpd/.ssh/id_rsa
```

Setup a graphical environment

It's easier to perform much of the day to day operations if we have a graphical user interface. And given the today's available resources for a development environment, it helps. If you are setting up a production server, then it should be done with scripts..

To activate *KDE* just run `systemctl set-default graphical.target`

In the next restart, you will have a graphical interface instead of a console.

Update date & time

Make sure the time & date are updated

```
systemctl enable ntpd.service
ntpdate pool.ntp.org
systemctl start ntpd.service
```

and to confirm running `date` and compare with our machine.

Disable SELinux

The Security-Enhanced Linux functionality enters into conflict with many components of the iCBD platform, this way there is the need for disabling it. `vi /etc/sysconfig/selinux`

Check if the flag is set to `SELINUX=enforcing`, if so change it either to `permissive` or `disabled` [1]

Ending Step One

Do a `reboot`, just to load everything up, including KDE.

Second Step

Now we start to lay the groundwork for the *iCBD* directories and much-needed mounts. In this sense, we need to start working with the *BTRFS* File System.

Format a second hard drive with BTRFS

You can check the available disks with `ls -l /dev | grep sd`

Let's assume that you have an empty disk ready to being formatted with *BTRFS* underneath `/dev/sdb`

To format the disk with *BTRFS* do a `mkfs.btrfs /dev/sdb`

The above command makes use of the whole disk. But the `mkfs.btrfs` tool as multiple configurations and you can first create some partitions or even multiple disks in a *RAID* configuration and then format them in *BTRFS*. But for simplicity sake (and even taking into account some compartmentalisation issues) let's use the whole disk.

For some follow up on the matter of structuring the disks and multiple partitions there are numerous articles and tutorials on the web. 2

Now you should see that there is a BTRFS file system in the OS.

Use `btrfs filesystem show` to make sure.

Third Step

Now the fun stuff. Mounts!

Caution: From this point on, it is necessary to pay close attention to the mounts, double checking them, as it is enough to fail one and the whole platform may not work.

Mounting the base for the iCBD BTRFS volume

The iCBD needs a *"couple"* of mount points, but every one of them will be under `/var/lib/`. Those will differ from server to server, given the task that it will perform. But this step is universal to every machine.

Let's create a temporary mount for the *BTRFS* disk we created earlier: Execute `mkdir /mnt/btrfs` and then `mount /dev/sdb /mnt/btrfs`.

As we are going to mount the root of the *BTRFS* file system under `/var/lib` there is the need to copy all files and directories first.

Create a sub-volume that will house the *lib* files `btrfs subv create /mnt/btrfs/Lib`, then copy everything to the new sub-volume `cp -a /var/lib/. /mnt/btrfs/Lib/`

Next mount the sub-volume `mount -o subvol=Lib /dev/sdb /var/lib` and check if the mount was sucessful `ls -lah /var/lib/`

Case it looks ok, edit the `fstab` file to make this change permanent: `vi /etc/fstab` Add the line `/dev/sdb /var/lib btrfs subvol=Lib 0 0`

(The arguments are separeted by a tab and the numbers by a space

`/dev/sdb[TAB]/var/lib[TAB]btrfs[TAB]subvol=Lib[TAB]0[SPACE]0`)

and `reboot`.

Fourth Step - iCBD-imgs

More sub-volumes!

These next steps are specific to the *iCBD-imgs VM*, that takes care of the administrations of the images, but also possesses the capability to serve them to the clients. In a future point, we will see the details for the other kind of roles.

Creating the iCBD sub-volumes

Let's create all the following sub-volumes:

```
btrfs subv create /var/lib/icbd
btrfs subv create /var/lib/icbd/.snap
btrfs subv create /var/lib/icbd/shared-vms
mkdir /var/lib/icbd/mounts
btrfs subv create /var/lib/icbd/mounts/vmware
btrfs subv create /var/lib/icbd/mounts/livirt
btrfs subv create /var/lib/icbd/mounts/tftpboot
btrfs subv create /var/lib/icbd/nfs_home
btrfs subv create /var/lib/icbd/nfs_root
btrfs subv create /var/lib/icbd/rw
btrfs subv create /var/lib/icbd/iso
btrfs subv create /var/lib/icbd/tmp
btrfs subv create /var/lib/icbd/icbd
```

The mouting of all this sub-volumes will come later.

Fifth Step - iCBD-imgs

In this installation package there should be a `iCBD-imgs_2017-11-17_bkk.tgz` file. This file is a backup of iCBD-Core and can be used to install.

Transfer the file to the VM, you can use a SSH feature for this:

`scp iCBD-imgs_2017-11-17_bkk.tgz user@host:/var/lib/icbd`

Navigate to `/var/lib/icbd/` on the VM and unzip the file directly to the folder `tar -xvzf iCBD-imgs_2017-11-17_bkk.tgz`.

After this, you can clean up the folder by removing the file: `rm iCBD-imgs_2017-11-17_bkk.tgz`.

Atention - This backup does not contain the folder `/var/lib/icbd/mounts/tftpboot`

Now the remaining mounts I promised. Edit the `fstab` and add this lines:

```
/var/lib/icbd/mounts/vmware        /var/lib/vmware    none    rbind    0 0

/var/lib/icbd/mounts/etc/iscsi    /etc/iscsi    none    rbind    0 0
/var/lib/icbd/mounts/etc/tgt        /etc/tgt    none    rbind    0 0
/var/lib/icbd/mounts/etc/httpd    /etc/httpd    none    rbind    0 0
/var/lib/icbd/mounts/etc/xinetd.d    /etc/xinetd.d    none    rbind    0 0
/var/lib/icbd/mounts/tftpboot        /var/lib/tftpboot    none    rbind
0 0

/var/lib/icbd/mounts/etc/hosts    /etc/hosts     none    bind    0 0
/var/lib/icbd/mounts/etc/exports    /etc/exports    none    bind    0 0
/var/lib/icbd/mounts/etc/dnsmasq.conf    /etc/dnsmasq.conf    none    bind
   0 0

/var/lib/icbd/icbd    /var/lib/tftpboot/icbd        none    rbind    0 0

/var/lib/icbd/bin    /var/lib/icbd/exports/bin    none    rbind    0 0
/var/lib/icbd/include    /var/lib/icbd/exports/include    none    rbind
0 0
/var/lib/icbd/client    /var/lib/icbd/exports/client    none    rbind    0
0

/var/lib/icbd/icbd    /var/lib/icbd/exports/icbd    none    rbind    0 0
/var/lib/icbd/tmp    /var/lib/icbd/exports/tmp    none    rbind    0 0
/var/lib/icbd/iso    /var/lib/icbd/exports/iso    none    rbind    0 0

/var/lib/icbd/shared-vms    /var/lib/icbd/exports/shared-vms    none
rbind    0 0
/var/lib/icbd/nfs_home    /var/lib/icbd/exports/nfs_home    none    rbind
0 0
/var/lib/icbd/nfs_root    /var/lib/icbd/exports/nfs_root    none    rbind
0 0
/var/lib/libvirt/images    /var/lib/icbd/exports/images    none    rbind
0 0
```

Save and `reboot`

Sixth Step - iCBD-imgs

Update the hosts file

Update `hosts` file. Remember, if any changes here done to this file before the last group of mounts this is now without effect. There is a sample `hosts` file in the install package. This server will serve as DHCP it's important that the IP's of the architecture are well defined.

Install the VMware Player.

Also, since we are working with virtualization, maby it's a good time to install one hypervisor. Go to the VMware site and download VMware Workstation 12.

If there is the need for some help in the installation process, check this link to the VMware KB.

Add line to sysctl

`vi /etc/sysctl.conf` and add the line `net.ipv4.ip_forward=1`

Then exe the command `sysctl net.ipv4.ip_forward=1`

Activate NAT

Add direct rules to firewalld. Add the `–permanent` option to keep these rules across restarts.

```
firewall-cmd --direct --add-rule ipv4 nat POSTROUTING 0 -o eth_ext -j
MASQUERADE
firewall-cmd --direct --add-rule ipv4 filter FORWARD 0 -i eth_int -o
eth_ext -j ACCEPT
firewall-cmd --direct --add-rule ipv4 filter FORWARD 0 -i eth_ext -o
eth_int -m state --state RELATED,ESTABLISHED -j ACCEPT
```

Source: https://www.centos.org/forums/viewtopic.php?t=53819

Firewall configuration

Open the firewall configuration GUI.

We need to configure the firewall to let a bunch of services let through. The profile we are going to use is the one named `internal`.

Then in this profile on the tab *Services* tick the following names:

```
dhcp
dhcpv6-client
dns
ftp
http
https
iscsi-target
mdns
mountd
nfs
ntp
rpc-bind
rsyncd
samba
```

```
samba-client
squid
ssh
tftp
tftp-client
```

And in the *Masquerading* tab tick the showed box.

Lastly in the `options` dropdown select the option `Runtime to Permanent`, this way the changes are saved.

Sixth Step - iCBD-imgs

We are close to the end of the configurations on the *iCBD-imgs* server!

Launch the need services

There are some key services that need to be running in order to the platform work.

Make sure that these services are successfully running:

```
systemctl start vmware
systemctl start vmware-workstation-server
systemctl start libvirtd
systemctl start dnsmasq
systemctl start tftp * NO NEED
systemctl start tgtd
systemctl start nfs-server
systemctl start httpd
systemctl start ntpd
```

Check with `systemctl status -l [service_name]`

Don't forget to enable them for when a restart occur:

```
systemctl enable vmware-workstation-server
systemctl enable libvirtd
systemctl enable dnsmasq
systemctl enable tftp
systemctl enable tgtd
systemctl enable nfs-server
systemctl enable httpd
systemctl enable ntpd
```

Other Roles Services

iCBD-rw

iCBD-rw sub volumes

```
btrfs subv create /var/lib/Home
btrfs subv create /var/lib/icbd
btrfs subv create /var/lib/icbd/.snap
btrfs subv create /var/lib/icbd/nfs_home
btrfs subv create /var/lib/icbd/nfs_root
btrfs subv create /var/lib/icbd/nfs_rw
btrfs subv create /var/lib/icbd/nfs_tmp
btrfs subv create /var/lib/icbd/rw
mkdir /var/lib/icbd/mounts
btrfs subv create /var/lib/icbd/mounts/tftpboot
```

iCBD-rw Services

```
systemctl start tgtd
systemctl start nfs-server
```

iCBD-rw fstab

```
/dev/sdb        /var/lib        btrfs   subvol=Lib      0 0
/dev/sdb        /home   btrfs   subvol=Home     0 0

/var/lib/icbd/nfs_home  /var/lib/icbd/exports/nfs_home  none    rbind   0 0
/var/lib/icbd/nfs_root  /var/lib/icbd/exports/nfs_root  none    rbind   0 0
/var/lib/icbd/rw        /var/lib/icbd/exports/rw        none    rbind   0 0
/var/lib/icbd/mounts/etc/hosts  /etc/hosts      none    bind    0 0
/var/lib/icbd/mounts/etc/exports        /etc/exports    none    bind    0 0
/var/lib/icbd/mounts/tftpboot   /var/lib/tftpboot       none    rbind   0 0
/var/lib/icbd/mounts/etc/tgt    /etc/tgt        none    rbind   0 0
/var/lib/icbd/mounts/etc/httpd  /etc/httpd      none    rbind   0 0
/var/lib/icbd/mounts/etc/tgt/macs.d     /var/lib/icbd/exports/macs.d
none    rbind   0 0
```

iCBD-home

iCBD-home sub volumes

```
btrfs subv create /var/lib/icbd
btrfs subv create /var/lib/icbd/.snap
btrfs subv create /var/lib/icbd/nfs_home
btrfs subv create /var/lib/icbd/nfs_root
btrfs subv create /var/lib/icbd/exports/nfs_home
btrfs subv create /var/lib/icbd/exports/nfs_root
```

iCBD-home fstab

```
/dev/sdb        /var/lib        btrfs   subvol=Lib      0 0
/var/lib/icbd/mounts/etc/exports        /etc/exports    none    bind    0 0
```

iCBD-home Services

```
systemctl start nfs-server
```

iCBD-Cache

In the file `/etc/hosts` there is the need to change one line. Where is

`10.0.2.251 imgs.icbd.local boot.icbd.local root.icbd.local adm-s.icbd.local`

now we should have two lines:

`10.0.2.251 imgs.icbd.local`

`10.1.2.251 boot.icbd.local root.icbd.local adm-s.icbd.local`

The second IP is the subnet to be used on the second NIC of the cache server, and only to communicate with clients.

iCBD-cache sub volumes

```
btrfs subv create /var/lib/icbd
btrfs subv create /var/lib/icbd/.snap
btrfs subv create /var/lib/icbd/shared-vms
mkdir /var/lib/icbd/mounts
btrfs subv create /var/lib/icbd/mounts/vmware
btrfs subv create /var/lib/icbd/mounts/livirt
btrfs subv create /var/lib/icbd/mounts/tftpboot
btrfs subv create /var/lib/icbd/nfs_home
btrfs subv create /var/lib/icbd/nfs_root
btrfs subv create /var/lib/icbd/rw
btrfs subv create /var/lib/icbd/iso
btrfs subv create /var/lib/icbd/tmp
btrfs subv create /var/lib/icbd/icbd
```

iCBD-cache fstab

```
/dev/sdb        /var/lib        btrfs   subvol=Lib      0 0

/var/lib/icbd/mounts/vmware             /var/lib/vmware none    rbind   0 0

/var/lib/icbd/mounts/etc/iscsi  /etc/iscsi      none    rbind   0 0
```

```
/var/lib/icbd/mounts/etc/tgt            /etc/tgt        none    rbind   0 0
/var/lib/icbd/mounts/etc/httpd  /etc/httpd      none    rbind   0 0
/var/lib/icbd/mounts/etc/xinetd.d       /etc/xinetd.d   none    rbind   0 0
/var/lib/icbd/mounts/tftpboot           /var/lib/tftpboot       none
rbind   0 0

/var/lib/icbd/mounts/etc/hosts  /etc/hosts      none    bind    0 0
/var/lib/icbd/mounts/etc/exports        /etc/exports    none    bind    0 0
/var/lib/icbd/mounts/etc/dnsmasq.conf   /etc/dnsmasq.conf       none
bind    0 0


/var/lib/icbd/icbd      /var/lib/tftpboot/icbd          none    rbind   0 0

/var/lib/icbd/bin       /var/lib/icbd/exports/bin       none    rbind   0 0
/var/lib/icbd/include   /var/lib/icbd/exports/include   none    rbind   0 0
/var/lib/icbd/client    /var/lib/icbd/exports/client    none    rbind   0 0

/var/lib/icbd/icbd      /var/lib/icbd/exports/icbd      none    rbind   0 0
/var/lib/icbd/tmp       /var/lib/icbd/exports/tmp       none    rbind   0 0
/var/lib/icbd/iso       /var/lib/icbd/exports/iso       none    rbind   0 0

/var/lib/icbd/shared-vms        /var/lib/icbd/exports/shared-vms
none    rbind   0 0
/var/lib/icbd/nfs_home  /var/lib/icbd/exports/nfs_home  none    rbind   0 0
/var/lib/icbd/nfs_root  /var/lib/icbd/exports/nfs_root  none    rbind   0 0
/var/lib/libvirt/images /var/lib/icbd/exports/images    none    rbind   0 0

home.icbd.local:/nfs_home       /var/lib/icbd/nfs_home  nfs4    _netdev,rw
     0 0
home.icbd.local:/nfs_root       /var/lib/icbd/nfs_root  nfs4    _netdev,rw
     0 0
data.icbd.local:/rw     /var/lib/icbd/rw        nfs4    _netdev,rw      0 0
data.icbd.local:/rw     /var/lib/icbd/exports/rw        nfs4    _netdev,rw
     0 0
data.icbd.local:/macs.d /etc/tgt/macs.d nfs4    _netdev,rw      0 0
```

iCBD-cache Services

```
systemctl start libvirtd
systemctl start dnsmasq
systemctl start tftp *NO NEED - USE DNSMASQ*
systemctl start tgtd
systemctl start nfs-server
systemctl start httpd
systemctl start ntpd
```

Change Log

2017-11-21 — Version 0.0.1 — Creation of this document.

2017-12-01 — Version 0.0.1 — Created the base structure for the description of the installation steps.

2017-12-10 — Version 0.0.1 — Added much of the content for the installation of the three main VMs. Some organisation is needed!

2017-12-12 — Version 0.0.1 — Step One formatted and updated.

2017-12-16 — Version 0.0.1 — Reference added.

2017-12-18 — Version 0.0.1 — Step Two edited.

2018-01-12 — Version 0.0.1 — Every step was edited

2018-01-14 — Version 1.0.0 — All steps tested in the installation of one physical cache server

2018-01-30 — Version 1.0.1 — Some clarifications on the introduction and on the cache server.

2018-08-15 — Version 1.0.1 — Removed email from Reditus

2018-10-09 — Version 1.0.1 — Added instructions on setting up RSA keys for the apache user.

References

CentOS 7 Documentation - Enable or Disable SELinux

HowToForge - A Beginners Guide To btrfs

ANNEX III

Bug on BTRFS affecting CoreUtils tool

III.1 Bug Report

The bug was reported in both *CentOS Bug Tracker* and *Red Hat Bugzilla* on 4 of December of 2017.

Description of problem: In a CentOS 7 VM with kernel `3.10.0-693.5.2.el7.x86_64` including a mounted disk formatted with BTRFS and btrfs-progs v4.9.1. When executing a copy of files with the command `"cp --reflink=always"` the command fails with the indication *"failed to clone 'someFile': Operation not supported"*

Issue: The above-mentioned copy fails, but in the files are created with zero bytes in the destination folder. While trying to find the cause, it seems to be some bug in the ioctl operation. A strace log of the copy operation can be found in the uploaded files.

This problem only manifests itself in the kernel `3.10.0-693.5.2.el7.x86_64`. If the same operation is executed in the system with the kernel `3.10.0-514.2.2.el7.x86_64` all goes well. More evidence that this is probably a bug introduced in this version of the kernel can be found in the git repository in the first commit of the new kernel [1]. In the file "SPECS/kernel.spec"line 15011 [2] that some changes were made in the ioctl - *"[fs] btrfs: fix uninit variable in clone ioctl (Bill O'Donnell) [1298680]"*

As a workaround, an older version of the kernel can be used, but this is not optimal, as future releases may have the same problem.

How reproducible: It is always reproducible. Happens every time.

[1] https://git.centos.org/commit/rpms!kernel.git/d6bfd60741b14479a15b43acaa1ea5a8d73df543
[2] https://git.centos.org/blob/rpms!kernel.git/d6bfd60741b14479a15b43acaa1ea5a8d73df543/SPECS!kernel.spec#L15011

Steps to Reproduce:

1. In a BTRFS mount execute:
2. `dd if=/dev/urandom of=testb bs=1024k seek=1024 count=128`
3. `cp --reflink=always testb testb_copy`

Actual results: The copy operation returns with "failed to clone 'testb': Operation not supported"and creates a zero bytes file in the destination of the copy.

Expected results: A clone of the file, looking precisely the same as the original with the same size.

III.2 Resolution

On 5 of April of 2018, the problem was acknowledged by Red Hat, and a patch was provided. However, also informed that the fix would not be included in later RHEL7 releases, due to Red Hat deciding to deprecate BTRFS as of RHEL7.5.

```
execve("/usr/bin/cp", ["cp", "--reflink=always", "testb", "test_reflink"], [/* 33 vars */]) =
(...)
access("/etc/selinux/config", F_OK)     = 0
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=106070960, ...}) = 0
mmap(NULL, 106070960, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f43ee219000
close(3)                                = 0
open("/usr/share/locale/locale.alias", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=2502, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f43f59db000
read(3, "# Locale name alias data base.\n#"..., 4096) = 2502
read(3, "", 4096)                       = 0
close(3)                                = 0
munmap(0x7f43f59db000, 4096)            = 0
open("/usr/lib/locale/UTF-8/LC_CTYPE", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or direct
geteuid()                               = 0
stat("test_reflink", {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
stat("testb", {st_mode=S_IFREG|0644, st_size=1207959552, ...}) = 0
stat("test_reflink", {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
open("testb", O_RDONLY)                 = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=1207959552, ...}) = 0
open("test_reflink", O_WRONLY|O_TRUNC)  = 4
fstat(4, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
ioctl(4, BTRFS_IOC_CLONE or FICLONE, 3) = -1 EOPNOTSUPP (Operation not supported)
open("/usr/lib64/charset.alias", O_RDONLY|O_NOFOLLOW) = -1 ENOENT (No such file or directory)
write(2, "cp: ", 4)                     = 4
write(2, "failed to clone 'test_reflink' f"..., 43) = 43
write(2, ": Operation not supported", 25) = 25
write(2, "\n", 1)                       = 1
close(4)                                = 0
close(3)                                = 0
lseek(0, 0, SEEK_CUR)                   = -1 ESPIPE (Illegal seek)
close(0)                                = 0
close(1)                                = 0
close(2)                                = 0
exit_group(1)                           = ?
+++ exited with 1 +++
```

Listing 5: Strace of the cp `--reflink=always` command

```
--- a/fs/btrfs/super.c
+++ a/fs/btrfs/super.c
@@ -2191,7 +2191,7 @@ static struct file_system_type btrfs_fs_type = {
         .name                = "btrfs",
         .mount                = btrfs_mount,
         .kill_sb        = btrfs_kill_super,
-        .fs_flags        = FS_REQUIRES_DEV | FS_BINARY_MOUNTDATA,
+        .fs_flags        = FS_REQUIRES_DEV | FS_BINARY_MOUNTDATA | FS_HAS_FO_EXTEND,
 };
 MODULE_ALIAS_FS("btrfs");
```

Listing 6: BTRFS patch on a/fs/btrfs/super.c

ANNEX IV

iCBD Cluster Rack Diagram

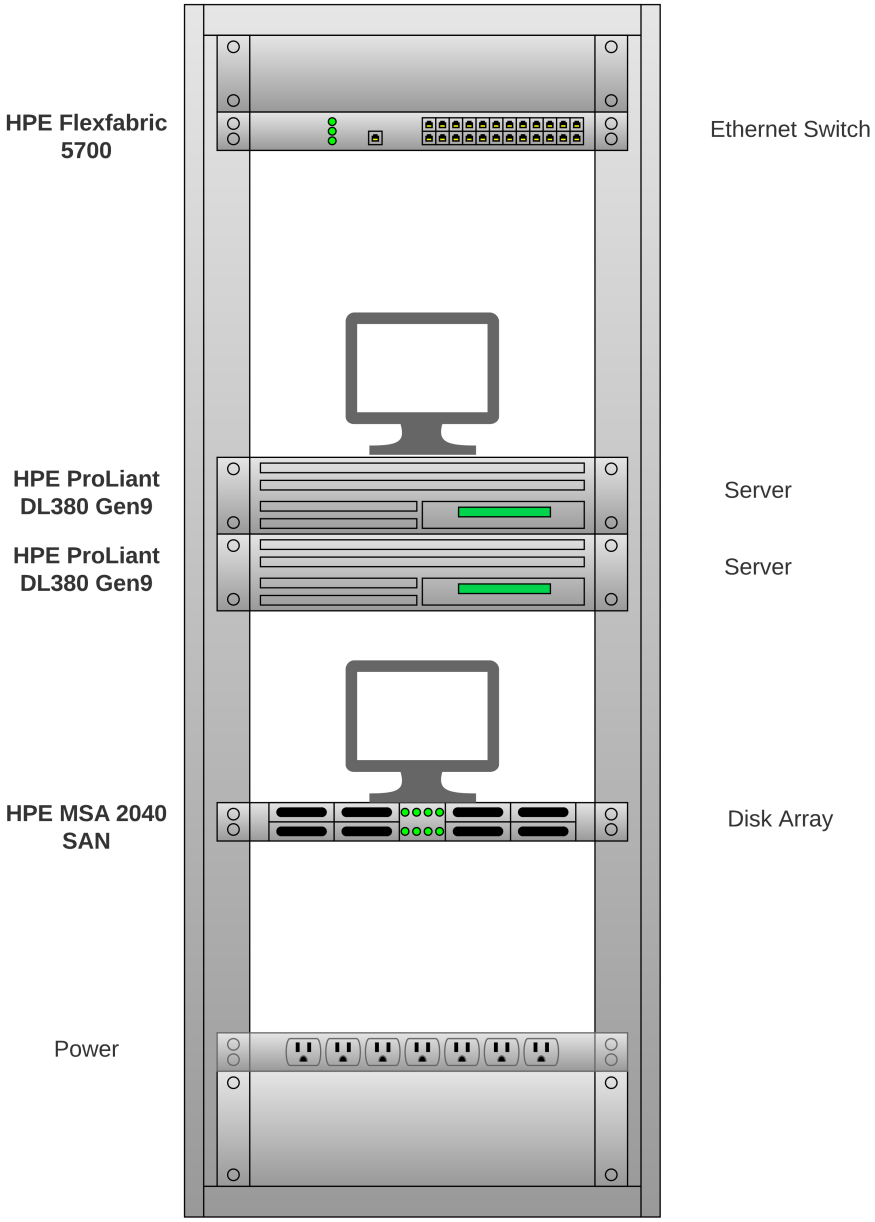

Figure IV.1: iCBD Cluster Rack Diagram