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.

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

Over the span of a few years, there were fundamental changes in the way computing power is used. The heightening of virtualisation changed the infrastructure model of a *data centre* and the way physical computers are managed. This shift is the result of enabling for fast deployment of Virtual Machines (VM) in a high consolidation ratio environment and with minimal need for management.

New approaches to virtualisation techniques are being developed at a surprisingly fast rate, which leads to an exciting and vibrating ecosystem of platforms and services seeing the light of day. We see big industry players engaging in such problems as *Desktop Virtualisation* with moderate success, but completely ignoring the computation power already present in their clients, instead, opting for a costly solution of acquiring powerful new machines and software. There is still space for improvement and the development of technologies that take advantage of the onsite computation capabilities with minimum effort on the configuration side.

This thesis focuses on the development of mechanisms for the replication and caching of *VM* images stored in a local file system, albeit one with the ability to perform snapshots. There are some particular items to address: like the solution needs to follow an entirely distributed architecture and fully integrate with a parallel implemented client-based Virtual Desktop Infrastructure (VDI) platform; needs to work with very large read-only files some of them resulting from the creation of snapshots while maintaining some versioning features. This work will also explore the challenges and advantages of deploying such a system in a high throughput network, maintaining high availability and scalability properties while supporting a broad set of clients efficiently.

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, a forma como são geridas as máquinas físicas e os modelos de infraestruturas num centro de dados sofreu grandes alterações. Esta mudança é o resultado de uma procura por uma forma de disponibilizar rapidamente uma *VM* num ambiente altamente consolidado e com necessidades mínimas de intervenção para a 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 seus PCs cliente, optando, por uma via de custo elevado, adquirindo máquinas poderosas e *software* variado. Existe ainda espaço para melhores soluções e para o desenvolvimento de tecnologias que façam uso das capacidades de computação que já se encontrem presentes, 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, alguns deles resultantes da criação de *snapshots*, de vários GB e acedidos em leitura, mantendo ainda 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

DaaS Desktop as a Service.

DHCP Dynamic Host Configuration Protocol.

HTTP Hypertext Transfer Protocol.

IaaS Infrastructure as a Service.

iCBD Infrastructure for Client-Based Desktop.

iMI iCBD Machine Image.

iSCSI Internet Small Computer Systems Interface.

KVM Kernel-based Virtual Machine.

NFS Network File System.

PXE Preboot Execution Environment.

TFTP Trivial File Transfer Protocol.

VDI Virtual Desktop Infrastructure.

VM Virtual Machine.

VMM Virtual Machine Monitor.

Chapter 1

Introduction

1.1 Context

The concept of virtualisation, despite all the recent discussion, isn't new. In reality, this technology has been around since the 1960s [10], but it was not until the development of virtualisation technologies for the x86 architecture [2] and the introduction of *Intel VT* [45] and *AMD SVM* [21] in the 2000s that virtualisation has entered the mainstream as the go-to technology solution for server deployment across many production environments.

With efficient techniques that take advantage of all available resources, and a lowering price point on hardware, an opportunity for the advance of new application models and a revamp in the supporting infrastructure was generated.

However, companies realised that the cost to run a fully fledged *data centre* in-house is unreasonable and a cumbersome task. Not only taking into account the cost of the hardware, but factoring in the many requirements like the cooling systems that take care of the heat generated by the running machines, physical security to protect the rooms, fire suppressing systems in case of emergency, people to maintain the infrastructure, all added, result in considerable costs on a monthly basis. Adding to this, the demand for instantaneous access to information and the extensive resources needed to store it does not stop growing.

This fact created an opening for a Infrastructure as a Service (IaaS) [36] model, outsourcing all the responsibilities of storing the data and providing the needed computation resources from third parties, which are experts in maintaining huge data centres and even provide all this in various geographic regions.

With influential industry players following this trend, supporting more and more types of services and with an increasing number of customers joining this model, new

ways to store the growing number of files have emerged. New file systems with a focus on reliability, consistency, performance, scalability, all in a distributed architecture are essential to a broad range of applications presenting a myriad of workloads.

1.2 Motivation

Virtualisation is the pillar technology that allowed for the widespread of the IaaS cloud providers in a model of economies of scale. These cloud providers, such as Amazon AWS [7], Microsoft Azure [37] and Google Cloud Platform [18], manage thousands of physical machines all over the globe, with the majority of the infrastructure being multi-tenant oriented.

The sheer magnitude of those numbers leads to an obvious problem. How to store all this data efficiently? Not only there is the need to store client generated data but also manage all the demands of the infrastructure and the many services offered. One approach taken by these companies was the development of their proprietary storage solutions. For instance, Google uses BigTable distributed storage system [11], to store product 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, designed to be reliable, scalable and fault tolerance.

One characteristic in particular that stands out and is present in many of today's systems is the use of snapshots with copy-on-write techniques. The adoption of such methods allows for quick copy operations of large data sets but saving resources. At the same time, it provides high-availability with read-only copies of the data always ready to use and allowing applications to continue execution of write operations simultaneously. All the above-mentioned properties joined with others such as replication and data distribution, to comprise the fundamentals of what is needed to run a highly distributed and scalable file system. For instance, the duplication of records across multiple machines, not only serves as a security net in case of a misfortune event avoiding having a single point of failure but can also be used to maximise availability and take advantage of network bandwidth.

One of these newer systems that have a significant adoption by the Linux community is the BTRFS [46]. At the start, this file system already adopts an efficient system of snapshots, and it has as a primary design principle to maintain excellent performance in a comprehensive set of conditions. The combination of this file system with replication and partitioning techniques opens the way to a solution that serves the needs of an up to date storage system, consequently having the possibility of being easily integrated into an existing platform, serving a vast number of clients and presenting outstanding performance.

1.3 Project Presentation

This dissertation work is performed in the context of a more comprehensive project with the name Infrastructure for Client-Based Desktop (iCBD) [34], under development at *SolidNetworks – Business Consulting, Lda* part of the *Reditus S.A.* group in collaboration with *NOVA LINCS* hosted at *DI - FCT/NOVA*.

The primary objective is to improve in a known model, the client-based Virtual Desktop Infrastructure, developing an infrastructure to support the execution, in a non-intrusive way, of virtualised desktops in conventional workstations.

1.3.1 iCBD Project

There are some leading-edge aspects of the Infrastructure for Client-Based Desktop (iCBD) project which sets it apart from other existing solutions such as the adoption of a diskless paradigm with a remote boot, the way the platform stores Virtual Machine (VM) images, and the support for a virtualised or native execution on the target workstation [1], depending on the user's choice. [33]

The remote boot of the users workstation requires the cooperation of HTTP, TFTP, and DHCP and image repository servers, that store VM templates as well as running instances based on them. To address the process of communication between workstations and the platform it is used the HTTP protocol, providing flexibility and efficiency in the communication of the messages. [3, 33, 35]

It is also interesting to briefly discuss some of the primary objectives of the project, being:

- Offer 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.
- Enable centralised management of the entire infrastructure including servers in their multiple roles, storage and network devices from a single point.
- Complete decoupling between users and workstations in order to promote mobility.
- Support the disconnected operation of mobile workstations.

With all the above in account, there is a clear separation from other solutions previously and currently available. As far as we know, no other solution is so comprehensive in the use of the resources offered by workstations whether they are PCs, laptops or similar devices.

[1] In this document workstation is a user's desktop PC, laptop, etc., any PC compatible computing device.

1.3.2 Previous Work

There have previously been two dissertations involved in this project. The work of those students has centred in the creation of the instances of virtual machines, more specifically in a creation supported by native snapshot mechanisms of the file system where the templates reside. Testing several Files Systems that present the requirements mentioned above. As well as, conduct an analysis on the performance of those systems when working with a typical load for the platform. In a few words, the work opens a path where the snapshots stop being tied to the hypervisor as a tool for the provisioning (thin or full) of clones and bestowing the job to the file system snapshot system, which is fundamental to deliver a centralised management solution. As is happening presently, the two theses have followed two different paths in an attempt to determine which type of file system best suits these objectives. In that sense, one of the works was used a local file system, named BTRFS, as the other followed the object-based storage path, adopting the CephFS.

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 the management of templates across multiple "zones", simple?*
- *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 answer these questions, we can formulate small topics that guide the work in a more focused way creating a list of general requirements to be addressed by this thesis. In general, the work will be divided into two major subjects, were the second is a direct consequence and requires the first.

Motivation and Goals Providing a mechanism that ensures the correct replication of data between nodes of the platform is paramount. This process also needs to confer other properties such as achieving the best performance on data transferences, being in a factor of speed or the used bandwidth. Another point of interest is the fact that the replication process should easily integrate with any File System used in the Storage Layer (taking into account that is fundamental that the file system provides some variety of snapshotting mechanism) trying not to add a sizeable computational load. Following this idea, we may start thinking on how to deliver to the clients the benefits of replicating data.

The replication procedure is not only a useful tool to propagate efficiently data changes throwout the nodes of the platform and act as insurance from a data loss disaster, but also capitalise on getting closer to the clients the resources that they need. Following this line of thought, an answer to the second question starts to appear. It is not only necessary to think about the location of the data, but also to study which services are used by the platform that are fundamental for running images on the client's workstations and how to integrate all so as to deliver the best experience to the highest number of clients. To summarise, the following list poses the key requirements had in mind during the planning and execution of this effort:

- The iCBD platform needs to be always available and maintain top-notch performance in multiple locations while serving a considerable number of clients.
- Produce a mechanism that allows the replication of data not only for security reasons (backup) but also providing that data closer to the client.
- With the data near its consumer study what is required to deliver that data to the client in the most efficient way.
- Boot a client with the minimal number of the platform functionalities possible, simplifying the processes near the consumer.

1.4.2 Main Expected Contributions

The main expected contributions are:

- Perform a throughout analysis of the already implemented iCBD platform modules and the several layers where it expands in order to understand the inner works, providing an excellent platform to build upon and allowing for an efficient planning of the remaining work.
- The study, develop, and evaluate an implementation of a distributed and replicated BTRFS file system for VM storage.
- 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 work previously developed and the existing infrastructure
- And finally, carry out a series of tests that lead to a meaningful conclusion and that provide help in the design of the remaining platform.

In a more concrete approach, we present the work plan, for achieving the above objectives, distributed by the two main topics.

Replication

- Create a middleware that integrates with the core functionalities already developed within the iCBD platform.
- Should aim to be storage provider agnostic (in this thesis work with BTRFS but remain easy to integrate with others).
- Work with compression algorithms to achieve a lower bandwidth consumption.
- Be able to use encrypted and unencrypted communications for all data flows.
- Capitalize the snapshotting features of the storage layer in order to minimise the volume of data transferred, only sending the differences between versions whenever possible.
- Provide a simple CLI and a REST API for interacting with the replication module functions.

This subject is further developed in **Section 4.2**.

Caching

- Dilute some cost of the infrastructure by having commodity hardware as proximity servers.
- Bring the data closer to the final clients giving enough storage capacity to servers, storing the most used images.
- Facilitate a user experience where the is no distinction on booting an OS from a local disk or the iCBD platform.
- Build a completing working iCBD platform on the FCT NOVA campus.
- Study the benefits of introducing cache servers and the platform performance as a whole, comparing the performance of the system to a traditional OS install base.

A detailed view of this work is presented in **Section 4.3**.

1.5 Document Structure

The remnant document is structured as follows:

- *Chapter 2* **Research Context** - This section presents existing technologies and theoretical approaches which were the target of study, such as, storage systems and several of its features, as well as several intrinsic characteristics of virtualisation techniques.

- *Chapter 3* **iCBD - Infrastructure for Client-Based Desktop** - In this chapter, there is a presentation of the iCBD platform as a whole. Giving also an overview of the solution with its multiple layers and trying to explain the conceptual and architectural decisions made, being essential to understanding the bigger picture and where the remaining work fits in.

- *Chapter 4* **Implementation of the *iCBD-Replication and Cache Server*** - Starts giving an in-depth view of the implementation of the iCBD Replication module, detailing the architectural decisions and the implemented components. Then, follows a description on the efforts to build a test infrastructure on the FCT NOVA grounds, solely dedicated to the iCBD project. Concluding the chapter with an explanation on how to build and deploy an iCBD Cache Server node that can serve the entirety of workstations on a laboratory in the Computer Science Department.

- *Chapter 5* **Evaluation** - The evaluation process employed in the validation of our implementation is presented in this chapter. Emphasizing the results of the work developed, analysing the results obtained and comparing with baseline values.

- *Chapter 6* **Conclusions & Future Work** - This last chapter concludes the dissertation by answering the questions stated in the Introduction, summarising the results achieved in the evaluation process and presents some improvements ideas that wore formulated during the implementation process and are believed to be 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 [42], 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). It is the VMM which has that task and should do it efficiently at the same time providing this three properties [42]:

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: much of the instructions in the virtual machine should be executed directly by the real processor without 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 a 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:

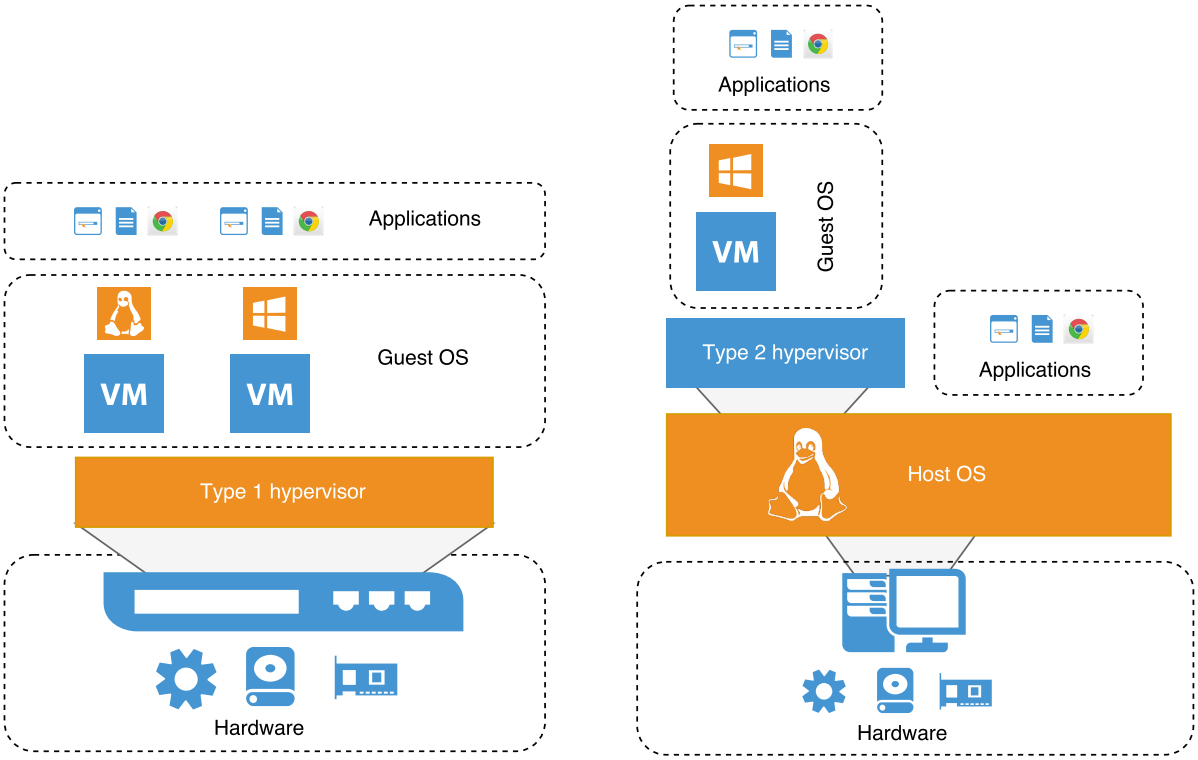

Figure 2.1: Virtualization architecture with type 1 and type 2 hypervisors

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 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.

Recognizing 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 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 Kernel-based Virtual Machine (KVM) [32] 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 for software testing, on users one workstation dismissing the requirement for a dedicated test machine.

Either way, the challenge lays in the fact that the hypervisor needs to execute the guests OS instructions in a safe manner and at the same time provide possible different machine configurations to each 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 components reside in a VM configuration file. In the case of VMware hypervisors, the file has the `.vmx` extension,[43, 54] while in a KVM environment, that configuration is stored in a `.xml` file. [12]

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 with the recourse to cloning techniques. Software updates can be hugely simplified and made available to all VMs at once if those VMs are created on-demand 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 configurations, different OSs and applications needs. Probably even supporting several versions of the same software is required for the day to day operations.

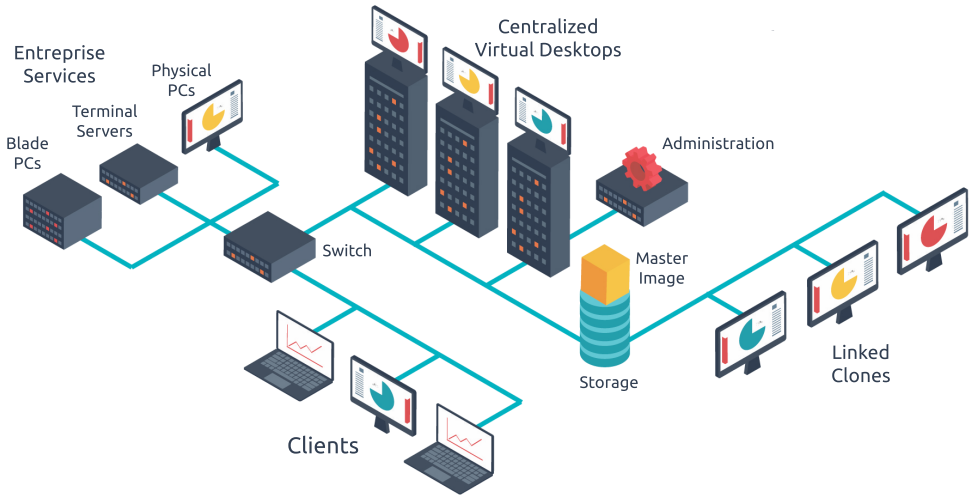

Figure 2.2: An exemple of a Virtual Desktop Infrastructure, adapted from AppDS [9]

Organisations debate themselves 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.
- A delay on the installation or reinstallation of new software and recovery from breakdowns.
- 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 [40]) 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, and 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 implementation with more relevance 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 [22]. Typically this solution offers a centralised architecture, where the user's environment resides on a server in a data centre, as shown on 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. 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 mentioned above.

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) [44] or the Remote Framebuffer Protocol (RFB) [50] 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 [52], XenDesktop from Citrix [56] and Microsoft with Microsoft Remote Desktop [39].

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. [13]. An example is a previously existing solution by Citrix, the XenClient [56]

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. [57] 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.

In this new field, a multitude of solutions is emerging with public cloud providers

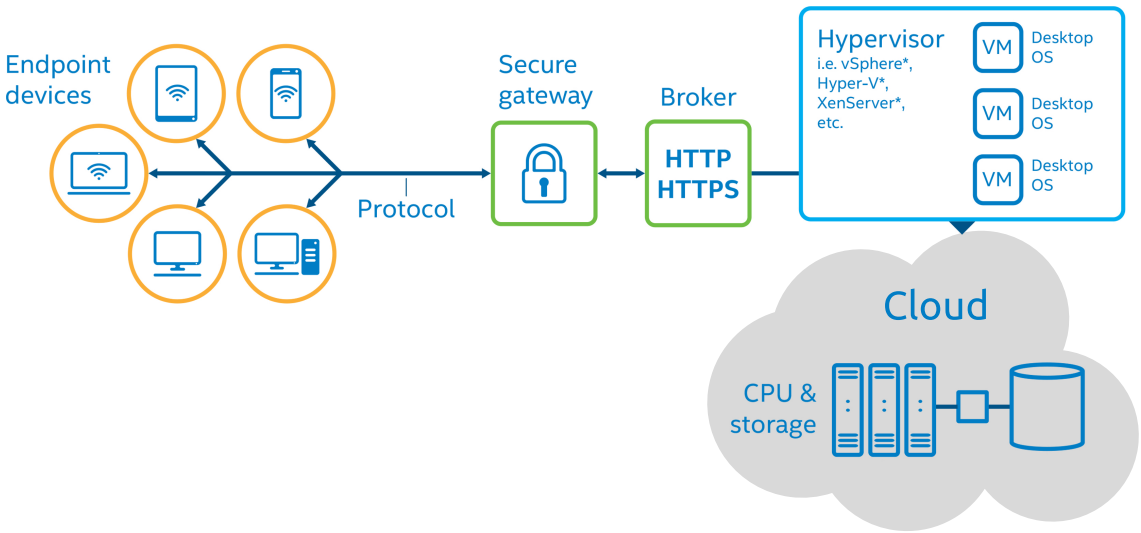

Figure 2.3: Conceptual overview of DaaS architecture, adapted from Intel [27]

leading the way. Amazon in its AWS portfolio delivers Amazon Workspaces [6], and Microsoft implements the RDS features [38] on the Azure product line. Nevertheless, there are also some smaller contenders, as an example, Workspot [55] (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 [54], 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 [24]

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. [14] We can give as example of file systems the well-known Network File System (NFS) [47] originally developed by Sun Microsystems, and the notable Andrew File System (AFS) [48] 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 [53], OpenStack Swift [49], and in a IaaS flavour the Amazon S3 [4] and Google Cloud Storage [16].

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 iCBD stands for Infrastructure for Client-Based (Virtual) Desktop (Computing). Is 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* part of the *Reditus S.A.* group.

The primary goal is to implement a client based VDI, a specialised form of VDI infrastructure, where client's computations are performed directly on the client hardware as opposed to performed on big and expensive servers. With the distinctive characteristic of not having the necessity of investing in hard disks for the client devices, as well as hoping to solve prominent predicaments in the administration and management of large-scale computer infrastructure.

This chapter will address the central concepts and associated technologies encompassed in this project, particularly:

Section 3.1 overviews the core concepts of the project and particularly note the limitations and peculiarities of current implementations in contrast with the chosen 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

The iCBD as a project aims to investigate an architecture that leads to the development of a platform that is similar to a client-based VDI, while maintaining all the benefits of both client-based and server-based VDI. Additionally, it should present the power of working as a Cloud Desktop as a Service (DaaS) without any of the bad traits of the approaches as mentioned earlier.

The 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 platform should be able to:

- Adapting to a wide range of server configurations, without prejudice to the defined architecture.
- Minimize disruption in the use of workstations for end-users, offering a work environment and experience of use so close to the traditional one, that they should not be able to tell it from a 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 workstation cost.
- Maintain an inter-site solution to efficiently support, 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 a network boot and custom-made to the operating system) and an assortment of configurations for services like PXE and iSCSI.

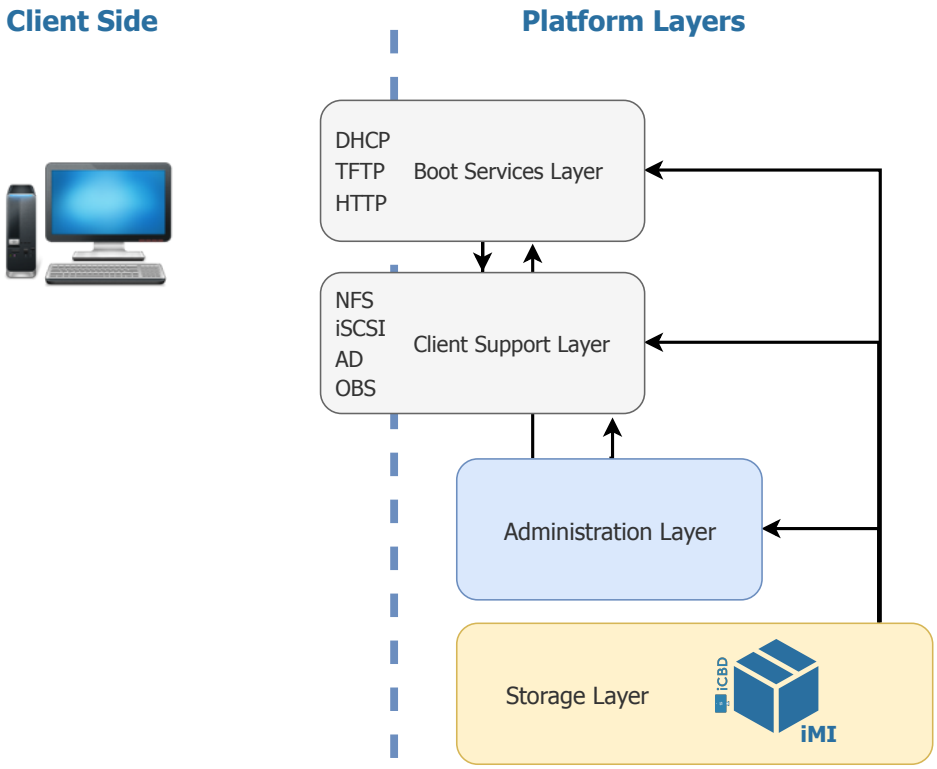

Figure 3.1: iCBD Layers View

Boot Services Layer is responsible for providing the initial process from which the client machines will boot from the network, and later on transfer a bespoke boot package, using services such as PXE, DHCP, TFTP and HTTP.

Administration Layer to maintain all the iMI life cycle, from its creation to its retirement, currently this is done with a custom set of scripts.

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

Storage Layer maintains the repository of iMIs and provides essential operations such as version control of the VM images files. Our work is fundamentally focused on this layer, extending it in such a way that a single repository abstraction is built on top of the local / individual repositories through replication and caching. These local repositories are implemented on BTRFS or CEPH and may be exported to clients using NFS or iSCSI.

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

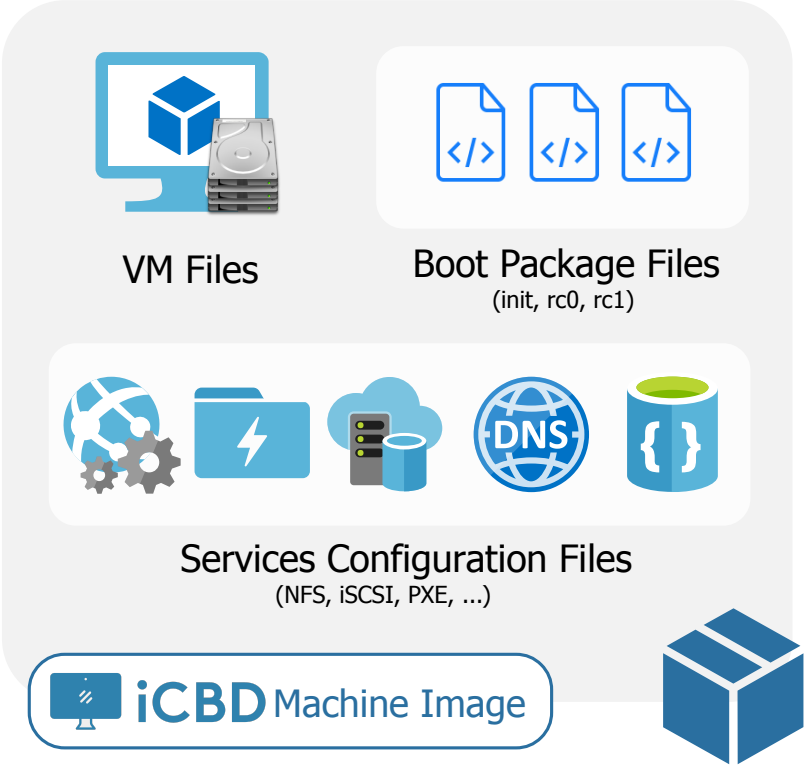

Figure 3.2: iCBD Machine Image Files

3.2.1 iCBD Machine Image

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

VM Template files The main component 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.

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 a workstation; these files can be included in the initial *ramdisk* or later transferred over HTTP when needed. Included are 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 in to the single-user level. With a tool such as *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 Among the services that support iCBD, some do need changes 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 to be set. 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 live deployment in use by multiple clients, and finally its decommission and placement in to temporary or cold storage.

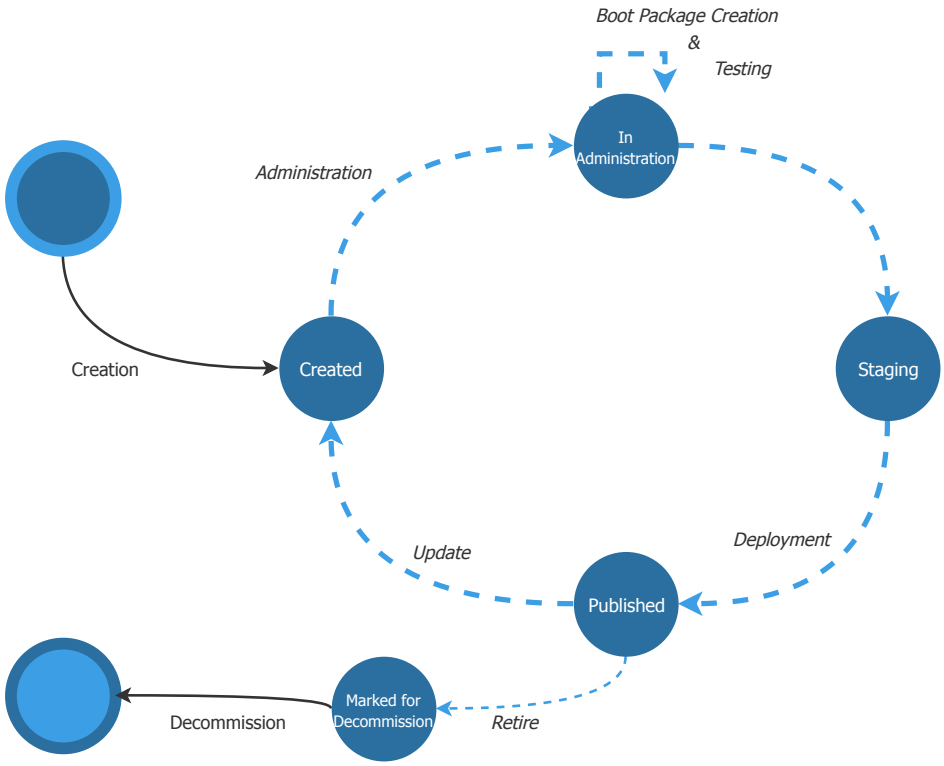

Figure 3.3: iMI Life Cycle inside the iCBD Platform

Each lap in the cycle is considered a new version. So every new update made to an iMI will spawn a new version. Through the snapshotting features of the storage layer, the creation of a new version is a rather straightforward and a computationally light operation.

During its life in the platform, an iMI can present it self into 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 a boot package.

In Administration An iMI goes through this phase in two moments: the first one, described above, where an image has just been injected into the platform. And the

second and most frequent case, where an image needs to be updated or 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 status symbolises that the image is ready to work but isn't yet published and so not visible to platform users. This phase holds a particular interest in the testing the iMI for the correctness in the boot process and to ensure that the modifications were applied. Only after the testing procedures should an iMI made available for general use.

Published The stage where the iMI is expected to spend most of its time. One can think o f this state as the proceeding into production of an iMI. After all the previous steps it is anticipated that the image is entirely ready to be delivered to the clients. At this stage, the platform in its Boot Services Layer announces to clients the possibility of choosing this image and provides the necessary support to its execution. Clients can instantiate the iMI as they please, taking advantage of the fact that can access their data and applications from nearly any device.

When an iMI completes a cycle and undergoes an update process, the old version is retired and goes to a fifth state, denominated **Marked for Decommission**, which is comparable to a stay in limbo. First, because when the administration process has initiated the probability of having clients using the image is significant, therefore the iMI needs to continue available for those clients. Even when the administration process ends, some client may still be using the image's old version. Thus only after all the clients cease utilising the iMI, can the image be transferred to its final state - **Decommission**. At this point, the version can be removed entirely from the platform or more wisely stored as a backup for some eventual failure in the future, or even if the administrator wants to recover an older state of the image.

3.2.2 Boot Services Layer

From an end-user perspective, the only layer that is visible and interactive is the boot layer. The interface is lean and provides a way to select the image to boot in the workstation, however not every single aspect is noticeable. In the background, there is a need to resort to multiple services for starting a client's workstation with an iCBD Machine Image.

The platform provides two processes to remote boot an iMI. One instantiates, from an iMI, the Operating System natively on the bare metal workstation in the fashion of a standard diskless network boot. The other uses the above mechanism for provisioning a minimal iMI that was a hypervisor installed and virtualises any other iMI available in the storage layer. Both approaches are entirely transparent to the final user that does not grasp the differences and doesn't know if the working OS is virtualised or running natively.

The first part of the boot process starts like any other network boot, where a series of DHCP requests are used to provide the suitable client network parameters and particularly the location (IP address) of the TFTP server. Then begins the transference of a small network boot manager program. In this traditional PXE boot environment, a friendly looking tailored made graphical menu displays to the user an assortment of choices that announce the different iMIs ready to boot.

Booting an iMI in a Workstation After the selection of an iMI in the PXE boot menu [20] the second-stage boot kicks in. Using *PXELINUX* as a bootloader there is the capability of transferring a compressed Linux Kernal (vmlinuz) and an initial ramdisk (initrd) (REF in comment)through either TFTP or HTTP, is also in this step that some parameters needed during the boot are set with the correct values according to the image picked. After everything loaded into memory, the stage 2 boot loader invokes the kernel image, and after booted and initialised, the kernel starts the first user-space application.

Commonly the first application is called init, and in the particular case of this platform, the init file starts the chain execution of other custom files (`rc0` and `rc1`). Those Run Control scripts configure every single aspect in the Operating System according to the characteristics of physical machine booting. The first step is to reconfigure the network card and obtain connectivity. Then, is determined if there is the need for getting more files indispensable for the remaining boot process if this need exists, then the missing files are transferred. The next script, `rc0`, deals with data volumes and their mounting method (i.e. r/w space, users home directories); in case of using the loading OS as a base for another iMI in virtualisation, some configurations are anticipated and applied. The file system of the underlying iMI is checked to verify if happens to be BTRFS or any other, in the case where BTRFS is adopted the Seeding capability comes into play in this step. After every aspect from the configuration is setup the `switch root` command is deployed moving the already mounted `/proc`, `/dev`, `/sys`, `/tmp` and `/run` to new root and makes this the new root filesystem.

At last, the residual configuration entails the update of the correct time with the NTP service and some last logging of statistics such as the elapsed time of the boot process and the bandwidth used by the sum of all operations.

3.2.3 Administration Layer

One of the most important features provided by the platform, and personified in this next layer, is the ability to perform administration operations on an iMI. That task becomes simplified by the use of a provided image administration tool. Armed with such a mechanism any systems administrator in an organisation can make the changes that understand necessary (stuff like Operating System and software in general updates, add new software, modify configurations) and then publish the image in the platform for widespread use.

The Administration Process The administration tool consists of a series of scripts triggered by the main one called `adm`. Calling this script with the name of one iMI sets off the startup of a VM in a VMware ESXi server with a base image that will support the administration process. Commonly the OS used will be some flavour of Linux (Fedora 27, CentOS 7 or even Ubuntu 16.04 LTS).

The whole process makes extensive use of the snapshotting capabilities of the Storage Layer (whether using BTRFS or CEPH), with no prejudice to the complete system performance. For each iMI, there is a snapshot with an index number that relates to its version and age (i.e. the higher the number the most recent the version is). Multiple versions of an iMI persist stored in directories named by the index of the version. So, is simple to create a new linked clone from the most recent version.

The administration VM, after its boot process, will start a hypervisor (VMware Workstation or KVM) that in turn will get a new linked clone of the most recent version of the iMI to administrate. In this sense, this process makes use of nested virtualisation to achieve its goal, which can result in some slowness (even considering the use of a Type 1 hypervisor), but in theory, all the operations to the new snapshot could be performed on a physical machine using only one level of virtualisation. In this step, the snapshot that is in the management process is in a working directory, a temporary storage area with a limited lifetime to the duration of the procedure. This method serves two proposes. First, all the clients that are using the latest version of the iMI can remain using it with an administrative process running in parallel. The second is the ability to quickly discard all the changes made in the working directory in case of unwanted changes.

Finishing the process, with all the desired actions performed in the iMI, the administrator gets to chose between saving the changes done or discarding them. When choosing to proceed with saving the new version of the image, a process starts by changing the working directory to a permanent one, through a new linked clone of the working snapshot but this time following the naming system so that the version number will name the directory.

Creating the boot package Even after all the process described above, the iMI is not ready to boot, being the next step the creation of the boot package. This phase is the responsibility of the `mki` script which can be called immediately at the end of the administration process by the textttadm tool depending on the type of the iMI. The procedure is different for each type of iMI (Windows or Linux), but with a set of steps in common, being that the Linux iMIs requires a particular number of customisations. The peculiarity of these requirement comes from the fact that Linux iMIs can run natively on a client workstation serving as host for other images. Which in turn entails the creation of custom `initramfs` and `vmlinuz` files, the addition of Kernel drivers and firmware, the tailor-making of Run Control Scriptfiles (rc0 and rc1) that start the network services with a configuration compatible with the platform and mount the correct remote file systems. In the end, is also added a script called `runvm`, that is instrumental in managing the launch

of a virtualised iMI, as well as configure the hypervisor parameters to take the best advantage of the client hardware. However, there is more to the `mki` script. For both flavours of the iMIs, the configurations of the *iSCSI targets* need an update to reflect the new version of the iMI, with the same happening to the *pxelinux* configuration that requires the new paths to the files served to the clients.

3.2.4 Client Support Layer

The Client Support Layer is the most fluid of all layers, containing an aggregation of services (were most can origin from the outside of the platform) working together to provide the environment to a client function correctly. Other essential feature of this layer is its connection to the storage layer, which, using protocols such as NFS and iSCSI, allows the client to provide the necessary data not only to the boot process but also provides read/write space for the temporary storage of changes to an iMI. Moreover, it is through these protocols that the connection to user directories is carried out. It is important to note that there are more essential services maintained and crucial to the provision of one iMI on the iCBD platform, not just the connection between this layer and the storage layer. This is where subjects such as address translation by a DNS or the synchronisation of time by NTP come by.

The second use case of this layer tackles the need to resort to services that aren't directly connected to the iCBD platform but are need to support a client. For example, it may be the case that we deploy a Windows iMI in the infrastructure of a company that already has the support for Microsoft's Active Directory; it is necessary for the iCBD platform to coexist with this type of services already present, or even facilitate the introduction of new services.

3.2.5 Storage Layer

In a way, we can say that the most significant part of our work will focus on this layer. The storage layer is responsible for saving all platform data, whether they are iMIs, databases, virtual disks (.vmdk) from support platform VMs, or code repositories; essentially it is a tier consisting of a set of file systems each with its purpose.

In the interest of this document, we will only focus on the file system that provides the storage of iMIs. Given the uniqueness of this type of data, the file system to use has to present the right features, notably the support for snapshots, being that the choice fell in the BTRFS. It is important to mention that this is not the only FS that supports snapshots. In connection with that, questions similar to those discussed in this document, but with a focus on an object-oriented file system, are being simultaneously worked on the overall scope of the project.

Particular features of BTRFS are broadly exploited by the iCBD platform: sub-volumes; snapshots; cloning of both sub-volumes and snapshots; BTRFS seed device; incremental

backup; just to name a few. These several characteristics achieve several objectives: multiple sub-volumes are used to store different parts of the platform, snapshots are widely used to create a kind of version control between changes of iMIs giving the possibility to access at any moment any of the versions, or even on creating backups of the entire platform. Another important feature is the capability of using BTRFS file systems as seed devices, conceding for the multiple mounting operations of the same file system in read-only mode. Thus allowing multiple clients to use the same iMI.

All these matters should bear in mind that from the point of view of the remaining layers, the type of filesystem or the way in which it implements certain operations, should be entirely transparent to them, interfacing through NFS or iSCSI, for instance. Thus, by attacking the problem of data replication between multiple file systems, transparency as an attribute is likewise fundamental. All of this will be discussed in detail within the next chapter, including an explanation of the various decision-making processes and the illustration of their implementation.

Chapter 4

Implementation of the *iCBD-Replication and Cache Server*

This chapter addresses the implementation of the central topics of this thesis, divided into two major fields. We first start with a section talking about the creation of a middleware system that provides replication features in an integrated way to the iCBD platform. While providing a detailed description of the concept and architectural model, as well as the implementation decisions, made along the accomplishment of this contribution. Next, we show the work done on improving the performance of the platform clients. Displaying how setting up a client-side caching system that stores images adjacent to the consumers obtain that sought enhancements. Concluding in exploring the challenges of recreating the complete platform in a new environment and implementing a real-world scenario at Nova University Computer Science department laboratories.

This chapter is partitioned as follows :

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

4.1.1 Motivation

4.1.2 System Architecture

4.2 Implementation of a Replication Module

One of the central objectives of iCBD is to provide a platform that can be both cloud-centric, with the administration and a portion of the storage burden gathered in a public cloud, or fully hosted on client premises. Either way, it becomes evident that data locality is an important subject, which means that there is the necessity to study how this data will flow between the multiple components of the iCBD platform. As can be imagined, this is a data-intensive platform, boasting multiple storage devices in many networks and an array of consumers demanding that data at any given time.

All these factors allied to the platform architecture result in the need to create a new component, whose chief mission is to ensure that the data is correctly replicated in the appropriate places, maintaining the consistency of the various versions of the iCBD Machine Images stored.

4.2.1 Requirements of the Module

Since the beginning of this work, the file system selected for use in the storage layer was selected, this is due to the fact that, there was already a functioning prototype of the core iCBD platform making use of BTRFS for all data storage matters. The most critical feature for the operation of the platform is that the file system supports snapshots. BTRFS is a modern file system based on the copy-on-write (COW) principle capable of creating lightweight copies of a file. We detailed the importance of this trait in Section 3.2.5.

The condition described above applies only in the choice of the File System, in theory, any File System that supports snapshots can be employed in the platform. That is, in fact, the case with the work developed in a dissertation carried out in parallel to this one, where the focus is the use of an object-oriented file system, in particular, the CEPH File System. Due to this imposition, it is key that this work makes the best use of the BTRFS features, exploring the incremental backup capabilities. More, the replication process should fully integrate with the core platform that already distributes iMIs to clients. Preservation of consistency of the iMIs is also a concern, assuring the distribution of new versions when they are created.

Moreover, it should be taken in account the locality of the data, since the communications could originate and end in the same data centre and the same local network, or happening between different locations that can be in opposite sides of the world. Such aspects as the bandwidth used and the encryption of the data becomes essential to address, requiring the examination of several compression algorithms that can be accommodated

to the way the data is processed and also ways of keeping this data secure by encrypting the communications.

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.

The BTRFS developers provide a userspace set of utilities that can manage BTRFS filesystems, called `btrfs-progs`. Within that set of tools, there is a pair of commands, `btrfs send` [31], and `btrfs receive` [30], that provides the capability to transport data via a stream and employ those differences in a remote filesystem.

The send command facilitates the process of generating a stream of instructions that describe changes between two subvolume snapshots. Also available in the command is the ability to use an incremental mode, where given a parent snapshot that is available in both send and receive sides, only the small delta between snapshots (e.x. *V2* and *V2-1* in fig 4.1) is going to integrate the stream. This feature is outstanding since considerably reduces the amount of information that needs to be transferred to reconstruct the 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.

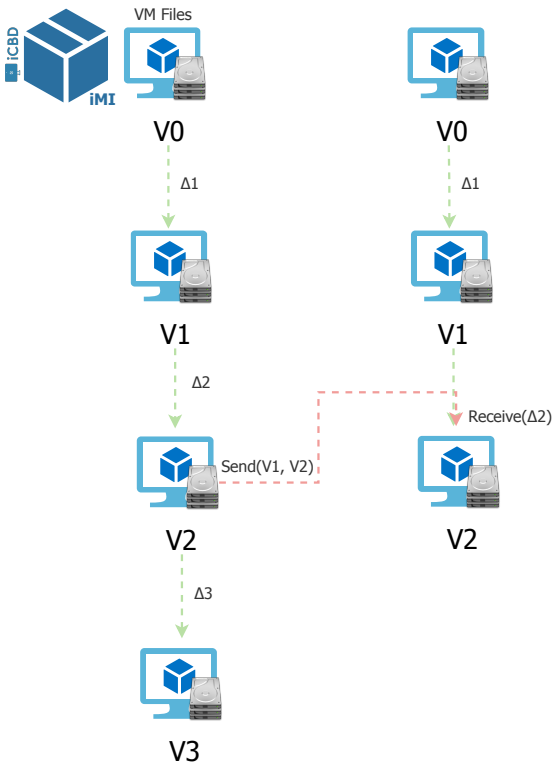

Figure 4.1: iCBD iMI Snapshots Structure

On the remote end, the received command accepts the stream generated by the send command and uses that data to recreate one or more snapshots. Contrary to the send command, receive executes in user space, replaying the instructions that come in the stream one by one, these instructions include the most relevant calls found in a Virtual File System, with Unix system calls like `create()`, `mkdir()`, `link()`, `symlink()`, `rename()`, `unlink()`, `write()`, along with others. [29]

4.2.2 System Overview

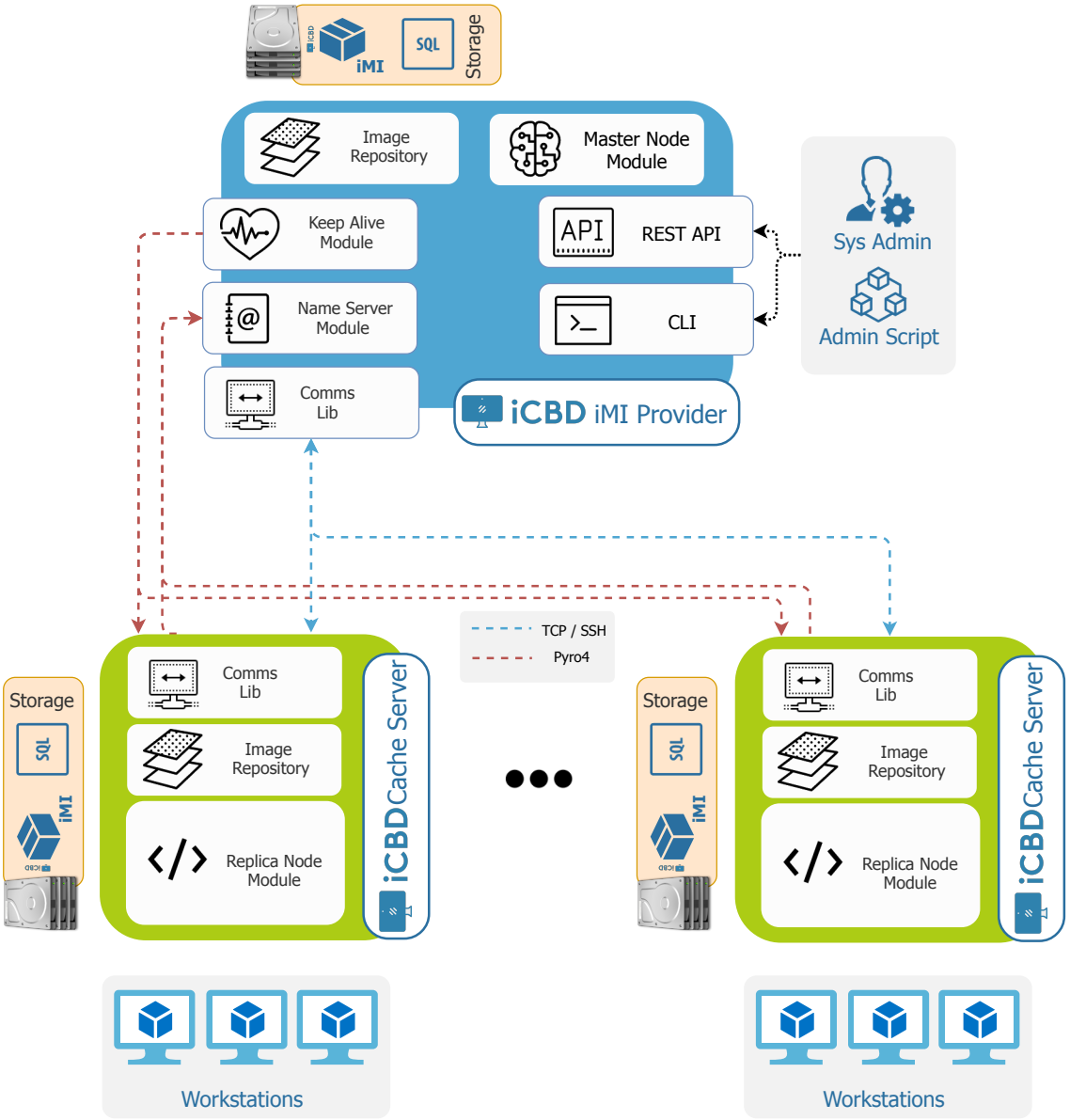

Figure 4.2: iCBD Replication Architecture

Since the replication module needs to interact with several bash scripts, with the core iCBD platform, including command line tools and some operating systems low-level

features, we betted that the most suitable approach was creating a python distributed middleware using a master-replica paradigm.

Python as a programming language enjoys some valid idiosyncrasies, functioning as an object-oriented language, possessing an extensive standard library and enjoys a big community delivering packages with a wide range of functionality, all facts that contribute to make it the best programming language to bundle everything together.

The several modules of the middleware comprise the main functionalities allowing a node to behave as a Master or Replica node, where each one maintains its individual Image Repository. Then there are also a number of libraries that were developed to interface with some tools, as the BTRFS tool talked about above **??**, to interface with an SSH connection, build wrappers for compression algorithms and even provide a REST API.

An overview with a small description of the functionality of each component of the 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. Is required that this node reside on an iCBD Administration machine since this is the node that will send changes made to an iMI to all replicas. It is also responsible for providing a communication interface with the Name Server, in this sense when an administrator wants information about which nodes are present on the platform, or what is their status; these requests obtain a route through the master node.

Replica Node The main task of this node is to maintain the subscribed list of iMIs updated, receiving the last version of those machine images and store them in the file system as soon as the Master Node makes the available. At request, the Replica Node can deliver the list of iMIs that is storing (including and very important the version numbers of the iMIs present in the node). As well as at any time subscribe to a new iMI which from that point forward will include the roster.

Name Server The name server service will run parallel to the Master Node, holding a phone book style record listing all the nodes in the replication system. Nodes register themselves in the name server during the startup process, and at any moment can query the location of another node. All this information is deposited in a simple database.

Keep Alive The keep-alive module consists of a thread, launched by the master node, tasked with periodically check if the replica nodes are still working correctly. When its identified that a replica node stopped responding, this service immediately sends a request to the Name Server for the removal of the dead node from the directory.

Image Repository This module is a custom-made data structure to hold a large set of iCBD Snapshot objects in a `key:value` store. In addition to storing these objects,

provides an interface to quickly know which iMIs are store within the node that accommodates the repository. Every node in the platform (i.e. Master or Replica) must necessarily launch one repository.

So that all the components presented above can function adequately, they require help from several libraries. A description of those implemented libraries ensue:

BTRFS Lib The BTRFS library holds two classes. The first, called `BTRFS Tool` that is a python wrapper for the `btrfs-progs` tool, described in section 4.2.1. The second, designated `BtrfsFsCheck` implements functions to validate if a given path is part of a BTRFS filesystem, case its true is also possible to verify if that path comprises a subvolume. Additionally, this tool is provided with a method to discover all snapshots in a directory.

Compression Lib Since multiple compression algorithms are used, makes all the sense to create a library that encapsulates multiple wrappers, one for each algorithm. These wrappers in reality only need to provide compression and decompression methods for a stream of data, with the remaining operations not present.

Serialiser Lib Some of the communications between nodes require the transmission of objects, to that effect a library containing serialisation and deserialisation methods for those objects is a requirement.

SSH Lib This library implements a wrapper for the SSH command, allowing the creation of a tunnel so that data can be funnelled through a secured connection between nodes in different networks.

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. Also enabling an easy way to expand that interaction to other modules of the platform.

Even though the libraries presented above are not enough for the appropriate implementation of the all functionalities of the replication module, more libraries implemented by the community were necessary to handle aspects like communication between nodes, algorithms for data compression and to secure the data transmission between networks. Some of which we list and describe below (a full list of Python packets imported in the project is shown in the Listing 1), also throughout the remaining text we detail in each module the libraries that help to accomplish its purpose.

4.2.3 Communications between nodes

Coordinate the multiple modules and its activities, demands from the middleware a need to share a communication channel connected through a network. The remote procedure

```
[[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: Pipfile for the iCBD-rep Project

call (RPC) brings support for inter-process communication allowing a procedure on a system to invoke an operation running in a process in a different location, most likely on a remote system.

As seen in figure 4.2, multiple processes are running in different machines any given time, and those processes need to continually send and receive information: perhaps operations to execute, metadata updates, or just monitoring if a process is running according to the desired plan or is in a faulty state. Managing the nodes is a perfect case for the use of the Pyro 4 library, that gives a holistic view of the system and allows triggering a multitude of operations in each node.

Pyro4 Library Pyro 4 [26] is a library that enables the development of python applications in which objects can talk to each other over the network through the use of RPCs. Its designed to be very easy to use and integrate into a project and at the same time provide considerable flexibility. This library can be imagined as an adhesive to integrate various components of a heterogeneous system easily.

There are some core features employed in the iCBD replication module, 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 that probably resides in a remote machine, also returning the resulting data

of the call. Which is very useful considering that the function that performs a call to the object does not need to know if it is dealing with a normal or a 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 its existence allows calls to that object that may originate in other programs.

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

Name Server Is this tool that keeps track of the objects actual locations in the network so they can move around freely and transparently. Works similarly to a yellow-pages book, providing lookups based on metadata tags.

Automatic reconnecting If a client (in our case a Replica Node) becomes disconnected to the server (Master Node), because of a server crash or a communications error, there is an automatic retry mechanism to handle this fault.

Secure communication Pyro 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 taken care of externally to 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 deserialises 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.

TCP Sockets and Secure Shell Protocol (SSH) Despite the facts presented above, system coordination is not all the burden laid on the network, the main task of this system is to replicate virtual machine images among the several nodes, so the network also has responsibility on carrying large volumes of data (result of transferring iMIs). The Pyro4 library gives the possibility to secure its communications, but that only covers method calls within replication nodes.

The delivery process of iMIs throughout nodes follow one 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 implies the transport of data between third-party networks, even the internet.

When talking about an organisational network, it's safe to assume that there are some security measures already in place (e.g. VLANs), so in this regard, we transfer the concerns about data security for that layer. That fact allows the use of a simple Stream

Socket [19] which provides a connection-oriented flow of data with error detection, in our case implemented on top of TCP. The application of this type of socket and the non-use of cyphers allows the best performance in the transfer of an iMI without the addition of computationally heavy tasks such as encrypting a large amount of data.

In the second case, to solve the question of the data travelling through open networks, an extension of the previous solution is presented, using the same type of socket but redirecting the flow through an SSH tunnel deployed between nodes.

This solution in addition to solving the issue of ensuring data security in the transferal process is a modular solution that allows future changes in the way data is encrypted without needing significant modifications to the code base. Even so, we do not believe that this is a perfect security model, there is room for improvement, but, not being the focal point of this work, we still wanted to provide some security features for conducting functional tests linking geographically separated data centres.

Data Compression Following one of the requirements stated above, our work should aim for reducing the bandwidth consumed by the operations of the iCBD platform, and that includes the replication of iMIs. Part of this subject is already addressed in the Storage Layer since the images, by default, are transparently stored in BTRFS compressed with zlib [28]. However, the replication process using the BTRFS send and receive features, as explained in section **??**, does not send the iMIs as is, send an instruction stream, and those instructions present as an excellent candidate to be compressed.

Given the design of the send receive feature, is unthinkable to hold in memory all the instructions waiting to be compressed, or store that information in a file, compress it an then send without creating a huge bottleneck and introducing a delay on the replication process. To expedite the process of transmitting the stream compressed and without postponements, only compression algorithms that provide a framing format (i.e. allowing compressing streams that can then more easily be decompressed without having to hold the entire stream in memory) were chosen.

In this work, we included three algorithms that presented the framing format characteristic, and that demonstrated to be popular in use, but maintained a code base modular where is very easy to add a new algorithm.

LZ4 is a lossless data compression algorithm centred on compression and decompression speeds, that belongs to the LZ77 family [58] of byte-oriented compression schemes. The reference implementation is in C and initially implemented by Yann Collet. There are also ports and bindings, from the original C implementation, in various languages like Java, C#, Python, Go, among others. In this work, we use a multi-threadable python biding by Iotic Labs called *py-lz4framed* [25].

zlib is a widely used, 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 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, can be found in the most diverse platforms: used by the Linux Kernel in multiple instances, multimedia libraries, databases, version control systems, and others. In the replication module, we employ the included zlib library [15] in python which provides an interface to the zlib C library.

Snappy is a compression decompression library, created by Google [17], that contrary to other algorithms, strives for very high speeds at reasonable compression rates, not maximum compression. The library is written in C++ but counts with several bindings for the most popular languages. In order to interface with python, we used the python binding [41] for the snappy C++ compression library, provided by Andrés Moreira.

4.2.4 Name Server

In a distributed systems environment the various nodes need to know how to communicate with each other: uniquely identifying themselves and refer to there locations. The mechanism that addresses this problem is commonly referred to as Naming. [51]

The iCBD Replication module implements this feature, attributing to each node an identity tuple with three elements: `(Node Name, Node URI, Tag)` that is registered in a name server allowing for subsequent locate a node by its name, or retrieve a set of nodes that are marked by the same tag.

The name server is a module that consists of a constant running thread and a local *SQLite* database. It makes use of 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 this feature with the remaining modules and starts up together with other modules of the Master Node.

Given the scenario where a node wants to make contact to another node and does not have its location. A request with the name of the node is made to the name server expecting in return a URI to call. If the requesting node already knows the location (IP and Port) of the name server, it makes the request directly. However, in the case where it also does not know how to contact the name server, resorts to a simple UDP network broadcast to locate the name server.

Methods in the Name Server API The Pyro Name Server presents an extensive API, but for the purposes of our work, only the subset presented below is utilised.

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

`register()` Enrol an object on 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.

```
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

`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.

This sub-module is present in every node (Master and Replicas) and presents itself with a central role in the replication process, not only, by the fact of acting as the backbone for the subscription of images, but also by tracking all versions of iMIs present in the platform working similarly to a versioning system. As described before, the iCBD

platform stores the multiple versions of one iMI as snapshots, that materialise as directories in the local filesystem. Like to what happens in the Name Server, the information stored by the Image Repository is backed in persistent storage in the form of an *SQLite* database.

The interface that the Image Repository offers is elementary, giving a hand full of mutator methods (get and set functions) to populate one main data structure. The structure used is the Python builtin Dictionary, allowing to establish 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 distinguish the multiple images present in the system unequivocally, but do not hold actual data.

In that sense, from this lightweight object, we can obtain: the name of the iMI, its version number, the full path to the VM files in the filesystem, the location of the boot package regarding the version in question, and the configuration file for the iSCSI target. Since the data stored within this object is appended on its 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; the iSCSI configuration files) the *icbdSnapshot* object only needs to maintain the paths to that data in relation to the local filesystem, leading to a clean and straightforward interface that can be seen in Listing 3.

4.2.6 Master Node

Following the architecture of the replication process, one of the fundamental roles is to manage the subscription of iMIs, disseminate new versions and provide an interface to interact with it all. Given those requirements, the Master Node resides in an iCBD Administration Machine allowing to hold a particular 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, recalls all iMIs in catalogue, maintains a list with all relations between iMIs and nodes - the subscriptions).

Moreover, adding to the above, this node also acts in a way as a gateway, delivering two methods of interfacing with the platform - a Command Line Interface (CLI) and an HTTP REST API.

More than being having a central role in managing and overseeing the several aspects of the replication platform architecture, this node holds the responsibility of sending the new versions of an iMI as they are created. This is the highlight feature that makes use of the BTRFS Incremental Backup functionality, in this case, the send operation.

```
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.

Sending a Snapshot to a Replica Node (Cache Server) The general idea is: when the administration process of an iMI comes to an end, the Master Node is notified that a new version of that iMI is available. So, a new entry for that version is created and stored in the local Image Repository. Then, the main replication procedure starts.

First, it is checked which nodes have subscribed to iMI. For those who have subscribed we need to check the latest version available in the Replica's Image Repository, this process will determine if the new version can be sent immediately, or if it is necessary to transfer some intermediate versions.

Any of the cases, we are transmitting only the differences between versions, not the whole iMI. Assuming that only the last delta (the most recent changes) will be shipped, it is determined whether the transfer will occur using an encrypted communication channel and/or whether some compression algorithm will be applied in 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 all this process begins the data transfer, being aided by the libraries mentioned above, btrfsLib; compressionLib and sshLib. The process of receiving an IMI can be found in the Section 4.2.7.

Keep Alive There is a point mentioned when presenting the Pyro4 platform mentioned that this library allows the automatic reconnection of clients to the Master Node, however, the opposite is not provided. In case of failure of a Replica Node, the Name Server is not automatically informed of this failure, in fact, this event would only be noticed when a connection with that node failed.

Since we want the Master Node view always to be consistent with the status of the platform, it is essential that the Name Server always hold its information up-to-date. It is to satisfy this accountability that this submodule enters in the picture. Similarly to the Name Server, the Keep Alive feature is also launched in conjunction with the Master Node and also runs on its own thread, making the execution flow independent of any other sub-module.

The principal task implies verifying the activity status of all Replica Nodes, indexed by the Name Server, in a quite simple manner. Taking the list of all Replica Nodes and with a periodicity of about ten seconds, each node is reached with a style of a ping expecting a response to the challenge. If there is no acknowledgement, two further contact attempts are performed, if those too fail, it is assumed that something wrong happened to that node and is therefore declared inactive. At this point, the Name Server is reached notifying that a node entered a failed state and should be excluded from the record.

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, which is nothing more than a program that launched from a terminal textually recognises a series of commands and returns its result.

This interface, although simplistic, allows to efficiently manage all the tasks related to the replication of iMIs between the several servers of the platform. 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

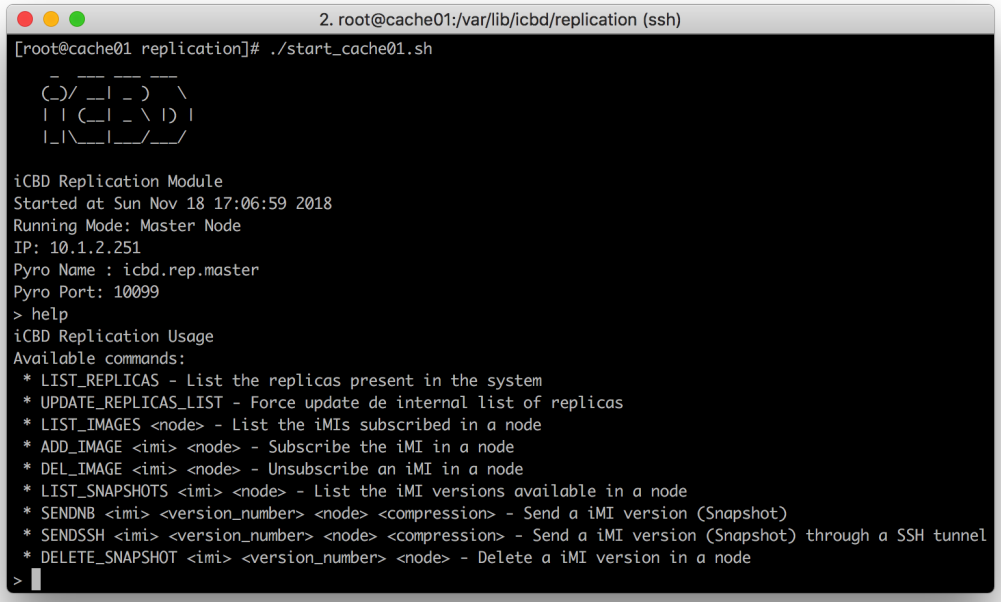

Figure 4.3: iCBD Replication Module help output

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.

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:

`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

4.2.7 Replica Node

Receiving a Snapshot TOPICS :

- Operations on the local repository
- The receive operation

4.3 Deploying an iCBD Platform with a Cache Server

4.3.1 The infrastructure

Network

4.3.2 Services

Roles in the Platform

iCBD-imgs

iCBD-rw

iCBD-home

iCBD-cache

Installing iCBD Core Services

BTRFS bug found in CentOS 7 Kernel As a curiosity during the process of building the *iCBDimgs* VM, we found in 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` fails 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 performance of the implemented replication module and the executability and the performance improvements to the introduction of a cache server role in the iCBD platform.

The chapter is divided into the following sections:

Section ?? .

Section ?? ..

5.1 Motivation

TOPICS :

- Benchmark the iCBD-Replication Module
- Assert the performance gained by storing iMI closer to client workstations

5.2 Metodology

5.3 Experimental Setup

Unless stated otherwise, all tests were executed ten times removing the best and worst result. The final result is the average of the remaining values.

5.4 Replication Testing

TOPICS :

	FCT NOVA	Reditus
Servers	2x HP ProLiant DL380 Gen9	2 x HP 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 5.1: Physical infrastructure of the FCT NOVA and SolidNetworks sites

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 5.2: 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 5.3: Specifications of the HPE MSA 2040 SAN Storage

- Replication with rsync (100 / 1000 mbps)
- Replication with iCBD-Replication - Plain Sockets and No Compression (100 / 1000 mbps)
- Replication with iCBD-Replication - Plain Sockets and LZ4 Compression (100 / 1000 mbps)
- Replication with iCBD-Replication - Plain Sockets and zlib Compression (100 / 1000 mbps)
- Replication with iCBD-Replication - Plain Sockets and snappy Compression (100 / 1000 mbps)
- Replication with iCBD-Replication - SSH and No Compression (100 / 1000 mbps)

5.5 Plataform Testing

TOPICS :

- benchmarking
- Boot time Lab PC
- Boot time iCBD VM in Cluster

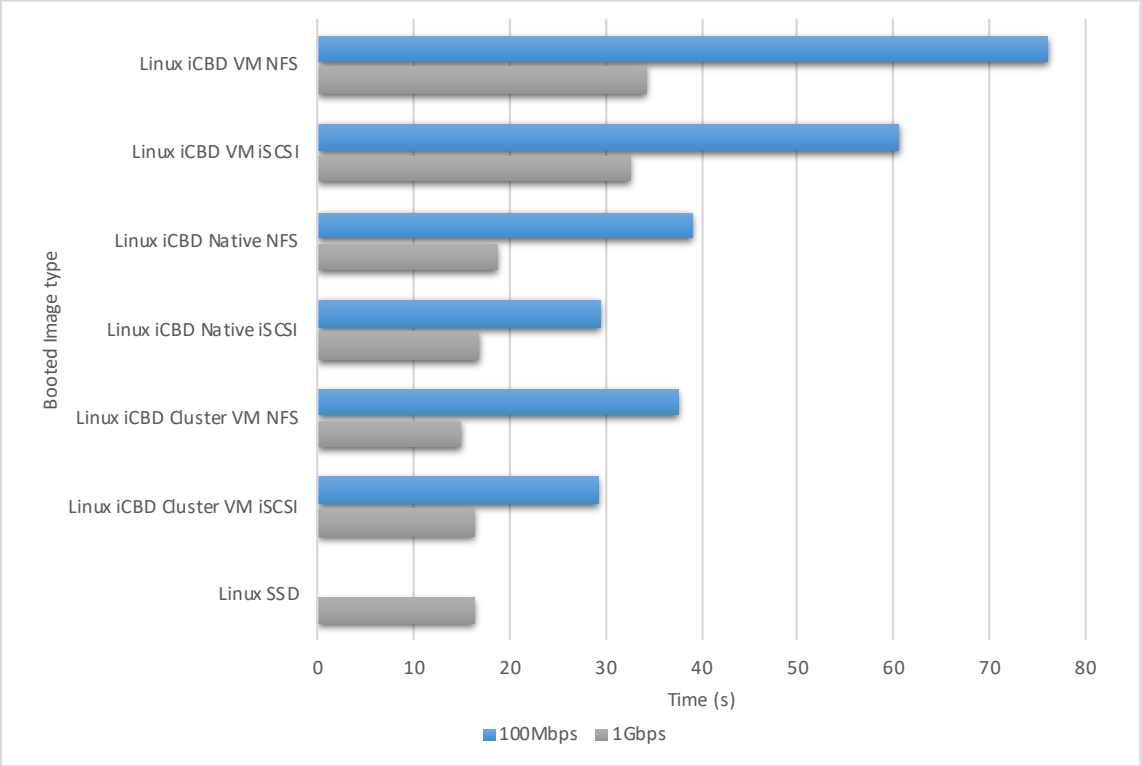

Figure 5.1: Boot time from iCBD-imgs VM

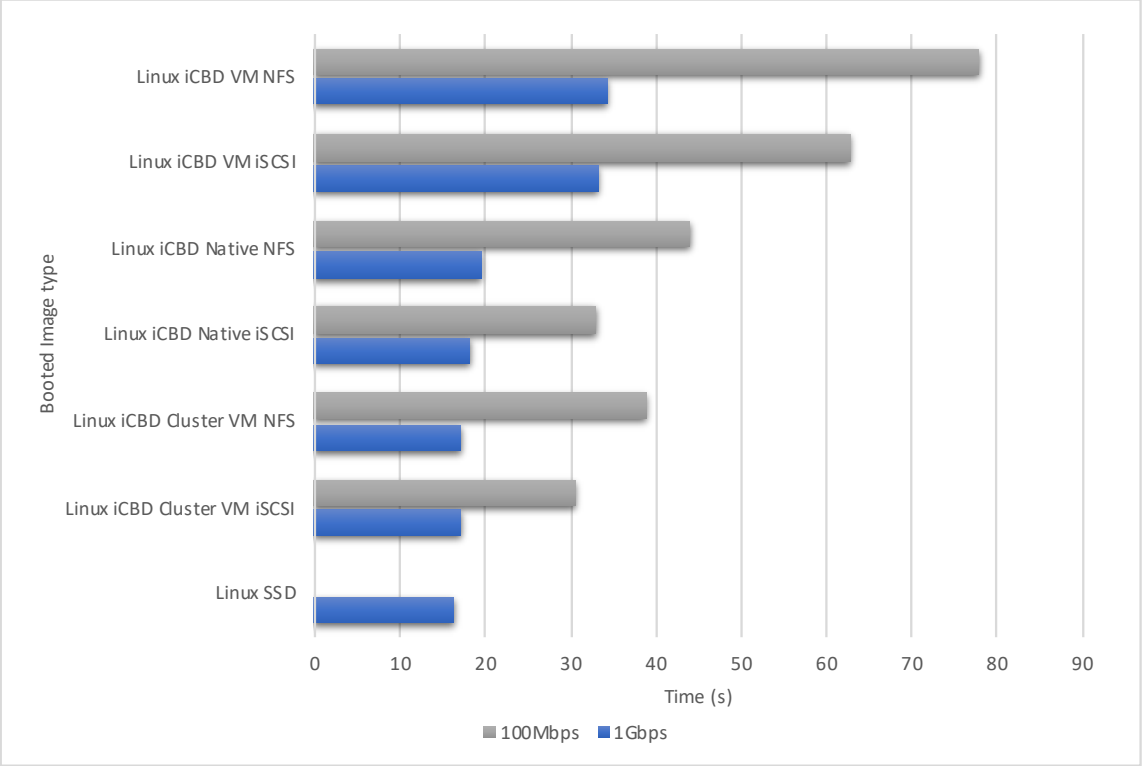

Figure 5.2: Boot time from iCBD-Cache02

- Boot time iCBD in Lab PC (100 / 1000 mbps) iCBD-Imgs VM
- Boot time iCBD in Lab PC (100 / 1000 mbps) iCBD-Cache

Chapter 6

Conclusions & Future Work

6.1 Conclusions

TOPICS :

- The requirements of the implementation of the iCBD-Replication were achieved
- Simple replication of iMI though multiple nodes with a subscription model, where the node only receives the iMIs that really want.
- Successful creation of a complete iCBD platform in FCT NOVA campus.
- Two department lab running the solution as a trial, and instrumental in getting experimental results

6.2 Future Work

TOPICS :

- Replica to Replica iMI transfer
- Diskless Servers / selfhosting - provision iCBD Servers from iMI
- Micro Services, as started with this thesis (building iCBD-Replication as a standalone application) the functionalities of the iCBD Core can be segmented in multiple small services, in order to achieve a better use of resources and an easier deployment in a multi homed scenario.
- Infrastructure as Code, orchestrate and automate all the process described in the chapter Implementation of a Cache Server.

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] 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.

[11] 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.

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

[13] *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).

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

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

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

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

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

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

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

[21] 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.

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

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

[24] 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.

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

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

[27] 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.

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

[29] 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).

[30] 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).

[31] 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).

[32] A. Kivity, U. Lublin, A. Liguori, Y. Kamay, and D. Laor. "kvm: the Linux virtual machine monitor." In: *Proceedings of the Linux Symposium* 1 (2007), pp. 225–230. URL: https://www.kernel.org/doc/mirror/ols2007v1.pdf{\#}page=225.

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

[34] 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.

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

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

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

[38] 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).

[39] *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).

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

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

[42] 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.

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

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

[45] 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.

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

[47] 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.

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

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

[50] 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.

[51] 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.

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

[53] 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.

[54] *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).

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

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

[57] 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.

[58] 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