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: Nuno Preguiça, Associate Professor,
NOVA University of Lisbon

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

Examination Committee

Chairperson: Name of the committee chairperson
Raporteurs: Name of a raporteur
Name of another raporteur
Members: Another member of the committee
Yet another member of the committee

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

March, 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

Lorem ipsum.

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.

Prof. Nuno Preguiça, Prof. Pedro Medeiros, Prof. Paulo Lopes

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 handled. 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 allowing for fast deployment of Virtual Machines (VMs) in a high consolidation ratio environment and with minimal need for management.

New approaches to virtualisation techniques are being developed at a never seen 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 already present computation power 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 conventional file system 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 system in a high throughput network, maintaining high availability and scalability properties while supporting a broad set of clients efficiently.

Resumo

Nos últimos anos, tem-se assistido a mudanças fundamentais na forma como a capacidade computacional é gerida. Com o grande aumento da utilização da virtualização a forma como são geridas as máquinas físicas e os modelos de infraestruturas de um centro de dados sofreram grandes alterações. Esta mudança é o resultado de uma procura por uma forma de disponibilizar rapidamente uma VM num ambiente altamente consolidado e com a mínima necessidade 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 a entrega de grandes empresas da industria das tecnologias de informação a problemas como a virtualização de desktops com algum sucesso, mas ignorando completamente o poder de computação que já está presente nos seus clientes. Optando ao em vez, por uma via de alto custo, adquirindo máquinas poderosas e vários softwares. Existe ainda espaço para melhores soluções e para o desenvolvimento de tecnologias que façam uso das capacidades de computação já se encontrem presentes com o mínimo de esforço na sua configuração.

Esta tese foca-se no desenvolvimento de mecanismos de replicação e caching para imagens de maquinas virtuais armazenadas num sistema de ficheiros convencional com a funcionalidade de fazer snapshots. Existem alguns pontos em particular a endereçar: a solução tem que seguir uma arquitectura distribuída e ser totalmente integrada numa solução client-based VDI; tem que funcionar com enormes ficheiros apenas de leitura alguns deles resultantes da criação de snapshots mantendo a característica de manutenção de versões. Este trabalho também incide na exploração dos benefícios de utilizar tal sistema numa rede com uma alta taxa de transferência de dados, em quanto mantem propriedades de alta disponibilidade e escalabilidade suportando um largo conjunto de clientes de forma eficiente.

Contents

List of Figures

List of Tables

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.
OS	Operating 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 virtualization, despite all the recent discussion, isn't new. In fact, this technology has been around since the 1960s [9], but not until the development of virtualization technologies for the x86 architecture [1] and the introduction of *Intel VT* [34] and *AMD SVM* [17] in the 2000s 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 machines, 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) [26] 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 major 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

Virtualization is the pillar technology that allowed for the widespread of the IaaS cloud providers in a economy of scale model. These cloud providers, such as Amazon AWS [6], Microsoft Azure [27] and Google Cloud Platform [15], 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 efficiently all this data? 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 own storage solutions. For instance, Google uses BigTable distributed storage system [10], 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 to 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 [35]. At the start, this file system already adopts an efficient system of snapshots and it has as a primary design principle to maintain an excellent performance in a wide 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 larger project with the name Infrastructure for Client-Based Desktop (iCBD) [24], under development at Reditus S.A.

in collaboration with 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 virtualized 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 solutions that already exist. Such the adoption of a diskless paradigm with a remote boot, the way virtual machine images are stored in the platform and the support for a virtualized or native execution on any workstation, depending on the user's choice. [23]

The Remote boot support is offered by HTTP, TFTP, and DHCP servers, and in turn, the image repository servers manage the storage of the VMs templates and the production of 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. [2, 23, 25]

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 centralized 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.

1.3.2 Previous Work

There have previously been two dissertations involved in this project. That work has centred in the creation of the instances of virtual machines, more specifically in the creation supported by native snapshot mechanisms of the file system where the templates are stored. This way instead of using the hypervisor itself as a method to provision full or thin clones the work is done by the file system snapshot system.

As is happening now, the two theses have followed two different paths in an attempt to determine which file system best suits these objectives. Being that one used a local file

system, the BTRFS, and the other followed the object-based storage path, adopting the CephFS.

1.4 Project Contributions

This work, as a part of a bigger project and building on previous contributions, has as premise a couple of existing technologies in the file systems field to create a replicated and distributed environment capable of storing large files consisting mainly of *VMs* templates and golden images. This work not only focuses on storage management aspects, as also attends the need of being integrated into a larger infrastructure and coexist with a wide variety of other systems.

1.4.1 Main Expected Contributions

The main expected contributions are:

- The study, develop, and evaluate an implementation of a distributed and replicated BTRFS file system for VM storage.
- Implement a server-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.

A detailed view of the planning can be found in **Chapter ??**.

1.5 Document Structure

The remnant document is structured as follows:

- *Chapter* **?? Related Work** - 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 virtualization techniques.
- *Chapter 3* **Proposed Work** - In this chapter, there is a presentation of the work plan for the elaboration of this dissertation. Giving also an overview of the solution to develop on the duration of this thesis.

Chapter 2

Research Context

The focal point of this dissertation is the challenges of implementing a distributed system based on a file system, that can store *VMs* images while leveraging the benefits of snapshots and caching techniques. Moreover, the work done should integrate smoothly into a broader infrastructure illustrated in detail in Section 3. In this chapter, we bestow 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 ...

Section 2.2 studies the principal characteristics of a file system, with emphasis on snapshot techniques ...

Section 2.3 talks about ...

Section 2.4 expands on ...

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 [31], in that order, it was the capacity to virtualise all of the 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. [1]

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 [31]:

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 the blend of an Operating System and a Virtual Machine Monitor. It can make use of a run-of-the-mill OS, such one of the several flavours of *Linux* and the *Microsoft Windows*, or a bare metal purpose-built one, such the *VMware ESX/ESXi* family.

Concerning the execution of a VM, the hypervisor kernel spins up a VMM, which holds the responsibility of virtualising the *x86* architecture and provide the platform where the VM will lie. This way, since the VM executes on top of the VMM, there is a layer of separation between the guest VM and the host hypervisor kernel, with the communications within being made through the VMM. This feature confers a necessary degree of isolation among the system. With the host kernel taking care of host-centric tasks as *CPU* and memory scheduling, and the network and storage *I/O* stacks, and the VMM assumes responsibility to provide those resources to the VM.

A hypervisor can be classified into two different types [7], symbolising two different design strategies to virtualisation, as shown in Figure 2.1:

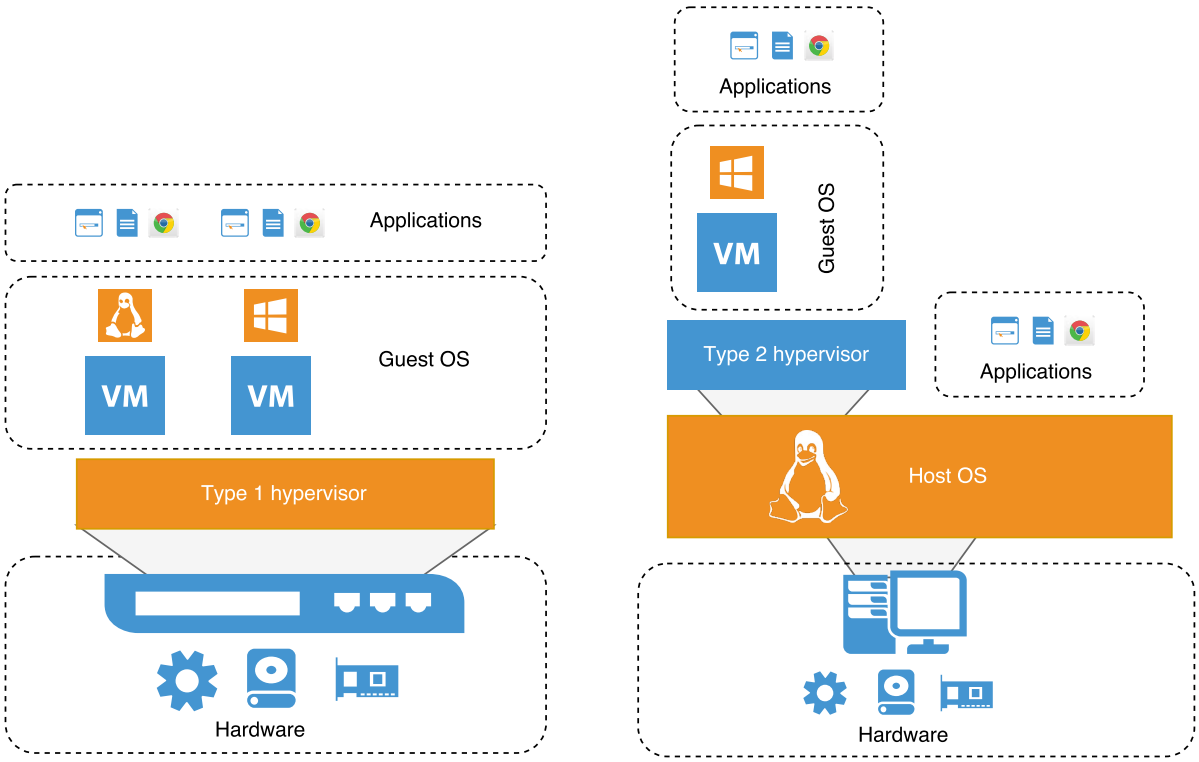

Figure 2.1: Virtualization architecture with type 1 and type 2 hypervisors

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

In addition to the improved performance provided by the partitioning of devices between the several guest VMs, this kind of 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 is 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. As do drivers for devices that have the capability of being shared amongst guests.

Furthermore, considering that Type 1 hypervisors do not have an underlying OS, the complexity of installing and configuring this type of solution increases.

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 are integrated seamlessly with the remainder of the system. An excellent illustration of this kind of paradigm is Kernel-based Virtual Machine (KVM) [22] and VMware Workstation/Fusion [1].

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. One advantage is quickness of installation and configuration since the installation process for most of these type of hypervisors only requires the execution of an elemental installer, delivering the instantaneous ability to run a multitude of different operating systems.

All the above culminates in some advantages. Type 2 hypervisors are regularly deployed for the use of software testing, dismissing the requirement for a dedicated test machine. Also, provide support for running multiple OSs on the same physical machine, which can be valuable for those who rely on multiple applications written for a particular or legacy OS.

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 is the hypervisor that is tasked to do all the management and load balancing. The settings of these components reside in a VM configuration file. In the case of VMware hypervisors, the file employs the `.vmx` extension.[32, 42] In a KVM environment, the configuration is in a `.xml` file. [11]

With a virtualised infrastructure there is an opening 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 resource to cloning techniques. Software updates can be hugely simplified and made available to all users at once. Even availability is improved since it is an easy task to launch a new VM from a template and migrate all the services that were being made reachable by one that suffered a failure.

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.

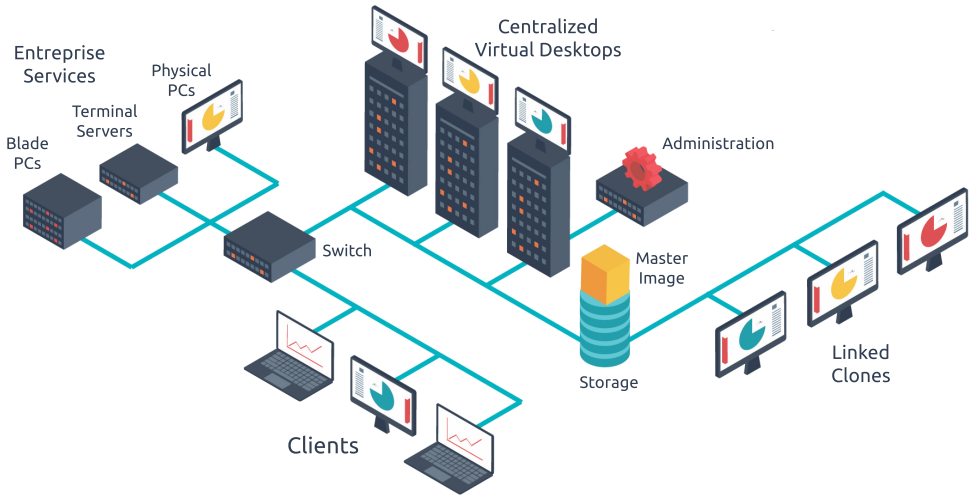

Figure 2.2: An exemple of a Virtual Desktop Infrastructure, adapted from AppDS [8]

Organizations 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 off work hours to produce the shortest disturbances.
- Periodical software updates (such Microsoft's famous Patch Tuesdays [30]) are ordinarily unleashed in the morning 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 above is to minimise the footprint of installed software having in account the locality of the data. It is possible to imagine a dimension where all the software required to execute a workstation (Operating System and applications) can be conjugated in a virtual machine. This mechanism allows for the

virtualisation of a workstation that can be performed 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 VDI.

The concept encompasses a series of techniques, providing on-demand availability of desktops, in which, all computing is performed employing virtual machines [18]. 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) [33] or the Remote Framebuffer Protocol (RFB) [39] 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 [40], XenDesktop from Citrix [44] and Microsoft with Microsoft Remote Desktop [29].

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. [12]. An example is a previously existing solution by Citrix, the XenClient [44]

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. [45] 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

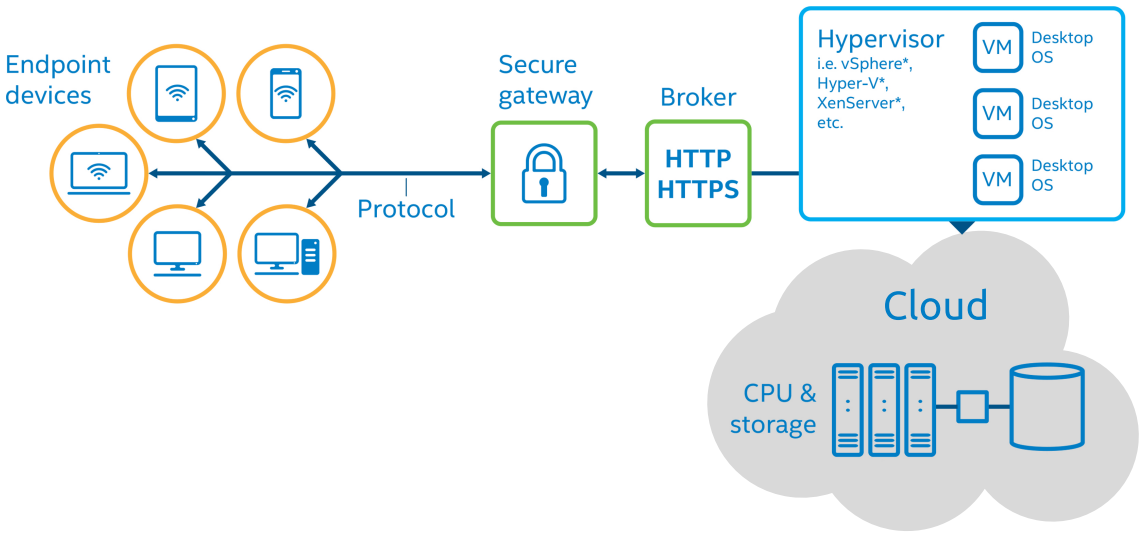

Figure 2.3: Conceptual overview of DaaS architecture, adapted from Intel [21]

leading the way. Amazon in its AWS portfolio delivers Amazon Workspaces [5], and Microsoft implements the RDS features [28] on the Azure product line. Nevertheless, there are also some smaller contenders, as an example, Workspot [43] (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 [42], 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 [20]

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 a specific care 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

2.2.1.1 I/O Storms

Boot Storm bla..

Login Storm bla..

Malware and Anti-Virus Software Scanning bla..

Big Applications Needs bla..

Operating System Updates bla .

2.2.1.2 Cost by workstation

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. [13] We can give as example of file systems the well-known Network File System (NFS) [36] originally developed by Sun Microsystems, and the notable Andrew File System (AFS) [37] developed at Carnegie Mellon University.

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. It has already been mentioned that previous work has been done to use the file system snapshot features as a base feature. This way the creation of linked-clones handled by the file system capabilities as an alternative to linked-clones created by virtualization software itself.

TODO - Talk about btrfs seed The importance of btrfs seeding relays on the fact that this feature alows for the multiple mounting operation of the same file system in read only mode. Thus allowing multiple clients to use the same image.. (Fulcral to the the platform) http://www.oracle.com/technetwork/articles/servers-storage-admin/advanced-btrfs-1734952.html

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 [41], OpenStack Swift [38], and in a IaaS flavour the Amazon S3 [3] and Google Cloud Storage [14].

2.2.3 Snapshots

TO DO - Expand

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.

x

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.

Where the primary goal is to achieve a particular kind of VDI infrastructure, a client based VDI, where client's computations are performed directly on the client hardware as opposed to 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 principal architectural components of the platform, with emphasis on the different layers and how they act together to serve the end-user.

Section 3.3 will at last state the problem and motivation for introduction replication techniques in the storage components of the platform. Moreover, the section prefaces the importance of the implementation of cache servers that hold part of the

distribution burden and crucial for the support of an increased number of clients.

3.1 The Concept

The iCBD as a project pretends to investigate and develop an architecture that leads to the birth of a platform that can operate desktop virtualisation (VDI). In a sense, the goal is similar to a client-based VDI, but with the distinction of 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 each physical workstation from virtual machine templates (VMs) kept in repositories. We will talk more about this subject in section 3.2

To summarise the platform should be able to:

- Tuning 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 from a standard local installation of an Operating System (OS) to the use of this platform.
- Simplify installation, maintenance and platform management tasks for the entire infrastructure, including servers in their multiple roles, storage and network devices from a single point.
- Allow for a highly competitive per workstation cost.
- Maintain an inter-site solution; such a geographically disperse multi-office structure.

3.2 The Architecture

The iCBD platform comprehends the use of multiple services that take responsibility for an essential set of tasks. To achieve a better understatement of the inner workings of the system we can group these services in four major architectural labels as seen in the figure 3.1.

iCBD Machine Image (iMI) a nomenclature borrowed and adapted from Amazon Web Services AMI [4] embodies the required files to run a iCBD platform client. Including a VM template (with an operating system, configurations and applications) the

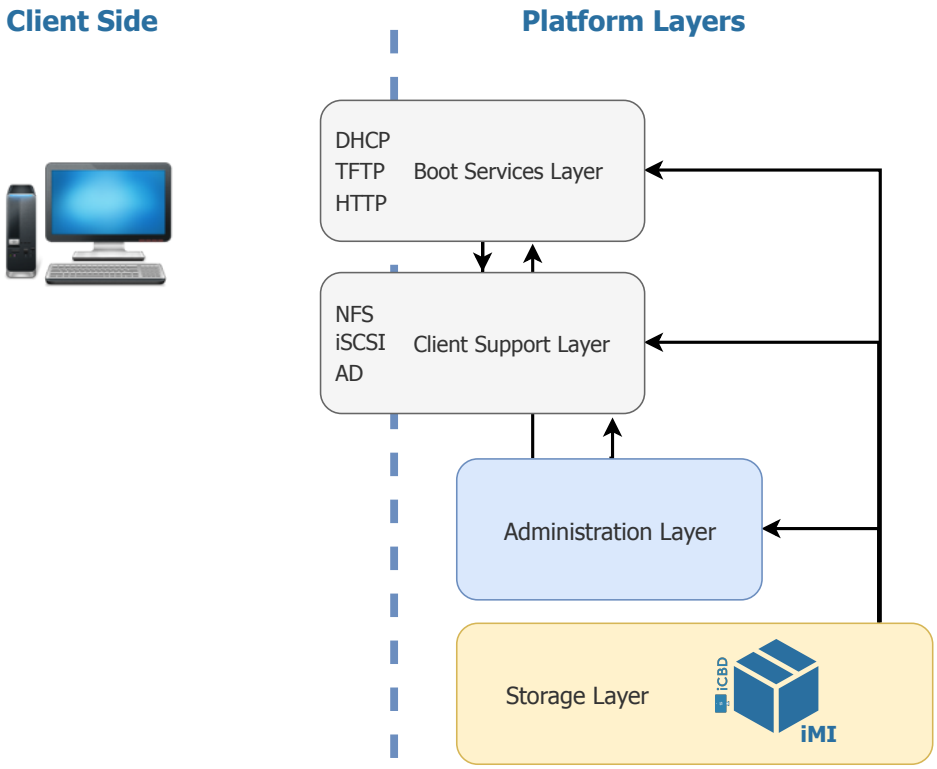

Figure 3.1: iCBD Layers View

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.

Boot Services Layer is responsible for providing the initial process from which the client machines will boot from the network and the posterior transference of a bespoke boot package. Employing services such as PXE, DHCP, TFTP and HTTP.

Administration Layer (in regards to software) takes advantage of a virtualisation stack (can be based in either KVM or VMWare products) to engage in maintaining all the needed aspects for the successful creation and update processes of an iMI lifecycle. Employing a custom set of scripts, the creation of an iCBD Boot Package is also a duty of this layer.

Client Support Layer deals with the demands of a deployed and running iCBD image, such as, providing read/write space (since iMIs run on diskless workstations) and storing users home directories. As well as, hosting domain controllers, centralised authentication amongst other services that can be already in place in the midst of a clients infrastructure. Granting the ability to deploy a customised iMI in any scenario.

Storage Layer is accountable for maintaining the repository of iMIs and facilitate essential operations like version controlling the VM images files. Is also in this layer that

we seize the potential of replication features provided by the file systems employed. In this project, the storage relies on two mainstream file systems: BTRFS and CEPH. Together with services like NFS and iSCSI enables a way to export data to clients.

Let's view in more detail each one of them.

3.2.1 iCBD Machine Image

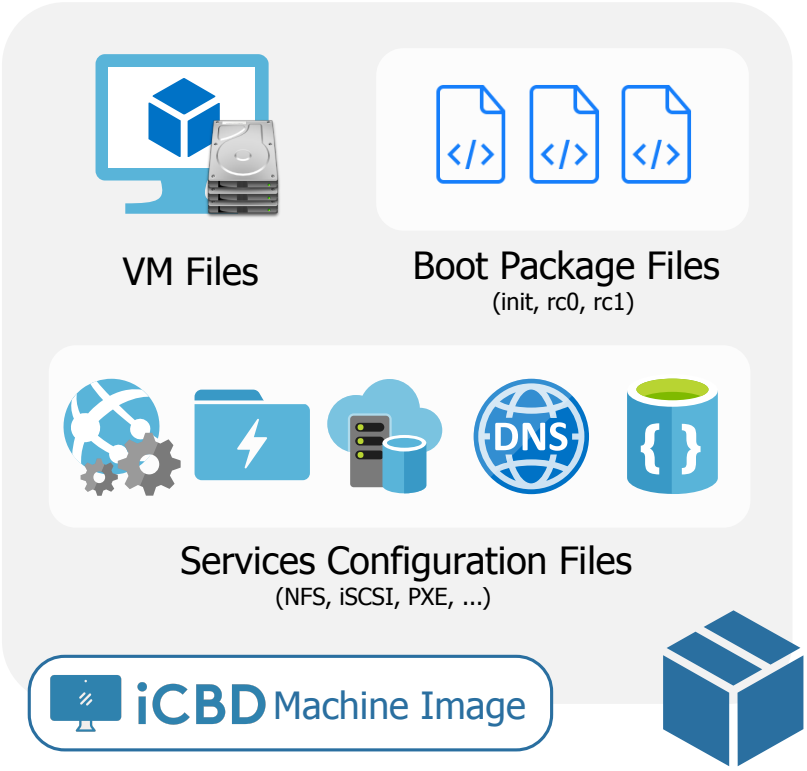

Figure 3.2: iCBD Machine Image Files

In its essence, an iCBD Machine Image is the compilation of everything that is needed to run an Operating System within the iCBD platform, that is data and configuration files. For the sake of simplicity, we may 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 from VMware and KVM VMs 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, as the one used, there is a need to keep a set of files that manage the boot of a workstation; these 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, mounts all file systems and ultimately bring the system

up in 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 configurations.

Services Configuration files Among the services employed there is the need for changes in some of the configurations files of these services. The NFS exports configuration file should reflect a structure of which file systems are exported, the networks that a remote host can use, as well as, a myriad of options that the NFS allows to be set. The same happens to iSCSI where an iSCSI targets need to refer to a backing store of the storage resource where the image resides.

3.2.1.1 iMI Life Cycle

The life cycle of an iMI encompasses all the stages that an iMI takes throughout its course within the platform, as seen in Figure 3.3. From creation, passing through live deployment in use by multiple clients and finally its decommission and temporary or cold storage.

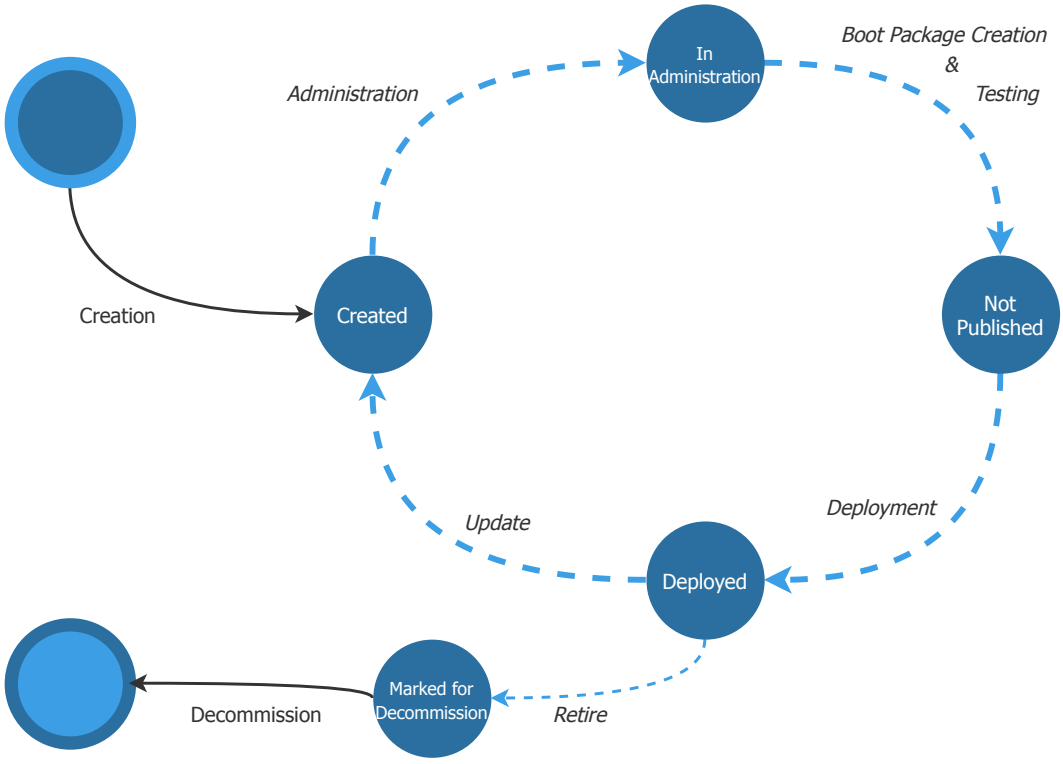

Figure 3.3: iMI Life Cycle inside the iCBD Platform

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 computationally light operation.

During its stay on the platform, an iMI can present four main states:

Created After the being inserted in the platform, an image is not instantaneously camera ready (i.e. able to be served to clients and booted in a workstation), it needs to pass through the administration layer for the generation of a boot package.

In Administration An iMI goes through this phase in two moments. The first is the case described above, where an image has just been injected into the platform, and it is necessary to create the conditions for it to work in this system. Continuing to the most frequent case, which happens when an image needs an update or any set of changes. It is here in this state that in interaction with the administration layer provides with a way to administer and update an iMI. The iMI will stay in this state as long as being managed (which can take from a few minutes to hours) and the duration of the process for creating the boot package.

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.

Deployed 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.

3.2.2.1 Booting an iMI in a Workstation

After the selection of an iMI in the PXE boot menu [16] 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.

3.2.3.1 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 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.

3.2.3.2 Creating the boot package

Points: The adm script The esxi use to avoid nested virtualisation The mki - crating the boot package

3.2.4 Client Support Layer

NFS and iSCSI to provide the iMI Same services to provide r/w space (also btrfs) and user homes Possibility to integrate Samba shares, LDAP, Active directory, more services embedded in the platform.

3.2.5 Storage Layer

For each iMI, there is a snapshot with an index number that relates to its version and age (the higher the number the most recent the version is). Multiple versions of an iMI are stored in directories named by the index of the version. So, is simple to create a new linked clone from the most recent version

Why btrfs? The way the files are stored. The use of BTRFS multiple volumes for different parts of the platform. The use of cloning to save multiple versions of an iMI, giving the possibility to roll back unwanted changes. The need to replicate data - multiple locations and cache server (one of the focus of the thesis) Should be transparent the the remaining layers. As being develop in Joao's thesis the use of CEPH it should be used with little to none modifications to other layers.

3.3 Replication and Caching - The Problem

3.3.1 Motivation and Goals

3.3.1.1 Replication

3.3.1.2 Cache Servers

To solve some of the enunciated problems with a DaaS solution that derive from limited bandwidth, latency and jitter from the limitation of accessing the image repositories from an internet connection and provide some scalability feature with the implementation of proximity cache servers are key. These cache servers can store replicas of the iMIs created and maintained in an administration server. Moreover, since they are located in the same LAN segment as the clients is from here that they will boot. For accomplishing this work, the cache servers need to have hard drives. (Although it would be possible to have diskless cache servers, they would be blocked if there was an interruption in the internet access, and it is to avoid that the local drives are necessary.)

3.3.2 System Overview

3.3.3 Requirements

Chapter 4

Implementation of *iCBD-Replication* and Cache Server

This chapter addresses the implementation of the central topics of this thesis, divided into two major fields. The first section talks about the creation of a middleware system that provides replication features in an integrated way to the iCBD platform. A detailed description of the concept and architectural model, as well as the implementation decisions, can be found in this section. Then, we show how the performance of the platform clients can be heightened by setting up a client-side caching system that stores images adjacent to the consumers. 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.

4.1 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 need 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, bosting 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 VM images stored.

4.1.1 Requirements of the Module

Since the beginning of this work the file system to be used as storage was set. Not because is belived to be the best for this type of work, but because is belived that is one of the best. And since the grandure of the project there is an analogous thesis doing work with other type of file systems, a distributed object storage oriented one named Ceph [41].

The are some good reasons for using the BTRFS File System some that only will show up in the decour of this document, but we can enunciate two that are fundamental. The foremost is the support for snapshots

Requirements

The file system is set BTRFS will be used. Many reasons for that: The most important is the support for snapshots Compression

4.1.1.1 Preliminary tests on the BTRFS Incremental Backup features

The first step is to try to understand the most efficient way to transfer this peculiar kind of data. 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. In this sense, we next present some preliminary tests in multiple ways of transferring snapshots between BTRFS file systems both in the same machine and in different ones. The results obtained here conjugated with the defined requirements are essential for defining the architecture of the replication module steering the implementation at its best path.

Before the start of any implementation, there was the need to validate the capabilities of the BTRFS file system regarding sending snapshots across different systems. As one of the requirements was the efficiency of the transference of data. So a small comparison was in order.

4.1.2 Image Repository

4.1.2.1 iCBD Snapshot Structure

In the replication module we treat a snapshot not only as raw data, but a collection of data and metadata that is essencial to unequivocally distinguish the multiple images present in the system.

4.1.3 Communications between image repositories

4.1.3.1 Pyro4 Library

4.1.4 Master Node

4.1.4.1 CLI Interface

4.1.4.2 REST API

4.1.5 Replica Node

4.2 Building a iCBD Cache Server

Found a Centos 7 kernel bug.

4.2.1 The infrastructure

Machines List: TODO Servers - 2x HP ProLiant DL380 Gen9 Switch - HPE flexfabric 5700 jg898a Disk array - HPE MSA 2040 SAN Storage

4.2.2 Services

4.2.3 Networking

4.2.4 Extra Efforts

GitLab

Since the work maily goes around replication and infrastructure problems, makes all sence to think in how the base code is handled. Thinking on this subject and avaluating the code backup system in place (talk what is the system in place), a svc system is ideial to what we what to acomplish. So a GitLab on premises system was deployed and configured. Also configured multiple repositorys that will back each module of the iCBD platform. There are two main objectives with this premisse. First, provide an safe envoiroment for backing up all the base code of the modules. As well as provide versioning control of the that same code. Second, facilitate a way to replicate the base code of the multiple modules though the various infrastructures running the icbd platform in a clean and transparent way. Talk a bit of git vantages. (Replication possibilities, backing with the cloud..) How is implemented. (Vm in reditus infra..)

Chapter 5

Evaluation

5.1 Motivation

https://stackoverflow.com/questions/1198691/testing-io-performance-in-linux https://dl.acm.org/citation
https://github.com/axboe/fio https://github.com/giantswarm/filesystem-benchmark

5.2 Functional Testing

https://stackoverflow.com/questions/5357601/whats-the-difference-between-unit-tests-and-integration-tests

5.2.1 unittest — Unit testing framework

https://docs.python.org/2/library/unittest.html
Memory profile of the module

5.3 Integration Testing

Chapter 6

Conclusions & Future Work

6.1 Conclusions

6.2 Future Work

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. (Cloud Native Infrastructure Pag.66)

Bibliography

[1] 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.

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

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

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

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

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

[7] 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.

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

[9] 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.

[10] 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.

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

[12] *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).

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

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

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

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

[17] 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.

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

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

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

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

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

[28] 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).

[29] *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).

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

[31] 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.

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

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

[34] 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.

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

[36] 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.

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

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

[39] 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.

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

[41] 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.

[42] *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).

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

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

[45] 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.

ANNEX I

Annex 1 iCBD-Replication Documentation

iCBD-Replication Documentation

Release 1.0.0

Luis Silva

Jan 16, 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**(*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*, *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.Serializer module

class lib.Serializer.**Serializer**
Bases: object

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

static icbdSnapshot_dict_to_class(*class_name*, *dict*)

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.**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*)
Bases: object

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.sshlib module

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

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

1.12 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

1.13 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

1.14 exceptions.ReplicasException module

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

Raise when a replica is not found

1.15 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**()

1.16 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()

1.17 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

i

l

t

INDEX

H

I

K

L

M

N

P

R

S

T

Z

ANNEX II

ANNEX 2 iCBD INSTALLATION GUIDE

iCBD Installation Protocol

Version 1.0.1 - Last Updated 30 Jan 2018

Luis Silva - `lmt.silva (at) campus.fct.unl.pt` | `luis.tsilva (at) reditus.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 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
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
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.

References

CentOS 7 Documentation - Enable or Disable SELinux

HowToForge - A Beginners Guide To btrfs