Designing a flexible and modular architecture for a private cloud: A case study

Conference Paper · June 2012

DOI: 10.1145/2287056.2287067

CITATIONS
6

READS
53

2 authors:

Valeria Cardellini
University of Rome Tor Vergata
108 PUBLICATIONS **2,711** CITATIONS

SEE PROFILE

Stefano Iannucci
Mississippi State University
21 PUBLICATIONS **238** CITATIONS

SEE PROFILE

Designing a Flexible and Modular Architecture for a Private Cloud: a Case Study

Valeria Cardellini
University of Roma "Tor Vergata"
cardellini@ing.uniroma2.it

Stefano Iannucci
University of Roma "Tor Vergata"
iannucci@ing.uniroma2.it

ABSTRACT

Cloud computing is an emerging paradigm used by an increasingly number of enterprises to support their business and promises to make the utility computing model fully realized by exploiting virtualization technologies. Free software is now mature not only to offer well-known server-side applications, but also to land on desktop computers. However, administering in a decentralized way a large amount of desktop computers represents a demanding issue: system updates, backups, access policies, etc. are hard tasks to be managed separately on each computer. This paper presents a general purpose architecture for building a reliable, scalable, flexible, and modular private cloud that exploits virtualization technologies at different levels. The architecture can be used to offer a variety of services that span from web applications and web services to soft real-time applications.

To show the features of the proposed architecture, we also present the design and implementation over it of a Linux Terminal Server Project (LTSP) cluster that benefits from the underlying IaaS services offered by the private cloud. The cloud infrastructure, as well as the LTSP, have been implemented exclusively using free software and are now in a production state, being used by approximately 200 users for their everyday work. We hope that our description and design decisions can provide some guidance about designing an architecture for a cloud service provider.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Distributed Systems*

Keywords

Private cloud, storage, virtualization

1. INTRODUCTION

Cloud computing has recently emerged as a paradigm for delivering computational services over the Internet. Small and medium organizations are attracted by this computing paradigm because it let them not to own an in-house datacenter with the associated risks and costs but rather to just rent what it is effectively needed time after time.

There are different types of clouds, each with its own benefits and drawbacks. With *public clouds*, providers offer their resources as services to the general public. Key benefits of using public clouds include no initial capital investment on the infrastructure and risk shifting to cloud providers. *Private clouds* are designed for exclusive use by a single organization. They may be built by the organization itself or by an external service provider. A private cloud offers the highest degree of control over performance, reliability, and security. However, it is often criticized for being similar to traditional proprietary server farms and does not provide benefits such as no up-front capital costs. Finally, *hybrid clouds* are in between public and private: they are primarily based on private clouds, but they can extend their capacity with public clouds should the need arise.

Cloud providers offer services at three different layers, respectively named: *Infrastructure as a Service (IaaS)*, *Platform as a Service (PaaS)*, *Software as a Service (SaaS)* [11]. These three levels offer a layered view of the cloud computing stack. The user of the lowest layer rents from a IaaS provider virtualized resources like computational power, storage, and network and has the task to manage the rented resources. The middle layer is the PaaS, in which the cloud user acquires the control over a software platform, ideally an application server and an application development environment, where he has to deploy and manage his own applications. At the uppermost layer, i.e., the SaaS, the cloud provider offers software applications to its users.

Virtualization technologies are at the basis of any cloud infrastructure because they enable a more efficient and flexible management of the infrastructure itself. Different kinds of virtualization techniques are often used in a cloud environment [6]. *Server virtualization* is employed when one or more virtual machines (VMs) are placed on a single physical server. The resulting benefits are not only to increment servers utilization and lower power consumption in most cases, but also to improve flexibility, because virtualization unique features such as VMs live migration can be exploited. With *storage virtualization* logical volumes are built on top of raw block devices. Logical volumes virtualize physical storage by offering an added level of indirection in the storage stack, managed by a software layer. *Network virtualization* aims at providing multiple overlay networks on top of a shared physical network infrastructure, there-

VTDC'12, June 18, 2012, Delft, The Netherlands.

fore shifting every configuration issue from the underlying physical devices to more flexible software components. Finally, we refer to *desktop virtualization* as the consolidation of multiple desktop machines into one or more servers that are often built on top of a virtualization stack and are accessible by cheap terminals named *thin clients*.

In this paper we propose a system architecture for a IaaS private cloud which exploits all the virtualization techniques exposed so far. It has been designed with modularity, extensibility, and dependability in mind and aims to be general purpose, that is, it is not restricted to any particular use case or service. Therefore, it can be used by enterprises to host any kind of applications, ranging from web services without any performance requirement to soft real-time applications with time constraints. To achieve such a flexibility we have introduced a decoupling layer between the computational and storage resources and the possibility to homogeneously manage both virtualized and non-virtualized environments. For example, we can host web services on a virtual machine on a shared virtualization host with a disk on shared storage, as well as a real-time application on a dedicated physical machine with a disk on dedicated storage.

The proposed architecture takes into account different needs of a IaaS provider: we aim at building an infrastructure that can be reliable, scalable, easily manageable and flexible. Indeed, on the storage side, it offers different kinds of replication, volume management, and easy asynchronous backups; on the network side, it is based on a redundant network topology in order to maximize both availability and throughput; on the computational side, it merges together different virtualization technologies and physical machines.

We have implemented the proposed architecture using only free software. To show its effectiveness, we have realized over it a Linux Terminal Server Project (LTSP) cluster: a terminal server for Linux that allows many people to simultaneously use the same computer using low-powered thin clients, that is, a desktop virtualization solution. This project is now in a production state, being used by approximately 200 users for their everyday work.

The rest of the paper is organized as follows. In Section 2 we present an overview of the overall architecture. In Section 3 we describe in detail the storage and the network components, while in Section 4 we present the computational resources arrangement. Section 5 describes the LTSP cluster offered upon the cloud architecture and we use it as a case study to prove the effectiveness of our design. Finally, we conclude in Section 6 and provide hints for future work.

2. ARCHITECTURE OVERVIEW

The architecture we present in this paper has the main objective to decouple storage resources from computational resources. Such a decoupling makes the entire infrastructure more scalable and more flexible because it allows the administrators to add or remove storage systems without any impact on the computational resources and vice-versa. The system architecture is partitioned in two logical levels, respectively named *back-end subnet* and *front-end subnet*. The back-end subnet is the architectural component aimed at both managing the storage resources and offering the storage as a service to the front-end subnet, while the latter contains computational resources, which are then attached to storage services to obtain high-level services which are finally exposed to the end users. In addition to being a gateway to storage facilities, the back-end subnet also offers a centralized management of the services provided by the front-end subnet to the end users.

3. BACK-END SUBNET

We have designed the back-end subnet in such a way to obtain a clear separation between the services offered to the end users and the hardware/software needed by those services. The basic idea is as follows: a service is just an application that performs a task required by an end user, where each application requires several system resources (CPU, memory, disk, and network) with different needs. These resources have to be assigned by the system administrator in an transparent way with respect to the users, whose expectation is to have a fully functional service without taking care of their administrative issues. Therefore, the back-end subnet goal is to provide all the basic infrastructure facilities, such as a redundant storage management and a highly available management of computational resources, that should be managed as much as possible through a centralized tool to facilitate the system administrator tasks. In the following, we analyze each component of the back-end subnet.

3.1 Redundant Network Topology

A basic requirement of every dependable infrastructure is a dependable network topology. We introduce a redundant network topology that can (i) scale according to the number of used physical links and (ii) automatically re-arrange itself by excluding malfunctioning links without any service interruption. The presented topology is generic and does not depend on the physical medium used to implement it: therefore, it is possible to use links based on Fiber Channel over optical fiber or Ethernet links on copper cables.

Left side of Figure 1 shows the components belonging to the back-end subnet. There are two storage subsystems (primary and replica) and two highly available storage gateways, that are interconnected by a redundant network topology, except for the connection between the storage gateways and the storage subsystems. The latter can also be replicated, whether the employed storage subsystems support multipathing; since not all storage subsystems support multi-path connections, we depicted only a single link, even if our implementation supports multi-pathing. We also observe that the network topology supports any number of links among its nodes: although we depicted only two links between the storage gateways and two links between each single storage gateway and the two switches, any number of links can be used, depending on the bandwidth and availability requirements. Specifically, in our implementation we use a single 4 Gbps Fiber Channel link between each storage subsystem and its storage gateway, while we use four 1 Gbps Ethernet links between each storage gateway and the border switches. Finally, the connection between the two storage gateways is a 4 Gbps Fiber Channel.

We designed the network topology with the goal to maximize both dependability and throughput, trying also to minimize the hardware costs, since storage subsystems are often expensive Storage Area Networks (SANs), that require expensive Host Bus Adapters (HBAs) and SAN switches to connect to. The introduction of two storage gateways decouples the native storage connections from where the storage needs to be attached. In this way, using the storage gateways as front-ends for the storage system, we reach the following

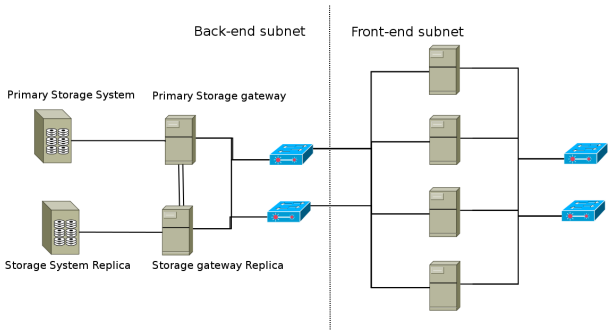

Figure 1: Network topology.

objectives: (i) to make the front-end servers unaware of the storage type; (ii) to aggregate different storage systems for reaching maximum expandability: that is, we can attach multiple heterogeneous storage subsystems to each storage gateway, hiding such a complexity to the front-end servers; (iii) to use arbitrarily complex storage management policies and replication; (iv) to minimize the hardware costs: we do not need to buy a lot of expensive HBAs and expensive SAN switches because only two highly-available servers are directly connected to the primary storage.

The channel aggregation of server NICs is realized using the Linux Bonding Driver. It provides a method to aggregate multiple network interfaces into a single logical "bonded" interface. The behavior of the bonded interfaces depends on the selected operating mode. It is possible to distinguish between two families of operating modes: the first only provides high availability connections (by using either hot stand-by or broadcast links), while the second also adds load balancing mechanisms on bonded links. The only drawback of using load balancing mechanisms relies in a more difficult network debugging when problems arise.

The overall back-end subnet can be seen as a black-box *storage unit* because it offers a high-level and feature rich storage service to the front-end subnet; we note that two storage gateways may not be sufficient to manage large amounts of I/O when thousands of disks are attached, but storage units can be replicated by using a flat or a hierarchical approach. Therefore, storage units are a scalable way to implement a modular interface to storage subsystems, since each storage unit implicitly provides the same interface to the front-end subnet layer even if the actual implementation could be very different.

3.2 Redundant Storage

When designing an architecture for an IaaS provider, an important issue is to ensure data protection from system failures: although a short unavailability period of a given service can often be tolerable, data loss is certainly intolerable. To address this issue, we introduced a redundant storage schema in our infrastructure architecture.

Suppose that the primary storage subsystem is a SAN with several RAID arrays: each RAID array in a SAN can host multiple Logical Unit Names (LUNs), each of which is seen as a physical disk by the operating system of the storage gateway connected to the SAN. We manage to introduce the storage replication through a software layer positioned just over the plain disk seen by the operating system and represented by *Distributed Redundant Block Devices* (DRBD) [1].

DRBD defines two roles in a replication scheme, named *primary* and *secondary*: only a node with primary role can access data if not using a distributed filesystem. We can use different working modes, that range from fully synchronous to asynchronous replication. With synchronous replication the filesystem on the active node is notified that the writing of the block is completed only when the block made it to both disks of the cluster. Using asynchronous replication, the entity that issued the write request is informed about completion as soon as the data is written to the local disk and to the local socket buffer. Finally, with memory-synchronous replication, the filesystem on the active node is notified that the writing of the block is completed when data is written to the local disk and has reached the write buffer of the remote server.

The choice of the working mode to be used strictly depends on the storages type and the interconnecting network. The best scenario is when we have a fast primary storage coupled with a fast[1] secondary storage, connected with a fast and reliable network. The worst scenario is when we have a fast primary storage with a relatively slow secondary storage, coupled with a slow and unreliable network (think to geographic data replication). We do not consider the scenario in which we have two poorly sized storage subsystems. Under the first scenario, synchronous DRBD replication is surely the best choice, because it ensures maximum data protection with a negligible performance loss. On the other hand, in the latter case we must take into account more variables, such as global disk speed, buffers, and network. If geographic replication is not needed and the infrastructure relies on a fast local network, we can choose either memory-synchronous replication or asynchronous replication, depending on the global disk speeds and the buffers offered by the secondary storage controller: the choice has to be done carefully and really depends on the workload the storage is supposed to face. If we choose a memory-synchronous replication without having enough buffers we can introduce a bottleneck, but if we use asynchronous replication with a fast and adequately buffered secondary storage we are not working in an optimal way.

In our implementation, we have a fast SAN as primary storage and a slow NAS (a server with direct attached storage) as secondary storage, with a fast interconnecting network. From not reported experiments, we found that for this specific configuration the best operating mode is the asynchronous replication with network congestion control: whenever the active storage gateway detects a network congestion[2], it temporarily detaches the secondary storage, thus going in "Ahead/Behind" operational mode. In such a way, the bottleneck is temporarily detached from the infrastructure and data synchronization restarts as soon as the network congestion disappears; the data synchronization process only involves those data blocks that have been changed on the primary storage while it was in "Ahead" mode. This advanced configuration has been pursued to avoid buying an expensive secondary SAN thus lowering the total cost of ownership, but at the same time increasing the overall infrastructure reliability by adding a storage replica.

[1]We consider a storage system to be *fast* when it is adequately sized to sustain the submitted workload.
[2]With a fast and reliable network, a congestion is likely to happen only when secondary storage disks utilization is 100% and the buffer is full.

Similar considerations can be applied to each storage unit in our architecture. Although different storage units have the same network topology and the same external interface, they can have different interconnections and storage speeds. Therefore, we are free to choose different replication policies depending on the components of each storage unit, offering a better QoS to users that are willing to pay more and a best-effort QoS to thrifty users.

3.3 Volumes Management

A modern storage system, besides being reliable, needs to be flexible. Volumes management introduces a considerable degree of flexibility, providing features like online volumes resizing, volumes snapshotting, and hot disk addition or removal. Volumes are considered by an operating system just like partitions, since we can use them to hold root filesystems or data directories. We briefly describe below the feature offered by logical volumes.

Online volumes resizing is a key feature: whenever we find out that we allocated less space than needed for a volume, it allows us to expand the space and, if the filesystem has the support, we can also hot-expand the filesystem without rebooting a server and therefore without service interruption.

Volumes snapshotting has a twofold use: we can snapshot a volume either to rapidly have new operating system images ready to use or to make consistent backups. Snapshots are nothing more than simple volumes, with their own allocated space, but they are originated from existing volumes and they use the Copy On Write (COW) [8] optimization strategy.

Usually, when using volume snapshotting for operating system image cloning, we have a 1:N ratio of gold images and derived snapshots. Therefore, to avoid disk overload, it is necessary to have a read-only gold image and read-write snapshots, otherwise each single write on the source volume triggers the COW on each snapshot. On the other hand, when using volume snapshotting for consistent backups, we usually have a 1:1 ratio between snapshotted volumes and snapshots, also having the source volume read-write mounted and the snapshot volume read-only mounted. In this case, we only need to take care of the snapshot size, that has to be enough to hold changes made on the source volume during the snapshot lifetime.

Disk addition and removal features are a must for a flexible architecture. When business grows, we can certainly expect an increase in the data to be stored. A disk addition feature allows to buy only needed disks from time to time, while a disk removal feature allows us to reduce the number of disks if they are no more needed and to replace old disks with newer ones without service interruption.

In our architecture we used Logical Volume Manager (LVM) to implement volumes management. LVM introduces three abstraction layers: physical volumes, volume groups, and logical volumes. LVM Physical Volumes (PVs) are simple partitions or physical disks initialized by LVM. After being initialized, PVs are aggregated to form Volume Groups. LVM Volume Groups (VGs) are an aggregation of PVs. This abstraction level let us overcome the single disk (or the single RAID array) capacity. LVM Logical Volumes (LVs) are flexible partitions built upon VGs and provide every feature described above. Since LVs are managed by the operating system just like partitions, LVM can be positioned either under or over DRBD. Our choice was to position LVM over DRBD on the primary storage and under DRBD on the secondary storage. The motivation is again the asymmetric configuration of the primary and secondary storage: the primary SAN can scale up to 99 disks; the secondary NAS can only scale up to 10 disks. Using larger disk on the NAS we can host DRBD resources over LVM volumes that turns out in consolidating the disk utilization.

3.4 Complete Storage Architecture

Figure 2 unveils the complete storage architecture we have realized: we can roughly divide the figure in two columns. The left column represents the primary storage subsystem, while the right column represents the secondary storage. Reading the figure bottom-up, we find that disks on the primary storage are organized in a RAID array, seen by the operating system with the name `/dev/sdc`. Over the RAID array, we created the physical partition `/dev/sdc1`, backing device for the DRBD resource `imgos`. The latter contains the operating systems images that can be attached to virtual machines (VMs) deployed on the front-end servers. The DRBD device `/dev/drbd0p1` is initialized as a LVM PV and is then inserted into the `imgos_group` VG. `imgos_group` thus contains as many LVs as the number of operating systems images stored in this storage unit. Looking at the DRBD

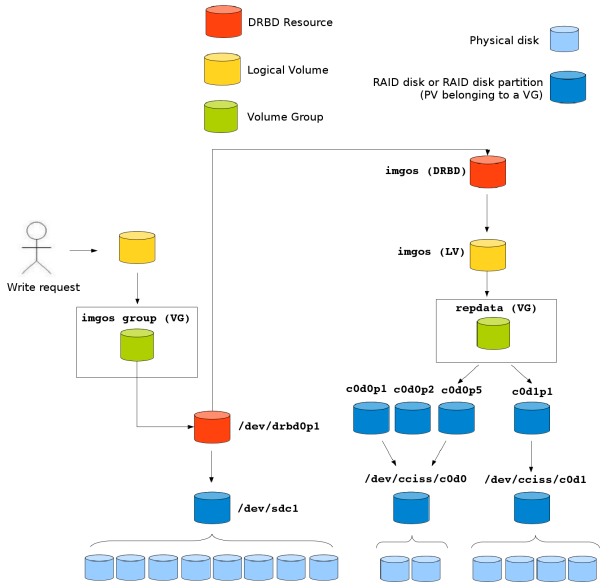

Figure 2: Data flow on the storage architecture.

level, it is coupled with a peer DRBD level on the secondary storage, whose backing device is a LVM LV `imgos`, which belongs to a VG named `repdata` and whose backing devices are two partitions on two different RAID arrays[3].

The designed storage architecture ensures high data reliability and throughput, flexibility and easiness of management, low costs, and finally a single asynchronous backup source. Data reliability is ensured by replication: local replication is achieved through RAID arrays, while remote repli-

[3]Actually, two RAID arrays on the secondary storage are not required, but we added them because the server used as secondary storage could not boot over a logical disk with GPT partitioning schema, which is required by logical disks larger than 2 TB.

cation is achieved through DRBD. Whenever the secondary storage fails, nobody will notice it and if the primary storage fails, the secondary one instantly replaces it. High data throughput is allowed by an asynchronous replication with congestion control (actually the perceived write speed is the one provided by the primary, fast storage). Flexibility and easiness of management are provided by LVM; to reduce the costs, we built a replicated storage only using free software and with an economic secondary storage. Last but not least, an interesting feature of the storage architecture is the ability to have a single asynchronous backup source: thanks to the LVM LV backing device for the entire DRBD resource on the secondary storage, we can instantly take a snapshot of the `imgos` resource and, since snapshots never modify original data, we can safely mount volumes found in the `imgos` snapshot, thus having a stable view of the entire storage unit data. This ensures easy, consistent, and economic backups: we do not need to backup every single server, but we can take them all together and backup them with a single backup application. This feature terribly lowers the effort of system administrators in keeping a working backup.

3.5 Choosing the IaaS Management Platform

In this section we briefly review and compare the major open-source platforms for the management of cloud infrastructures: Eucalyptus [3], OpenQRM [5], OpenNebula [4], and Nimbus [2]. We classify the various platforms according to the abstraction level they provide with respect to the underlying hardware and the customization degree. More detailed and recent analyses and comparisons of open-source cloud computing platforms can be found in [7, 9].

Eucalyptus is an open-source cloud management platform that offers the highest abstraction level among the others, by letting the user choose only from a fixed set of VM templates. It provides a framework similar to Amazon Web Services, by implementing interfaces compatible with Elastic Compute Cloud (EC2) and Amazon Simple Storage Service (S3), also realizing a distributed storage system called *Warlus*, which is designed to imitate Amazon's S3 distributed storage. Whenever a new virtual machine instance is requested to Eucalyptus, its operating system is copied from the storage system to the physical server (from now on *Compute node*) which will execute it.

Nimbus offers a higher customization degree compared to Eucalyptus, but only from the administrators' point of view: it let them configure at a finer level the VM instances that will be offered to end-users. Like Eucalyptus, Nimbus implements a storage system similar to Amazon S3. Both Nimbus and Eucalyptus are architected in order to decentralize resources: for this reason, each time a VM is requested, its image template is copied to the Compute node that will run the VM. In this way, both the platforms can achieve maximum scalability, as well as a good fault isolation because each Compute node is independent from the others. However, decentralizing storage systems also leads to a more complex management of replication policies as well as backup policies. For this reason, neither Eucalyptus nor Nimbus are compatible with our architecture.

OpenNebula is inspired by different principles compared to both Eucalyptus and Nimbus: it aims to provide a finer level of customizability for front-end users and its architecture tends to centralize resources by offering a central storage. A fine customizability level requires smart users because some details about the underlying infrastructure cannot be hidden anymore. Therefore, OpenNebula is more suited for private cloud where the entire environment is trusted and users have more skills. However, besides offering a central image storage, OpenNebula also provides the ability to run the image locally on the compute node, thus achieving the same flexibility of Eucalyptus and Nimbus.

The last platform we consider is OpenQRM, which provides the highest level of customizability, because it lets the administrators of the private cloud configure each single aspect of the datacenter. Its core is very small and its architecture is completely plugin-based, e.g., there are plugins to connect to different storage systems (NFS, iSCSI, AoE, etc.), as well as plugins to manage several virtualization technologies (primarily Xen and KVM, but also VMWare, Virtualbox, etc.). OpenQRM can be used just to administer the datacenter. Anyway, installing the *Cloud plugin*, we can exploit every typical cloud feature, like scheduled VM provisioning and deprovisioning and a pay-per-use model. Unlike the other platforms, OpenQRM lets the administrator of the cloud infrastructure configure the products offered to the cloud users at a finer granularity level: instead of configuring a fixed set of instances templates, it is possible to configure parameters like the quantity of CPU and RAM users are allowed to demand (with associated cost), as well as several possibility for data storage. Users can dynamically assemble their VM instances by choosing each single component from a drag-n-drop palette.

We chose OpenQRM for our infrastructure implementation, because it offers the higher customizability level, but our infrastructure is compatible with OpenNebula as well. Such a software can be added to our system architecture in two different positions. In case of a single storage unit, OpenNebula or OpenQRM can be installed on the two storage gateways; otherwise, in case of multiple storage units we need two additional servers at the back-end subnet boundaries to host the datacenter management software.

4. FRONT-END SUBNET

The front-end subnet, represented on the right side of Figure 1, is composed by a bunch of servers with lots of RAM and CPU. The connection schema is the same used for the back-end subnet: each front-end server can be equipped with a number of NIC, in our case each server has 8 NICs (2 quadport Gigabit Ethernet), where 4 links are directed to back-end switches and the remaining 4 links are directed to front-end switches. The channel aggregation is again achieved through the Linux bonding driver.

The front-end servers do not need any internal hard disk because they are network booted by OpenQRM, using DHCP, TFTP and iSCSI or NFS protocols with a minimal Linux distribution with only the software required to run a KVM virtual machine.

Because of virtualization, the front-end servers host a multilevel network stack (Figure 3). The first level is composed by several physical network devices (8 in our case), half of which is connected to back-end switches, while the other half is connected to front-end switches. The second level is given by link aggregation with bonding driver. We created two bond devices: `bond0` (directed to back-end switches) and `bond1` (directed to front-end switches). The first bond is used for storage access, while the second one is used for service delivery because the front-end switches are attached

to a generic internet gateway. Finally, the third level is the bridge, which allows virtual machines to use networking. Virtual machines support two networking configurations: `bridge` and `NAT`. The simplest configuration is bridge, because it involves network layers from physical to Logical Link Control (LLC), belonging to the Data Link layer. However, because of its simplicity, bridging configuration can only be used when complex network segmentation and isolation are not needed, because they can only be achieved using VLANs. VLANs imply switches configuration and, when the network becomes large, they can be very hard to manage. On the other hand, using NAT we raise the OSI layer to Network/Transport, so we can use more complex firewall rules to manage subnets communications at the price of losing the ability to use non TCP and non UDP applications.

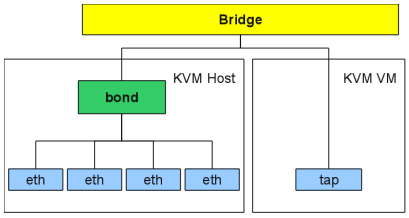

Figure 3: Network stack of the front-end servers.

Each front-end server has two network stacks: the first for data access and directed to storage gateways through back-end switches, and the latter for service delivery.

5. CASE STUDY: LTSP

Linux Terminal Server Project (LTSP) is a free and open source terminal server for Linux that allows many people to simultaneously use the same computer. The applications run on the server with a terminal known as thin client, which handles input and output. Terminals are typically low-powered, lack a hard disk, and are quieter than desktop computers because they do not have any moving part.

We implemented a LTSP cluster, designing it for a maximum of 200 users and using the system infrastructure presented in this paper. This specific implementation offers several features which make it a competitive solution for desktop virtualization. In the following, we describe the components of the LTSP cluster and how these components fit into the proposed system architecture.

5.1 LTSP Cluster Components

LTSP cluster components are virtual machines and storage provided by the underlying infrastructure. The first problem we have to face is the need of a shared `/home` hierarchy, because a single server cannot sustain the workload submitted by 200 users. The idea is that if we have a shared `/home` and multiple LTSP servers mounting that shared directory, the user could log on any server without noticing any difference.

NFSv3 is surely the lightest protocol we can use for `/home` sharing, but its lightness is countermeasured by the lack of some important functionalities, above all those for user authentication. To overcome this lack, a global name-space for users and groups is needed. The simple way to obtain such a global naming service is with Network Information System (NIS). NIS is a domain server capable of providing centralized authentication (and therefore a global naming service) as well as other useful services like global `/etc/hosts`, `/etc/ethers` and so on. In order to have multiple LTSP systems running in parallel, we need an additional highly-available system running NIS and NFS servers. It is now clear that we need at least two different appliance templates: the first for managing users and for sharing home directories (we will call it `domainmanager`), the latter for providing the actual LTSP service (we will call it `gentoo-ts`). Both these appliances are based on virtual machines created on top of front-end servers. In particular, we use one virtual machine for the first type and several virtual machines for the second type. In the following sub-section we will describe in detail the software and architectural issues regarding these appliances. Then we will present some results provided by the infrastructure monitoring system.

5.1.1 `domainmanager` *Appliance*

The `domainmanager` appliance has the main task of maintaining a common naming service for the entire system. In addition, it exports the `/home` directory to `gentoo-ts` appliances. Since `domainmanager` is the only appliance that directly mounts the filesystem containing the `/home` directory content, we also installed Samba on this appliance, in order to make it possible for user data to integrate with possible Windows clients. The software collection is enriched with CUPS, with which we provide a unified printer management, and clamav, a well-known Linux anti-virus.

Last but not least, `domainmanager` hosts the thin client Gentoo minimal system, a DHCP server and a TFTP server. Summing up, it hosts all the services, except for the terminal server service itself.

Differently from the front-end servers, `domainmanager` is supposed to face an intensive disk workload. For this reason and since NFS does not support recursive exports, both the root and home filesystems are mounted via the iSCSI protocol from the storage gateways.

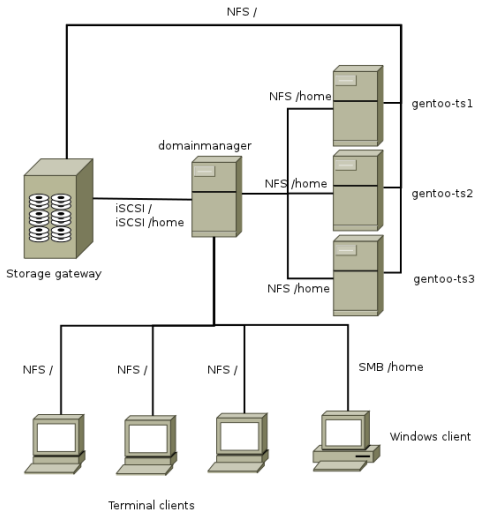

Figure 4: LTSP cluster overlay.

Figure 4 shows the architecture of the LTSP implementation and its dependence on the `domainmanager` appliance. We found out that possible `domainmanager` crashes just pause

the terminal servers executions because they all remain in a wait state for accessing data (users perceive a freeze of their desktop that can last from some second to a minute or two). When `domainmanager` recovers, the NFS TCP connection between the terminal servers and `domainmanager` is re-established and I/O queues are flushed. We observe that with this solution users do not lose their desktop session.

5.1.2 `gentoo-ts` *Appliances*

`gentoo-ts` appliances are VMs with a Gentoo operating system installed with a desktop/kde profile and the `ltsp-server` and `ltspfs` LTSP packages. The former contains a set of scripts used to exploit specific LTSP functionalities like local applications; the latter is a fuse filesystem to mount USB local storage devices and IDE/SATA CDROMs. `gentoo-ts` appliances boot over network by mounting a NFS root filesystem from the storage gateway and a NFS `/home` filesystem from `domainmanager` (Figure 4). They only differ for the configuration of the host name and IP address. Other configurations are exactly the same because we want to offer a transparent terminal server service, regardless from what server the user logs on. The workload `gentoo-ts` appliances are supposed to face is strictly related to the SSH protocol configuration, because LTSP heavily uses SSH, from X redirection to data transfer from/to local devices. The encryption largely impacts on the SSH configuration. Without encryption, server performances are boosted and a dual quad-processor server can manage even 50 users. Otherwise, when encryption is enabled, the processor load increases both on clients and servers, also requiring a relatively powerful CPU on thin clients. Summing up, SSH encryption must be enabled in case of security concerns; otherwise, in a trusted network unencrypted connections are preferable.

The load balancing of terminal server appliances is performed at session level and is carried out by the login manager, which runs on client machines and selects the best server available at login time through a server-state aware algorithm. To acquire server state information, the login manager queries every `ldminfod` daemon, which runs on every terminal server and computes a score for that server. Then, the login manager selects the server with the highest score. To compute each server score we use a linear combination of some server load indexes, specifically, the CPU utilization and the amount of available RAM; anyway, the formula used to compute each server score can be configured.

5.2 Infrastructure Monitoring

We now present some results obtained with the infrastructure monitoring system for which we adopted the open-source tool Zabbix [10], that is one of the most powerful and easiest platforms to work with (other popular open-source distributed monitoring systems include Ganglia and Nagios). Zabbix is based on a client-server approach: it is composed by a server backed by a DBMS (MySQL in our case), agents that have to be deployed on the monitored system and send all the information to the server, and a web interface, often installed on the server, to both configure the sensors and analyze the results. We installed the Zabbix server on the same server of OpenQRM and the agents on every machine (either physical or virtual). We configured the sensors to collect several types of data, ranging from CPU usage to disk and network I/O at every virtualization level. In the following, we present some data collected on the key nodes of the infrastructure in a typical working day.

Figure 5 shows the CPU idle percentage on 3 terminal servers (VMs with 16 CPU each) and the number of connected users. The CPU utilization follows the typical pattern of a working day, increasing from 8 AM and decreasing around 4 PM, but never growing more than 50% because we sized the system for 200 concurrent users. However, when the load increases, a new physical or virtual server can be added to the LTSP cluster in a few minutes and it will be immediately known to the login manager which carries out the load balancing for new user sessions. The opposite holds when the load decreases, e.g., during nights, when most servers can be turned off to reduce power consumption.

Figure 6 represents the network usage of `gentoo-ts-1` from the front-end side connected to end users. As expected, the outgoing traffic always overcomes the incoming traffic because the thin clients only send data to servers when a local device has to be used (e.g., a USB storage device). However, the peak outgoing throughput is around 32 MB/s, which can be sustained even with a single Gigabit interface.

The workload `domainmanager` is supposed to face mainly regards `/home` NFS serving. Figure 7 shows the CPU idle percentage of a VM with 6 CPUs, where we observe up to 80% usage during peaks. Although NFSv3 is a lightweight protocol, its CPU consumption cannot be ignored during the infrastructure capacity planning: in our case, NFSv3 requires almost 12 CPUs for 200 users (in the analyzed day only 6 as there are up to 100 users), that is about 20% of the entire LTSP cluster CPU power.

Finally, Figure 8 shows the throughput of `/home` volume over time. It roughly follows the same pattern of the CPU usage, but with much more spikes due to large file transfers.

6. CONCLUSIONS

We presented a system architecture for a reliable, scalable, easily manageable, and flexible IaaS cloud provider. To demonstrate its effectiveness, we described the implementation of a LTSP cluster that relies on the proposed architecture. The architecture reliability has been achieved by introducing high availability and fault tolerance in all the infrastructure components. The scalability regards both the storage and the computational power: the first is achieved by adding or removing disks or storage units, the latter by adding or removing physical machines from the pool of front-end servers. The easiness of management is ensured by OpenQRM. Finally, the flexibility is obtained at the storage level through LVM, which provides features like volume resizing, snapshotting, and shrinking, and at the computational resources level through OpenQRM, which manages uniformly different virtualized and physical resources, thus providing the administrator with the ability to migrate from virtual to physical environments and vice versa.

From our case study, we found that it is quite difficult to size the system for LTSP, because the workload varies highly from user to user. We managed to size properly the system; however, our sizing may not fit adequately for another organization because a single desktop connection is a gateway to a large number of activities a user can accomplish. Therefore, we plan to investigate the use of self-adaptation mechanisms to avoid underprovisioning or overprovisioning the system. Such adaptation mechanisms require various components, including a monitoring system, strategies to ana-

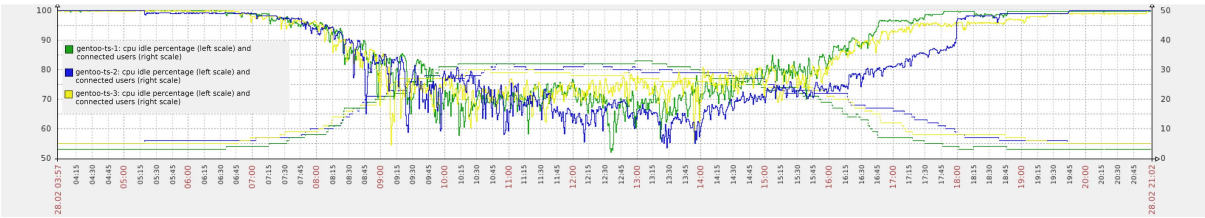

Figure 5: CPU idle percentage (left side of y axis) and number of connected users (right side of y axis) over time for Gentoo terminal servers.

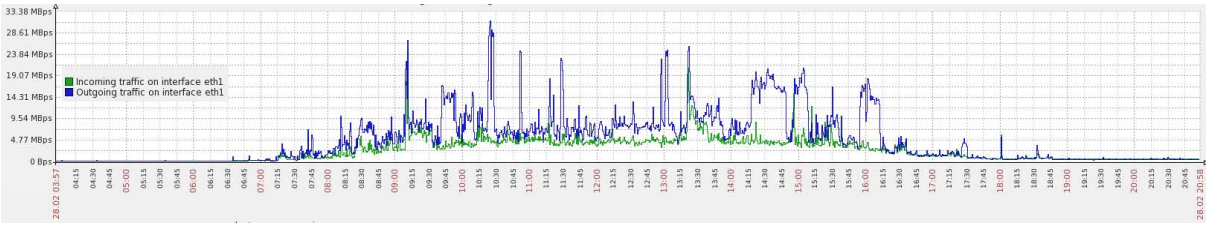

Figure 6: Network throughput over time for `gentoo-ts-1`.

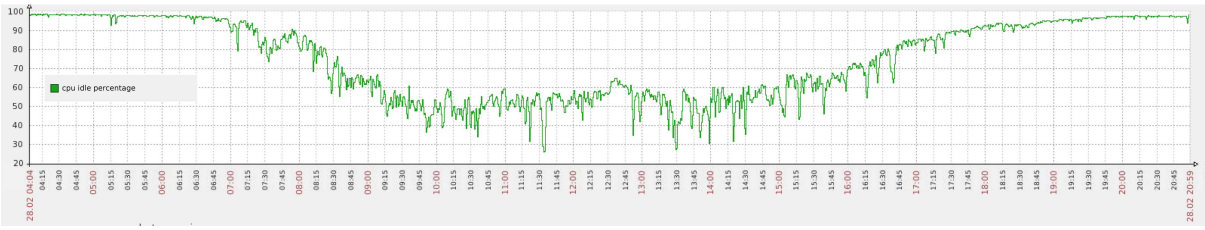

Figure 7: CPU idle percentage over time for `domainmanager`.

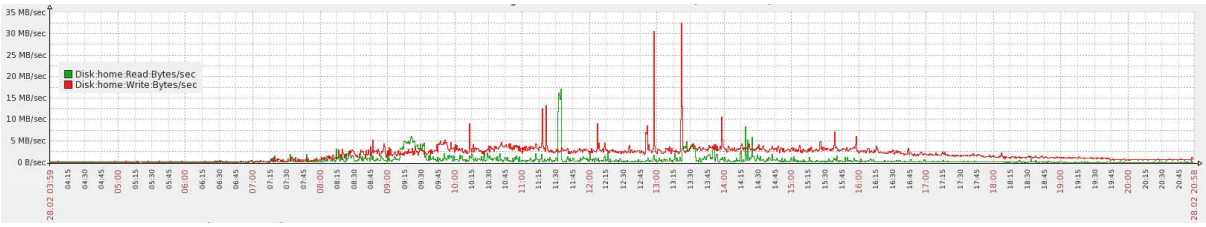

Figure 8: Disk throughput over time for `/home` volume.

lyze the monitored data, an adaptation policy, and a flexible system which can scale. In this paper we presented a flexible and scalable system with a monitoring component. In future work we will study techniques, such as PCA, to analyze the monitored data and adaptation policies to optimize performance with cost and energy constraints.

7. REFERENCES

[1] DRBD. `http://www.drbd.org/`.

[2] Nimbus. `http://www.nimbusproject.org/`.

[3] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. In *Proc. of IEEE/ACM CCGRID '09*, 2009.

[4] OpenNebula. `http://opennebula.org/`.

[5] openQRM. `http://www.openqrm.com/`.

[6] B. Rimal, A. Jukan, D. Katsaros, and Y. Goeleven. Architectural requirements for cloud computing systems: An enterprise cloud approach. *J. Grid Comp*, 9:3–26, 2011.

[7] P. Sempolinski and D. Thain. A comparison and critique of Eucalyptus, OpenNebula and Nimbus. *Proc. of IEEE CloudCom '10*, pages 417–426, 2010.

[8] D. Teigland and H. Mauelshagen. Volume managers in Linux. *Proc. of USENIX ATC '01*, 2001.

[9] S. Wind. Open source cloud computing management platforms: Introduction, comparison, and recommendations for implementation. In *Proc. of IEEE ICOS '11*, pages 175–179, 2011.

[10] Zabbix. `http://www.zabbix.com/`.

[11] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, May 2010.