Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Προσαρμογή συστήματος διαχείρισης εικονικών μηχανών για υποστήριξη υψηλού ρυθμού διεκπεραίωσης εντολών σε περιβάλλον υπολογιστικού νέφους

Διπλωματική Εργασία

του

Μπλιάμπλια Δημήτρη

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εργαστήριο Υπολογιστικών Συστημάτων
Αθήνα, Απρίλιος 2014

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Προσαρμογή συστήματος διαχείρισης εικονικών μηχανών για υποστήριξη υψηλού ρυθμού διεκπεραίωσης εντολών σε περιβάλλον υπολογιστικού νέφους

Διπλωματική Εργασία

του

Μπλιάμπλια Δημήτρη

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 9η Απριλίου, 2014.

.....	.....	.....
Νεκτάριος Κοζύρης	Νικόλαος Παπασπύρου	Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.	Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Απρίλιος 2014

.....

Μπλιάμπλιας Δημήτριος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικος Υπολογιστών Ε.Μ.Π.

Περίληψη

Στις μέρες μας, οι υποδομές Cloud Computing προσφέρουν ευελιξία, διαφάνεια, και ασφάλεια για την εκτέλεση ενός συνεχώς αυξανόμενου πλήθους εφαρμογών και υπηρεσιών. Οι υποδομές αυτές αποτελούνται κατά κανόνα από συστοιχίες υπολογιστών (clusters), χρησιμοποιώντας τεχνικές εικονοποίησης για να διαμοιράσουν τους φυσικούς πόρους σε εικονικούς, οι οποίοι θα χρησιμοποιηθούν από το cloud περιβάλλον. Οι πάροχοι των cloud υπηρεσιών, θέλουν οι υπηρεσίες τους να έχουν δυνατότητες κλιμάκωσης (scaling), και να λειτουργούν με χαμηλούς χρόνους απόκρισης (latency), ανεξαρτήτως του φόρτου των υπηρεσιών τους. Αρκετοί παράγοντες επηρεάζουν την απόδοση των cloud περιβαλλόντων, όπως το δίκτυο που χρησιμοποιείται για στη διασύνδεση των φυσικών πόρων, ή το υλικό που χρησιμοποιήθηκε για την υποδομή, όπως η CPU, η μνήμη, και ο δίσκος. Συνήθως, κάποιο εργαλείο λογισμικού αναλαμβάνει τη διαχείριση των κόμβων της συστοιχίας των υπολογιστών, όπως και την διαχείριση των εικονικών πόρων. Η παρούσα διπλωματική στοχεύει στη βελτίωση της απόδοσης ενός τέτοιου λογισμικού, και συγκεκριμένα του Ganeti, παρέχοντας υποστήριξη για εναλλακτικές μεθόδους που θα εξυπηρετούν τις απαιτήσεις του εργαλείου σε αποθηκευτικό χώρο. Η υλοποίησή μας, ενσωματώνει την CouchDB, μία NoSQL βάση διαχείρισης δεδομένων, χωρίς σχήμα, και προσανατολισμένη γύρω από έγγραφα στο Ganeti, και αξιολογεί την απόδοση του λογισμικού μετά από αυτή την τροποποίηση. Οι πρώτες μετρήσεις είναι ιδιαίτερα ενθαρρυντικές, καθώς παρουσιάζουν εμφανή βελτίωση στην απόδοση του Ganeti. Οι λόγοι αυτής της βελτίωσης θα παρουσιασθούν λεπτομερώς στη συνέχεια της παρούσας διπλωματικής.

Λέξεις Κλειδιά

cloud computing, cloud, εικονοποίηση, εικονική μηχανή, NoSQL, Ganeti, JSON, CouchDB, Synnefo, okeanos, διαμοιρασμός, python, κλιμάκωση (scaling), απόδοση (throughput), συστοιχία (cluster), κόμβος, instance, qemu, KVM, module, πακέτο δομή b+δέντρου, MVCC, ACID, CAP, views, daemons

Abstract

Nowadays, cloud computing exhibits agility, transparency, and security to the execution of a continuously increasing number of applications and services. Those infrastructures are designed on top of clusters of physical nodes, using virtualization techniques to appropriately separate the physical resources to create virtual dedicated ones, which will power the cloud environment. Cloud providers want their applications have the ability to scale, and operate in low-time latency, regardless of the load of the cloud services. Many factors affect the performance of those environments such as the network that is used for the intra-cluster communication, or the underlying hardware resources used, in terms of CPU, memory, and disk i/o. A software tool is commonly used that manages the physical nodes of the cluster, and the virtual resources as well. This thesis aims to improve the performance of such a tool, and specifically Ganeti's, by providing support for alternative engines to serve its storage requirements. Our design integrates CouchDB, a NoSQL, schema-less, and document oriented database in Ganeti, and evaluates the performance of the tool under the new storage layer. Early performance evaluations look very promising and show a noteworthy speedup on the performance of Ganeti, that will be discussed in details in the rest of the document.

Keywords

cloud computing, cloud, virtualization, virtual machine, NoSQL, Ganeti, JSON, CouchDB, Synnefo, okeanos, replication, python, scaling, throughput, cluster, node, instance, qemu, KVM, module, package, b+tree structure, MVCC, ACID, CAP, views, daemons

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τον αναπληρωτή καθηγητή κ. Νεκτάριο Κοζύρη για την ευκαιρία που μου έδωσε να ασχοληθώ με το συγκεκριμένο τομέα της επιστήμης των υπολογιστών στο Εργαστήριο Υπολογιστικών Συστημάτων, καθώς και για τη θετική συμβολή του καθόλη τη διάρκεια των σπουδών μου.

Η εκπόνηση της διπλωματικής αυτής εργασίας, αποτελεί έμπνευση του διδάκτορα Ευάγγελου Κούκη, τον οποίο θα ήθελα να ευχαριστήσω θερμά τόσο για την βοήθειά του και τις συμβουλές του σε επίπεδο σχεδιασμού και υλοποίσης, όσο και για τις τεχνικές γνώσεις που μου μετέδωσε κατά τη διάρκειά της. Επίσης θα ήθελα να ευχαριστήσω ιδιαίτερα τον Χρήστο Σταυρακάκη για την συμβολή του και τις παρεμβάσεις του που βοήθησαν στην επίτευξη της διπλωματικής μου εργασίας.

Τέλος ένα μεγάλο ευχαριστώ στην οικογένειά μου, για τη συνεχή τους στήριξη όλο αυτό το διάστημα των σπουδών μου, καθώς και στον κύκλο των φίλων μου για την ωραίες αναμνήσεις που μου προσφέρουν όλα αυτά τα χρόνια.

Μπλιάμπλιας Δημήτρης

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Nowadays, cloud computing has emerged as a popular computing model for enabling on-demand network access to a shared pool of computing resources such as storage, networks, application, and services. Cloud computing is not a completely new concept, but an evolution that has been ongoing for over a decade, if not since the very beginning of computers at around 60 years ago, when a *time-sharing* computing server served multiple users. Cloud provides computation, software, data access, and storage services on a pay-for-use model that do not require the end-user knowledge of the physical location and the configuration of the system that delivers those services.

Cloud computing providers need to support hundreds of thousands of users and services, and to ensure that they are fast, available, and secure. In order to accomplish this goal, the infrastructure that will be build should ensure a set of properties: scalability, reliability, security, and improved performance are only some of the properties that must be fulfilled for a cloud infrastructure. In order to meet up with those requirements, and with the increased data management needs in terms of users, and big data, new solutions appeared. The adoption of the NoSQL technology as a viable alternative of the relational databases has become a fact. NoSQL is increasingly considered a proper solution for the cloud needs, especially as more organizations recognize that operating at scale is better achieved in clusters of simple, commodity servers, and using a schema-less data model is often better for the variety of data that are being processed today.

It is common for cloud computing to be confused with the virtualization term. Although these two technologies are totally related, they have a significant difference. From an abstract point of view, virtualization is the software that separate the physical resources to create various dedicated resources, that power the cloud environment. Virtualization is a software that manipulates hardware, while cloud computing refers to a service that results from that manipulation. The technology behind virtualization is known as a virtual manager, which creates virtual computational environments from actual physical infrastructure.

1.1 Thesis motivation

In this thesis we will design, and study the performance impact of integrating a NoSQL database in a software used for managing clusters of physical nodes. The motivation

behind this thesis emerged from concerns about the performance, and scalability requirements of **Ganeti** [1], a software tool used for the physical node management of a cluster, and the low level VM management as well. Ganeti is used from **Synnefo** [2], an open source cloud software used to create massively scalable IaaS clouds. **Synnefo** [12], powers the ~**okeanos** public cloud service [11]. ~**okeanos** is an *IaaS*, i.e., *Infrastructure as a Service*, that provides virtual machines, virtual networks and storage services to the Greek Academic and Research Community. It is an open-source service that has been running in production servers since 2011, by GRNET S.A. [3].

Synnefo is a complete open source cloud stack written in Python, and has three main components providing the corresponding services:

- ***Cyclades***, Compute/Network/Image/Volume services.
- ***Pithos***, File/Object Storage services.
- ***Astakos***, Identify/Account services.

Synnefo manages multiple Ganeti clusters at the backend for handling the low-level VM operations. As we mentioned previously, improving the performance and scalability of Ganeti, by testing it under alternative storage engines, and specifically *CouchDB* [4], a NoSQL database system, was our motivation. In addition, a design document [5], that was proposed a long time ago by *Guido Trotter*, one of Ganeti's senior Engineers, amplified the conduction of this thesis.

1.2 Thesis structure

This thesis is organized in the following sections:

Chapter 2:
We provide all the necessary theoretical background for the concepts discussed in this thesis.

Chapter 3:
We present the architecture of Ganeti. A small documentation to facilitate the reader with the most basic components of Ganeti, and how they interact with each other.

Chapter 4:
We analyze two of the basic components of Ganeti by providing more technical details, and specifically the configuration data file and the job queue components. We also examine the main factors that prevent Ganeti from achieving better performance, and reduce its scalability capabilities. Next, we discuss the main reasons we chosen CouchDB to provide solution to some of those issues, and we make a quick presentation of its main features. Finally, we explain in details the design of

[1] `http://code.google.com/p/ganeti/`
[2] `http://www.synnefo.org`
[3] `https://www.grnet.gr/`
[4] `http://couchdb.apache.org/`
[5] `https://groups.google.com/forum/#!topic/ganeti-devel/jLvStCCTZ2Q`

the CouchDB driver, along with all the compromises we have to make during the implementation.

Chapter 5:
We evaluate the performance of the CouchDB driver. We compare it with the Ganeti's default filesystem approach, and we weight the pros and cons of each implementation. Finally, we extensively explain the reasons behind the differentiations that are arisen.

Chapter 6:
We provide our conclusion remarks, and our thoughts about future work for further improvements on the tool we designed, along with the final deduction of our work.

Chapter 2

Background

In this chapter, we will cover all the necessary theoretical background needed in order to familiarize the reader with the rest of the document. We will analyze the basic axis around that document, to avoid setting further understanding difficulties to an unfamiliar with the subject reader.

More specifically, Section 2.1 covers the virtualization topic since as we will see later, Ganeti is based on virtualization technologies. Section 2.2, explains the term of Cloud Computing and how it is evolved since it is first introduced in the computer science. Finally, Section 2.3 describes the new trend in the databases market, the *NoSQL* databases, as we used one of them to deal with some of Ganeti's performance and scalability issues.

2.1 Virtualization

In recent years, virtualization has become an important consideration and has gained popularity in many different areas such as server consolidation, cloud computing, corporate data centers, and the academic world. This is largely due to an increase in hardware performance of about ten fold in the past decade, the need to reduce the capital and operational cost to the minimum [9], and the desire to run multiple operating systems in a single host.

The term virtualization in computing, from a high level of view, refers to those technologies designed to provide a logical view of computer resources, and an abstraction layer between hardware systems and the software running on them. Virtualization traces its roots back to the *mainframes* of the 1960's and 1970's. There are various types of virtualization [19], and it can refer to a variety of computing concepts such as operating systems, user-space applications, storage services, computer network resources, and more. Maybe the most famous virtualization term is the *Virtual Machine*. In the early 1970's, Gerald J. Popek and Robert P. Goldberg defined the virtual machine as:

> *"A virtual machine is taken to be an efficient, isolated duplicate of the real machine."* [15]
>
> Gerald J. Popek and Robert P. Goldberg

The definition of virtual machine has evolved since then, and current virtual machine uses have not direct correspondence to any real hardware, necessarily, and are used in a number of subdisciplines ranging from operating systems to programming languages to processor architectures [18]. Depending on the correspondence degree of a physical machine, virtual machines can be classified into two major categories. The *Process Virtual Machines* and the *System Virtual Machines*.

A *process virtual machine* is a virtual platform that executes an individual process in isolation from the physical computer system. It is often called an *application virtual machine* and its objective is to only support the process it is assigned to. It is created and terminates as the process is created and terminates respectively. This approach allows a user to run a program that might otherwise be incompatible with the normal operating system. The virtualization software that implements a process machine often termed as *"Runtime software"*. Such virtual machines are usually suited to a programming language and build with the purpose of providing portability and flexibility to the language. That type of VM has become popular since the *Java Virtual Machine (JVM)* introduction, used by the Java Programming Language. They provide a high-level of abstraction and are implemented using an interpreter to transform the instructions written in a specific programming language into a machine language, which will run in the virtual environment that the VM creates.

In contrast, a *system virtual machine* provides a complete environment in which an operating system with multiple users and many different processes can coexist. With this type of VM, many different *guest* operating systems can run in a single *host*, independently, isolated, and transparently from each other. The guest operating system also provides access to virtual hardware including network drivers, I/O communication, along with a virtual CPU (vCPU) and memory. The software that implements a system virtual machine often referred as the *"Virtual Machine Monitor"*, VMM in short. These VMs usually emulate an existing architecture, and are built with the purpose to provide a platform for running programs when the real hardware is not available. In addition, having multiple VM instances on a single host leads to a more efficient use of computer resources, in terms of energy consumption and cost effectiveness; the key that lead to the Cloud Computing evolution.

2.1.1 Hardware Virtualization

Hardware, or platform virtualization refers to the technique used by the system virtual machines. In hardware virtualization, the *host* machine is the actual machine on which the virtualization takes place, and the *guest* machine is the virtual machine. The words host and guest are used to distinguish the software that runs on the physical machine from the software that runs on the virtual machine. VMs of that type use a separate software layer called *Hypervisor*. Hypervisor is the most basic virtualization component and it is responsible for the creation and the execution managing of the virtual machine. Hypervisors are classified on two different types depending on running directly on the host's hardware, *Type I*, or running within a conventional OS environment, *Type II*.

Type I hypervisors, also called native or "bare metal" ones, run directly on the physical hardware, control it, and manage the guest operating systems that run on a different level above the hardware, as Figure 2.1 denotes. They are completely independent from the operating system, and are also responsible for many basic operations like VM scheduling,

memory management, and more. They allow multiple commodity operating systems run concurrently and share conventional hardware in a safe way, but without sacrificing either performance or functionality. Most known hypervisors of that type are the *VMWare vShpere Hypervisor* [1] and the *Xen Hypervisor* [3].

Type II hypervisors, are running on top of a native operating system as a separate distinct software layer, and below the guest's operating system, as Figure 2.1 presents. This type of hypervisor heavily relies on the operating system, and is as secure as the OS itself. On the other hand, a native operating system provides the ability to the hypervisor to take advantage of the functions that are already implemented by the OS, like scheduling, memory management, device drivers, and more. The most known hypervisors of that category are the *KVM* hypervisor [2] which actually turns the Linux kernel into a hypervisor, *VirtualBox* [3], and the *VMWare Workstation* [4].

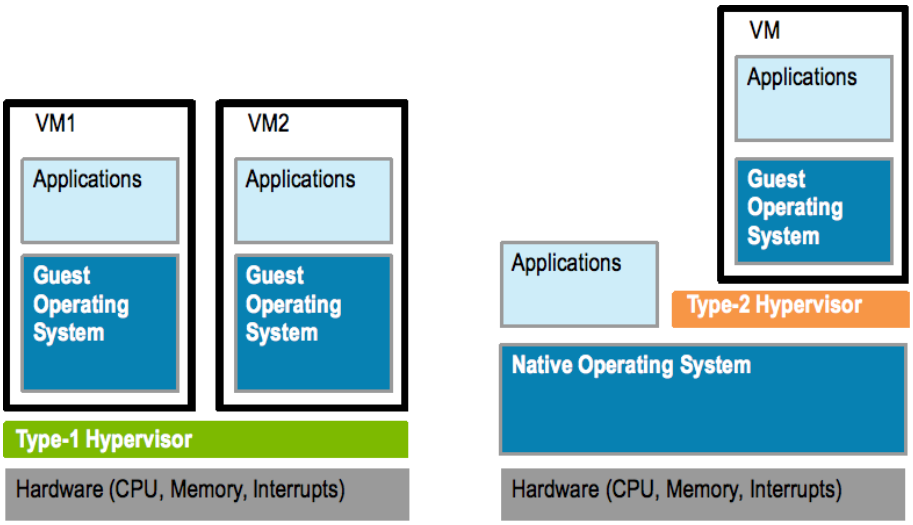


Figure 2.1: Hypervisor types

Finally, we should mention another important hardware virtualization technique, the *Emulation*. It refers to the replication of system functions from other, so that the emulator will act similarly to the emulated system. *QEMU* [5] is maybe the most known hardware emulator software, written by Fabrice Bellard [4]. It emulates central processing units through dynamic binary translation, and provides a set of device models, making it feasible to run a variety of unmodified guest operating systems. It also provides an accelerated mode for supporting a mixture of binary translation, for kernel code, and native execution, for user code, in the same fashion as VMware Workstation and VirtualBox do. QEMU can also be used purely for CPU emulation for user-level processes, allowing applications compiled for one architecture to be run on another [6].

[1] http://www.vmware.com/products/vshpere-hypervisor/
[2] http://www.linux-kvm.org/
[3] https://www.virtualbox.org/
[4] http://www.vmware.com/products/workstation/
[5] http://www.wiki.qemu.org/
[6] http://en.wikipedia.org/wiki/QEMU

2.1.2 Full Virtualization

Full Virtualization is a hardware virtualization type on which the virtual machine simulates almost completely the actual hardware to allow the guest operating system software, mainly, to be run transparently and in isolation from the rest system. That implies that every single feature of the physical hardware can be reflected into a virtual machine, including interrupts, memory access, and whatever other elements are used by the software that runs on the bare machine.

Full virtualization is possible only with the right combination of hardware and software elements. One of the most famous virtualization architectures *x86 architecture* [1], could not offer full virtualization until the additions of the Intel VT-x [7] and AMD-V [8] virtualization extensions, made at about 2005-2006. A key challenge for full virtualization is the interception and simulation of privileged operations, such as I/O instructions. Every operation performed within a virtual machine should not affect other virtual machines, or alter the state of the hardware. Many techniques have been developed to provide the appearance of full virtualization. An interest approach was implemented by VMware with a technique called *"Binary translation"* [20]. We will not present further techniques because are out of the scope of that document. *KVM* and *VMware* are well-known examples of full virtualization solutions.

2.1.3 Paravirtualization

This technique does not simulates a hardware environment. However, the guest applications are executed in their own isolated domains, as if they are running on a separate system. With Paravirtualization the term of *hypercall* is introduced. The guest paravirtualized operating systems should make a system call to the underlying hypervisor when it have to perform a privileged operation. By allowing the guest OS to indicate its intents to the hypervisor, improved performance and efficiency can be achieved, as each OS can cooperate to obtain better performance when running in a virtual machine. As a result, the guest OS have to be modified for the hypervisor, since the virtualization code is integrated into the operating system itself. *Xen* and *User Mode Linux* are examples of paravirtualization solutions.

2.2 Cloud Computing

Nowadays, the term *Cloud* has become one of the most popular words worldwide constituting a new trend in the computer science. Cloud Computing is an overloaded term with many formulated definitions, which make us realize the high interest of the topic. Although nearly everybody talks about cloud computing, the concepts remain somewhat unclear to many, because a new definition arises according to the field of interest. However, what we could say about all the different definitions, is that they have in common the concept of IT applications, infrastructures, and platforms provided on demand and standardized as services over the Internet where the resources are provided to the consumers logically rather than physically. According to that basic understanding and based on our literature

[7] http://en.wikipedia.org/wiki/X86_virtualization#Intel_virtualization_.28VT-x.29
[8] http://en.wikipedia.org/wiki/X86_virtualization#AMD_virtualization_.28AMD-V.29

review we will try to provide a definition from a computer resources perspective rather than a technical point of view:

Cloud Computing is a design solution for an IT deployment, based on virtualized resources offered over the Internet, where its services in terms of infrastructure, software, storage, networking, and more, are offered on demand by a service provider, guaranteeing scalability, security, reliability and high-availability, and can be billed based on a per-usage paying policy.

2.2.1 The evolution of Cloud Computing

Although the Cloud Computing became known to the public the last decade, that does not constitutes a new invention or a revolution for the computer science. It is more an evolution of already existing technologies rather than a new computing paradigm. The following short historical review of the development of computers and the Internet will show us that the idea of a centralized computer utility was always existed and inevitably lead us to the beginning of Cloud Computing [13].

Going back in time we stop in *1947*, the year when John Bardeen, Walter Brattain, and William Shockley first introduced the *transistor*. This is a milestone in computer evolution because it marked great advancements in the computer development. Computers evolved from simple calculating, Turing capable to general-purpose machines [9]. The underlying concept of cloud computing dates back in late 1950s, when large-scale *mainframe* computers become available in academia and corporations. *IBM 704*, in *1954* was the first mass produced mainframe computer with floating-point arithmetic. Eventually, in *1964* the *IBM System/360* followed. Mainframe computers was accessible via thin client/terminal computers. Primarily used by corporate and governmental organizations for critical applications and bulk data processing. The term originally referred to the large cabinets that housed the central processing unit and the main memory of early computers. The peripheral components that started to appear for that product family, in addition to further developments and the miniaturization of the mainframes computers lead to the appearance of *Minicomputers*, such as *DEC PDP-8*, in *1965*.

Mainframes were highly-costed computers. To make more efficient use of costly mainframes, a practice evolved that allowed multiple users to share both the physical access to computers from multiple terminals, as well as sharing their CPU time. The practice of sharing the CPU time on mainframe computers, became known in the industry as *time-sharing*. That trend of centralized, shared computer resources is very similar to the idea of cloud computing. Many researchers in the 1960ies talked about the need of computation to follow the electricity or telephone system paradigm and be organized as a public utility [7].

[9] Actually, *ENIAC* (aka Electronic Numerical Integrator And Computer), considered to be the first general-purpose computer, which announced one year earlier in *1946*, but after *1947* a burst in computer engineering occurs, with the first mass produced computers.

"Computing may someday be organized as a public utility just as the telephone system is a public utility ... Each subscriber needs to pay only for the capacity he actually uses, but he has access to all programming languages characteristic of a very large system ..."

Professor John McCarthy, at MIT's centennial celebration in 1961

Douglas Parkhill's, *1966* book, *The Challenge of the Computer Utility*, explores deeper this idea. With the *Personal Computer (PC)* development in the 1970ies significant performance leaps could be achieved, graphical user interfaces were established, and the continuing miniaturization eventually lead to the development of laptops and mobile devices. From *Intel 4004* first microprocessor in *1971*, since *1975 Altair 8800* first Home Computer, many PCs have been developed and even more followed from Apple, IBM, and more companies that entered the industry.

In the *1990ies*, the Internet achieved a real breakthrough with Tim Berners-Lee's invention of the *World Wide Web*. The concept of *Grid Computing* got established where a collection of computer resources from multiple locations used to reach a common goal. The first appearance of the term *Cloud Computing*, happened at *2007* some months after Amazon made the test version of its *Elastic Computing Cloud (EC2)* public. The rapid development of *Virtualization*, which allowed more efficient hardware utilization made the cloud computing a technological innovation. Nowadays more and more cloud solutions appear as mobile phones begin to overtake PCs as the most common Web access devices worldwide.

The following Figure 2.2, gives us an overview of the computing evolution.

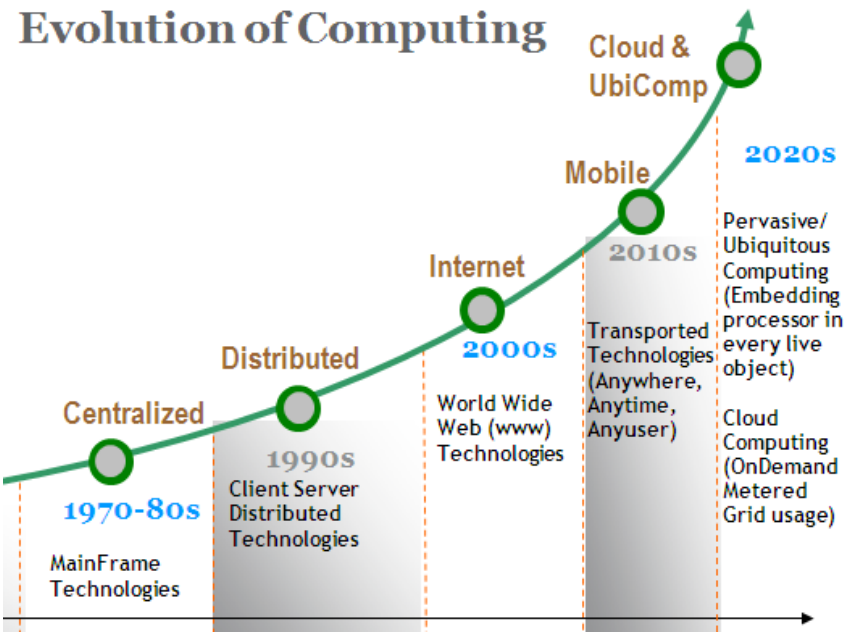


Figure 2.2: Computer history timeline

2.2.2 Service Models

Cloud computing services can be classified along different layers, according to the level of the capability they provide. There are three primary models, as shown in Figure 2.3, namely: Infrastructure as a Service (*IaaS*), Platform as a Service (*PaaS*), and Software as a Service (*SaaS*). These abstraction levels can also be viewed as a layered architecture where services of higher levels can be composed from the underlying layers.

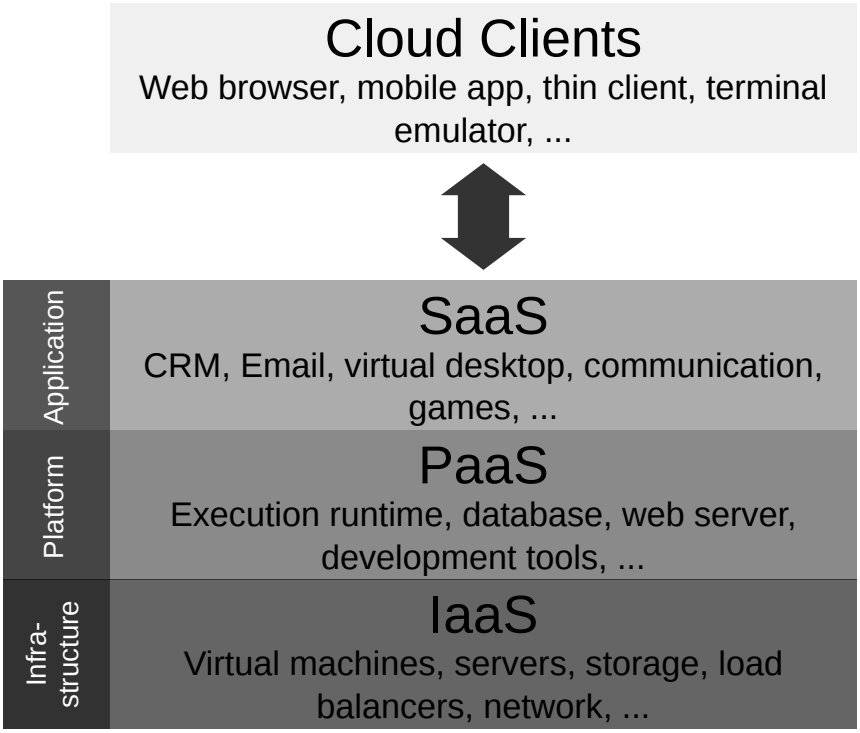


Figure 2.3: The layers of Cloud Computing

Infrastructure as a Service

It is the most basic cloud-service model. Providers of IaaS offer computers in terms of virtual machines, which are controlled by a hypervisor, such as Xen or KVM. Consumers can control and manage their systems in terms of operating systems, storage, applications, firewalls, and network connectivity but do not control the cloud infrastructure themselves. Examples of IaaS providers are the *Amazon Web Services (AWS)* [10], Synnefo [11], Openstack [12], Rackspace [13], GoGrid [14] and more.

Platform as a Service

In the PaaS model, cloud providers offer a computing platform, typically including an operating system, a web server, a database and a programming language execution environment. Consumers purchase access to those platforms, enabling them to

[10] `http://aws.amazon.com/`
[11] `http://www.synnefo.org/`
[12] `http://www.openstack.org/`
[13] `http://www.rackspace.com/`
[14] `http://gogrid.com/`

deploy their own software and applications in the cloud. The operating system or the network access are not managed by the consumers, which may have constraints on which applications they can deploy in the cloud. Some providers of a PaaS solutions are: Heroku [15], Openshift [16], AppFog [17], the Google App Engine Platform [18], and many other.

Software as a Service
In that service model consumers purchase the ability to access and make use of application software and databases that are hosted in the cloud. Cloud providers manage the infrastructure and platforms that run the applications. SaaS can be referred as *on-demand software* and is usually priced on a per-usage basis. The cloud users access the software most commonly from cloud clients. Salesforce [19] is the best known SaaS provider, which also considered to be the founder of the SaaS industry.

Other -aaS models
There are also many subsets of these models that are related to a more particular industry or market. Communication as a Service (*CaaS*), is a model used to describe hosted IP telephony services. Network as a Service (*NaaS*) provides network connectivity services and resource optimization allocations by considering network and computing resources as a unified whole. Database as a Service (*DBaaS*), is another submodel offered by the cloud providers. As it seems, in coming years almost anything will be provided by the cloud as a service.

2.2.3 Deployment Models

Deploying cloud computing can differ depending on consumers requirements. Four primary deployment models have been identified, each with specific characteristics, used for several services and user needs.

Private Cloud
The cloud infrastructure has been deployed, and is maintained and operated for a single organization. It can be hosted either internally or externally and managed internally or by a third-party.

Community Cloud
This deployment model is very similar to the private cloud, but the infrastructure is shared among several organizations with common interests and requirements. The cost for the cloud establishment is shared between organizations. The operation may be in-house or in third-party premises.

Public Cloud
The cloud infrastructure is available to the public by a service provider over the network. The most important technical difference compared to the private cloud is

[15] `https://www.heroku.com/`
[16] `https://www.openshift.com/`
[17] `https://www.appfog.com/`
[18] `https://developers.google.com/appengine/`
[19] `http://www.salesforce.com/`

the security consideration, which can be considerably different depending on services like applications, storage, and more.

Hybrid Cloud
As it names stands for, hybrid cloud is a composition of two or more different cloud deployment models. The various cloud models which implement the hybrid one, are unique entities. The unified cloud instead, has the ability to communicate through its sub-cloud interfaces, allowing data or applications to be moved among them, and offering the benefits of multiple deployment models. Various use cases for hybrid cloud composition exist, like a combination of a public and private cloud that support both the requirement to retain some data in an organization, and also the need to offer some services in the cloud community.

2.3 NoSQL databases

Over the last 15 years, interactive applications have changed dramatically, and so have the data management needs of those. The inception of Cloud Computing gave higher priority to the application scalability, resource utilization, and power saving. In addition to the growth in the global Internet usage, and the growing popularity of smartphones and tablets, it is very common for applications to have millions of users per day. Their usage requirements are hard to predict, because user number rapidly grows or shrinks, and it is important to dynamically support those differentiations. The large number of users created the need of handling large amount of data, so databases should be able to effectively deal with all this information [5].

The traditional SQL *Relational DataBase Management Systems (RDBMS)*, based on relational algebra [20], should be modified in order to meet up with the new requirements. At the beginning, companies tried the traditional approach. Invested to more and faster hardware as it became available. When that did not work, they tried to improve and make the current relational models scale by simplifying the database schema, introducing various caching layers, partitioning the data, and many more solutions in an attempt to respond to the requirements of the new community that was being developed. Although each of these techniques extended the currently rigid database model and addressed the core of the limitations that arose, they introduced additional overhead to the applications. As a result, the previously optimal relational database design, start to introduce limitations to the newly designed systems. This is mainly due to the fact that when the majority of the relational databases was designed, the predominant model for hardware deployment involved buying large servers attached to storage area networks ,i.e., SANs. In other words, the databases objective was to provide as much concurrent access as possible with the given machine's limitations.

The common architecture of relational databases was the main reason that failed to deal with scalability, latency, high-availability, failover options, speed, fault-tolerance and other requirements of the largest sites during the massive growth of the Internet. *NoSQL* databases started to emerge, providing solutions to the above requirements and in addition providing a great advantage; agility. NoSQL designed and evolved in a different environment with different needs and goals, and as a result provide better suited solutions for many of the today's data storage problems. Most agree that the term *No*SQL, stands

[20] `http://en.wikipedia.org/wiki/Relational_algebra`

for *"Not only"* SQL, showing that the target is not to completely reject the existing relational SQL models, but to overcome some of the limitations that exist. Database architects sacrificed many primary aspects of the relational model, such as joins or strong consistent data, while the schema devolved from strictly related tables with primary/foreign key relations to something much more like a key/value look up. Amazon's introduction of *DynamoDB* and the underlying paper [8] considered by many the first large, web-scale production of a NoSQL database.

2.3.1 NoSQl compromises

Companies want solutions that would scale, be fast, and operationally efficient. They also ideally expect operations to speed up by simply adding new commodity hardware, at almost a linear rate. One major bottleneck to accomplish that goal are the database systems. So, in order to achieve the desired behavior, some compromises should be made, and a bunch of tradeoffs should be taking into account [5].

The CAP Theorem

In *2000*, Berkeley's CA researcher Eric Brewer published the CAP Theorem [21], also known as Brewer's Theorem. What Brewer claimed is that it is impossible for a distributed system to continually maintain perfect *C*onsistency, *A*vailability, and *P*artition tolerance.

- *Consistency*: All node see the same data at the same time.
- *Availability*: A guarantee that every request receives a response about whether it was successful or failed.
- *Partition tolerance*: The system will continue to operate despite arbitrary message losses.

The theorem states that when designing a database distributed system you must make tradeoffs among the above features because you cannot simultaneously maintain all three of them. Peter Mell, a senior computer scientist for the National Institute of Standards and Technology (NIST), said that:

> *In the database world, they can give you perfect consistency, but that limits your availability or scalability. It's interesting, you are actually allowed to relax the consistency just a little bit, not a lot, to achieve greater scalability.*
>
> *Well, the big data vendors took this to a whole new extreme. They said that we are going to offer amazing availability or scalability, knowing that the data is going to be consistent eventually, usually. That was great for many things.* [14]

What Mell actually said is that maybe we need a balance. So systems should be developed with CAP tradeoffs, relative to operations that the product provides, rather than relative to the product as a whole. This is what a NoSQL solution actually does. It employs less constrained consistency models than traditional relational databases, but with higher availability and partition-tolerance.

[21] `http://en.wikipedia.org/wiki/CAP_theorem`

ACID vs BASE

In computer science, *ACID* is a set of properties which outlines the fundamental elements of transactions. In terms of database, a single logical operation on the data is called a transaction. ACID stands for Atomicity, Consistency, Isolation, and Durability [22]. In order to make a transaction more agile, and to deliver scalability, the NoSQL solutions should relax, or redefine some of those properties. Consistency and durability are the first "runners".

In distributed systems where there is a great deal of communication involving locks, scalability can not be easily achieved. One solution is to relax the consistency property and pass from *"strong"* consistency to something called *"eventual"* consistency. This actually means that updates made by a part of a system should become known to the rest parts of the system within a short period of time and not directly after the transaction is completed. For many applications the acknowledgment that the information will eventually arrive to all nodes satisfy the requirements. One other approach is using a concurrency control method called *Multi Version Concurrency Control (MVCC)* [23]. Both of those techniques add an extra overhead to the programmer of the application who is responsible of dealing with the coming information appropriately. An another property is durability. Many NoSQL solutions choose not to save data to disk at once, because writing to disk slows down the whole system, but keeping them in memory instead. A balance should be found between durability and speed of performing read/write operations. Eventually, that approach keeps a small window open where seemingly committed transactions can be lost. Many other approaches designed, but as with any other databases, when evaluating a NoSQL solution, you should choose depending on your application requirements [5].

The result of that *"relaxing ACID"* approach followed by NoSQL solutions, characterized by the *BASE* acronym:

- *Basically Available*: By using data replication or sharding among many different servers, we result in a system that is always available, even if subsets of data become unavailable for short periods of time.
- *Soft state*: In ACID systems, data consistency is one of the most painful requirements. On the other hand, NoSQL systems allow inconsistent data between nodes and leave that inconsistencies to the application developer.
- *Eventually consistent*: In contrast with ACID systems that enforce consistency at transaction commit, NoSQL guarantees consistency only at some undefined future time.

Figure 2.4, sums up everything discussed in the current subsection in a single image.

2.3.2 NoSQL Models

Relational and NoSQL models are totally different. Relational models have rigid schema which means that they require a strict definition of a schema prior to storing any type of data into a database. Changing the schema once data are inserted is a big deal, may be disruptive, and it is frequently avoided. In addition, relational databases take data and

[22] `http://en.wikipedia.org/wiki/ACID`
[23] `http://en.wikipedia.org/wiki/Multiversion_concurrency_control`

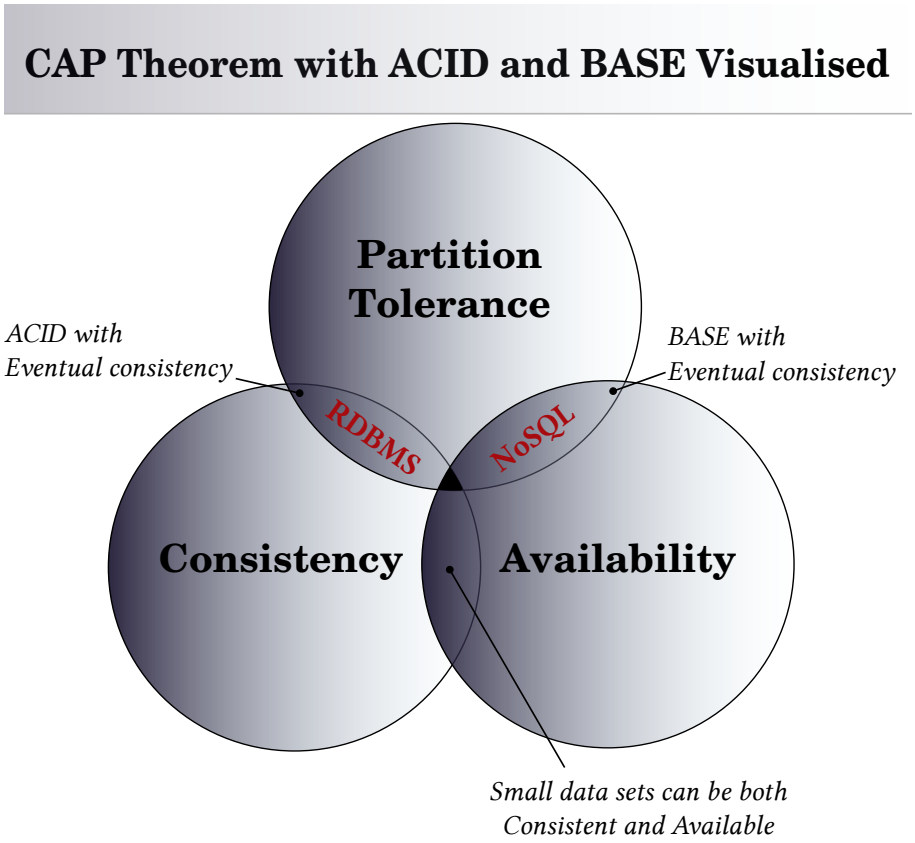


Figure 2.4: CAP theorem with ACID and BASE visualized

separate them into many different tables that contain rows and columns. Tables can be related through different foreign keys that are stored in columns as well. A simple look up of a data, requires a search among many tables and the subresults are combined before the final answer can be provided to the application. Similarly, a write requires data to be coordinated and performed on many different tables.

NoSQL databases follow a totally different approach. They are schema-less, allowing to freely add fields without having to define the changes earlier. The application is not disrupted while the format of the data being inserted changes. They also diverge from one another as well as from the RDBMS. There are three main representation models within the NoSQL family: Document-oriented, key/value, and graph related databases. *Key-value* databases allow the application to store its data in a schema-less way. The data could be stored in a datatype of a programming language or an object. Many different key/value types exist such as: KV-eventual consistency, KV-cache in RAM, KV-solid state or rotating disk and more. *Document-oriented* database store data in formats that are more native to the languages and systems they interact with. JSON or BSON, are commonly used as a simple dictionary/array representation. *Graph-based* databases store information about nodes and edges and provide simple, highly-optimized interface to examine the connections between them. We can classify NoSQL into more models like, Collection, or Columnar/Tabular oriented databases, but we chose to present the three most known categories of them. In Figure 2.5 [24], we present a bunch of the most known NoSQL

[24]Source: `http://blog.nahurst.com/visual-guide-to-nosql-systems`

databases, and the models in which each one belongs, in combination with the CAP theorem graph, to describe the various areas covered by each NoSQL database.

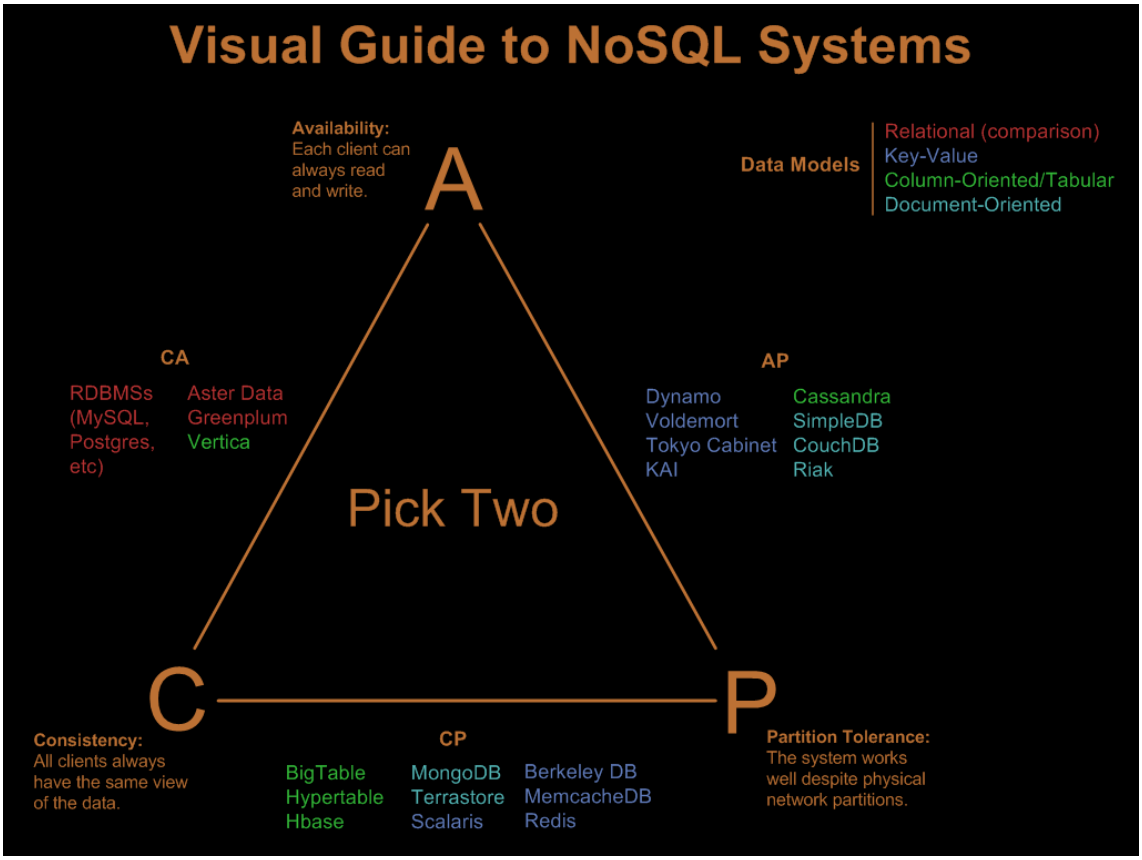


Figure 2.5: Visual guide to NoSQL systems

NoSQL models have been designed with the intention to handle large amount of data. Their main characteristics have been designed and developed with respect to scaling and performance. NoSQL solutions provide auto-sharding techniques. A NoSQL database automatically spreads data across servers, without requiring application to participate. Servers can be added or removed from the data layer without application downtime. Most NoSQL databases also support data replication, storing multiple copies of data across the cluster, or even more across data centers to ensure high-availability and failover from hardware failures. In addition to distributed data, they provide distributed processing techniques, mainly based on MapReduce, providing them with powerful query capabilities even across hundred of servers. Furthermore, to reduce latency and increase sustained data throughput, advanced NoSQL database technologies transparently cache data in the system memory. Summing up, NoSQL vs RDBMS debate will continue. Each has its advantages and weaknesses, and neither will entirely replace the other. Spending some time on understanding how the NoSQL tradeoffs will impact the application's design, may result in a solution that fits the product's special needs, sometimes better than a traditional RDBMS solution would.

Chapter 3

Ganeti backend

This chapter concentrates on Ganeti. We will discuss the architecture of Ganeti, and we will analyze in details its most basic parts and how they interact with each other. We will try to provide small documentation to familiarize the reader with this tool, avoiding presenting superfluous information. Summing up, the objective for the reader is at the end of this chapter to have a comprehensive view of Ganeti and its basic structure.

3.1 Overview

Ganeti is a software tool to manage computer clusters, and also assumes the management task of the virtual instances of the cluster. Is is being developed by Google and is an Open Source Project since 2007. It is build on top of existing virtualization technologies, such as XEN or KVM hypervisors, and other open source software. It also uses LVM for disk management, optionally DRBD for disk replication across physical hosts, and other disk templates such as RBD, external shared storage providers and more.

Ganeti uses a daemon-based model. Each daemon deals with specific tasks that the cluster has to face, and communicates with other daemons using various protocols, mainly HTTP based, and custom ones like LUXI. Many of those daemons are written in Haskell, but most of the project's code is in Python. Ganeti is actually a wrapper around hypervisors. Once installed, the tool will take over the management part of virtual instances. It also makes it convenient for system administrators to setup and handle clusters of physical nodes.

Some of the main features that Ganeti provides, and controls are the following:

- Cluster management of physical nodes.
- Support for XEN virtualization.
- Support for KVM virtualization.
- Support for virtual console to control instances ,e.g, VNC.
- Support for live instance migration.
- Support for virtio or emulated devices.

- Disk management: plain LVM volumes, files, Across-the-network RAID1 (using DRBD) for quick recovery in case of physical system failure.
- Export/import mechanisms for backup purposes or migration between cluster.
- Fast and simple recovery after physical failures using commodity hardware.

3.2 Terminology

In this section, we provide a small introduction of the most basic Ganeti terms, in order to facilitate the reader in the rest of the document. In Figure 3.1, we present an abstract Ganeti architecture, which will help us to briefly explain those terms.

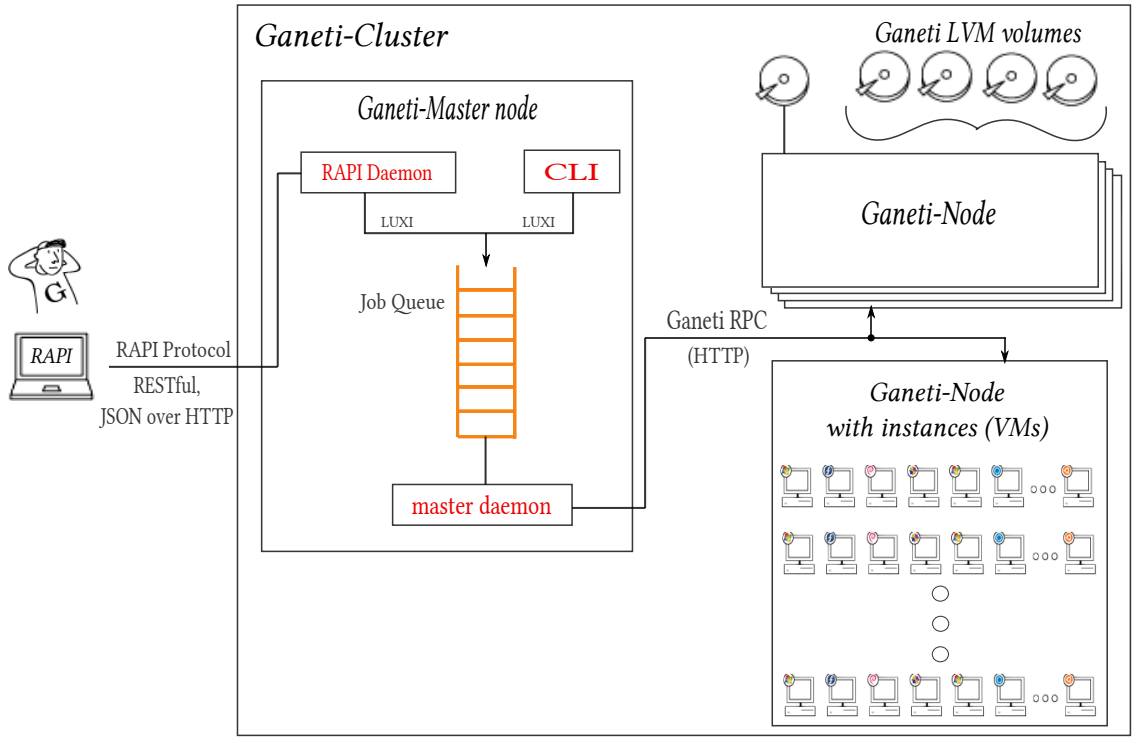


Figure 3.1: Ganeti base components, version 2.7.2

Cluster
A set of computers (nodes) working together to provide a coherent, reliable, scalable, highly-available virtualization service under a single domain.

Node
A physical machine which corresponds to the basic cluster infrastructure. If they host no instances, nodes can be added and removed at will from the cluster. They do not have to be fault-tolerant in order to achieve high availability for the instances they host. The loss of a single node, in a HA cluster, will not cause disk data loss for any of the instances it hosts.

A node belonging to a cluster can serve different roles; VM-hosting and/or Administrative roles, which will be explained later in this document. Independently of its

role, nodes can be in different statuses like online, drained or offline. Depending on the role they serve, each node will run a set of daemons:

- ganeti-masterd
- ganeti-noded
- ganeti-rapid
- ganeti-confd

Instance
A virtual machine that runs on a cluster. Instances in Ganeti are highly-available entities, that can also become fault-tolerant depending on the disk template they use, e.g., DRBD. An instance has various parameters which can be modified either at instance level or at cluster level via cluster default parameters. Those parameters can be classified in three main categories: hypervisor related parameters, called `hvparams`, general parameters, called `beparams`, and network interface parameters, called `nics`.

Disk Template
The layout disk type for the instance. Instances in Ganeti see the same virtual drive in all cases, but the node-level configuration varies between them. The available storage templates are the following:

diskless
This creates an instance with no disks. Useful for testing purposes, or other special cases.

file
Disk devices will be regular plain files. No redundancy is provided.

sharedfile
Disk devices will be regular plain files under a shared directory. This option allows live migration and failover of instances.

plain
Disk devices will be LVM volumes.

drbd
Disk devices will be drbd on top of LVM volumes, compatible with DRBD versions 8.x.

rbd
Disk devices will be rbd volumes, short for RADOS block device, residing inside a RADOS cluster.

blockdev
Pre-existent block devices will be used as backend for its disks.

ext
The instance will use an external storage provider as disk backend, through the ExtStorage Interface, using ExtStorage providers.

***Primary* and *Secondary* concepts**
An instance has a primary node, and depending on the disk configuration chosen might also have a secondary one. Every DRBD instance runs in its primary node and uses the secondary for disk replication and fault-tolerance. When those terms

used in node level, they refer to the instances having the given node as primary and secondary, respectively.

IAllocator
A framework for using external user-provided scripts to automatically compute the placement of new instances on the cluster nodes. This eliminates the need to manually specify the exact locations of an instance addition/move, and make the node evacuate operations an easy, and common cluster operation.

In order for Ganeti to be able to use those scripts, we should place them under the `$libdir/ganeti/iallocators` folder path.

Jobs and OpCodes
A *Job* in Ganeti is the basic operation to modify the cluster's state. A job consists of multiple *OpCodes* internally, short for "Operation Code". This is the basic element of operation in Ganeti. Most of the commands in Ganeti are equivalent to one opcode, or in some cases a sequence of opcodes, all of the same type ,e.g., shutting down all instances in a cluster. The opcodes of a single job are processed serially, but different jobs can be executed in parallel, in different order than they have been submitted, depending on hardware resource availability, locks, or priority given by user.

3.3 Architecture

As we mentioned earlier in this section, Ganeti has a daemon-based architecture. Every Ganeti-related command, (`gnt-*` commands), is an individual client which "talk" to the master daemon who executes every cluster operation.

In Figure 3.2, we present the architecture of a Ganeti cluster in a more detailed form than in Figure 3.1, and we show how the most basic daemons and elements interact with each other.

Nodes are the basic cluster infrastructure in Ganeti. They serve different roles and they can, and usually do, serve more than one. We could group node roles into two major categories; The *Administrative* and the *VM-capable* nodes. Nodes belonging in the first category, can modify the cluster state, or take part in cluster related decisions like the master node voting procedure. Nodes in second category can simply hosts instances (VMs).

In more details, a node can belong in one, or more of the following roles:

Master
It is the cluster coordinator node and it holds the authoritative copy of the cluster configuration. Every decision that could affect the cluster state managed by this node, because it is the only node which can execute commands. Only one master should exist every time in a Ganeti cluster.

Master Candidate
Nodes in this category have the full copy of the live cluster configuration and jobs. Only nodes belonging in this role can become master. This set of nodes called *candidate pool*, and there is also a parameter called `candidate_pool_size`, which represents the number of candidates the cluster tries to maintain, automatically. Because the `candidate_pool_size` can have a huge impact in Ganeti performance,

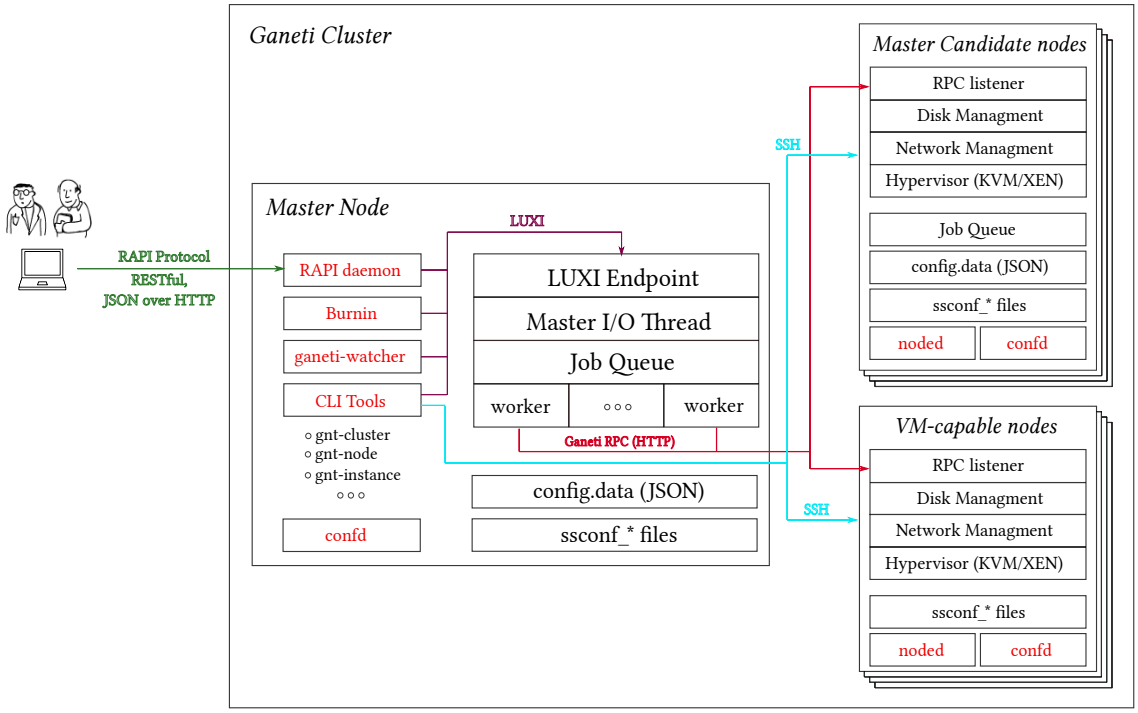


Figure 3.2: Ganeti architecture, version 2.7.2

for reasons which we 'll explain later, it can be configured during initialization or modified via cluster related commands (`gnt-cluster *`).

Master Capable

Nodes in this category are not master candidates, but can become and promoted to master node, in cases when nodes in the candidate pool are less than the desired size. In such case, a randomly selected master capable node is promoted to master candidate. We could disable this flag and exclude some nodes from being master candidates in case when they have a less reliable hardware and we do not want to store sensitive information to them.

VM Capable

This is the default node state and means that the node can host instances. More specifically, the node will participate in instance allocation operations, capacity calculations, cluster checks, and other operations.

Offline

Nodes having this flag set have some special characteristics. They are still recorded in the Ganeti configuration, and can only take part in the master voting procedure, to ensure consistency. They are not allowed to become master. Enabling this flag to a master candidate node will demote it from candidate possibly, causing another node which is master capable to be promoted. Additionally, these nodes are not allowed to host primary instances. The main reason this role was added to Ganeti was to allow broken machines that are being repaired to remain in the cluster without introducing further problems.

Drained
: Nodes in this state, will not participate in instance allocation operations, but all other operations as queries, or starting and stopping instances, are working without any restrictions. The actual intention is that nodes in this role have some issue and they are being evacuated for hardware repairs.

Previously, we made some references to Ganeti *Cluster Configuration* and *Jobs* which are stored in the system as files. These files are stored under the `/var/lib/ganeti` directory, and actually form a database for the cluster. Every single piece of information that the cluster needs to operate normally is stored in those files.

3.3.1 Cluster Configuration

The cluster configuration is a set of files which are present in all, or a subset of nodes depending on their usage. We could group them into three main categories, namely:

config.data
: The cluster configuration database is a single JSON config file called `config.data`. The master node keeps a valid version of this file and it also replicates it to the master candidate nodes, for reliability reasons. Ganeti has a special way to handle `config.data` updates; It holds the `config.data` both in master memory and disk. The canonical version of the config exists at every moment in the master node memory, and the disk version will be updated from there. Every operation that updates a single object in the memory version of the `config.data` automatically causes a flush of the whole file to disk. If the config does not flushed successfully to disk, the operation will fail. The `config.data` file contains information about all the major Ganeti objects such as cluster, nodes, instances, networks, and their attributes. We will extensively talk about the `config.data` structure in the next chapters.

ssconf_*
: Besides the objects contained in the `config.data`, which change quite often, Ganeti also holds a set of configuration files which contain information that does not change frequently and needs to be present to all Ganeti nodes. These files are stored in the same directory as the `config.data` file and start with a `ssconf_` prefix. For example, the ssconf file which contains the master IP value is called `ssconf_master_ip`, and so on. The main reason for the existence of the ssconf files is that the most frequent Ganeti operations should not need to contact the master node and overload him. In addition, we want some information to be accessible at every moment, even if the master node is down, so that we can use it from services external to the cluster, and avoiding the single point of failure that a master hard shutdown could introduce.

SSL certificates
: Ganeti uses OpenSSL for encryption on the RPC layer, and SSH for executing commands. These SSL certificates are stored under the same directory as the rest of the configuration files and exist in all Ganeti nodes. The SSL certificates are automatically generated when the cluster is initialized, and are copied to the newly added nodes automatically along with the master's SSH host key. The cluster SSL key is stored in the `server.pem` file. There is a similar key for the RAPI daemon, the

`rapi.pem` file. The `spice.pem` and `spice-ca.pem` files are used by SPICE connections to the KVM hypervisor, the `hmac.key` is used by the `ganeti-confd` daemon, the `cluster-domain-secret` file is used to sign information exchanged between separate clusters via a third party, and finally the Ganeti `known_hosts` file are all the certificates maintained by Ganeti.

3.3.2 Jobs

Jobs are the basic Ganeti operation, and the only way to modify the cluster's state. They are stored as individual files in the file system, and serialized using JSON format, which is the standard Ganeti serialization mechanism. A job consists of one or more opcodes. That list of opcodes is processed serially, and if an opcode fails, later opcodes are no longer processed and the entire job will fail.

At any time, each job and each opcode can be, in a different status depending on the stage of its execution. The job status is actually the status of its first processed opcode. A complete status description follows:

Queued
: The job/opcode has been submitted, but has not been processed yet.

Waiting
: The job/opcode is waiting for locks, or other factors, to proceed.

Running
: The job/opcode is currently being executed.

Canceled
: The job/opcode is waiting for locks, but is has been marked for cancellation by the user. It will never return to the *Running* status.

Success
: The job/opcode ran and finished successfully.

Error
: The job/opcode has failed while executing, or the master daemon stopped before the job finishes its execution.

While job opcodes execute serially, jobs do not. Their execution order depends on a variety of factors, apart from their incoming order, like their ability to acquire all necessary locks, their priority, or probable dependencies with other jobs. At any time, there are jobs that can be in one of the above statuses. Similarly to the global cluster configuration files, jobs are stored under a directory in the configuration path called *queue*, which is located by default under the `/var/lib/ganeti/queue` path. The *Job Queue* structure speeds up operations because every job which is ready for execution can run independently and so in parallel with the other jobs. In addition, storing the jobs in a common folder makes it more convenient for the user to handle and watch their progress, independently of Ganeti, and it also makes the consistency checks that Ganeti does to the job list and jobs themselves a simpler procedure. Queue structure also gives us the choice to prevent new jobs from entering it by enabling the *drained flag*. This is a feature used mainly in cases when we

have to make maintenance-related operations to the cluster and we do not want any new incoming jobs affecting us.

Besides the regular jobs, the *Job Queue* structure always contains three more files, even if there are not any jobs running, pending, or canceled in the cluster. Those are the `version` file, which denotes the queue format version, the `lock` file, which is opened by the queue managing process in exclusive mode, and the `serial` file, containing the last job ID value used. Listing 3.1, presents a high-level view of the Ganeti Job Queue structure:

```
/var/lib/ganeti/queue/
  job-1   (JSON encoded job description and status)
  [...]
  job-99
  job-100
  lock
  serial
  version
```

Listing 3.1: Job Queue structure

In Listing 3.2, we present the internal structure of a randomly-selected job in Ganeti, with its most basic fields:

```
{id : 17238,
 ops : [{'end_timestamp' : [1383218135, 558098],
         'exec_timestamp' : [1383218073, 542380],
         'input' : {'OP_ID': 'OP_INSTANCE_CREATE',
                    'depends': 'null',
                    'dry_run': False,
                    'hypervisor': 'kvm',
                    'iallocator': 'hail',
                    'opportunistic_locking': False,
                           [...]
                    'priority': 0},
         'log' : [[1,
                   [1383218074, 501763],
                   'message', '- INFO: Selected ... '],
                  [...]],
         'priority' : 0,
         'start_timestamp' : [1383218073, 349666],
         'status' : 'success'}],
 received_timestamp : [1383218073, 199686],
 start_timestamp : [1383218073, 349666],
 end_timestamp : [1383218135, 558281]}
```

Listing 3.2: Job structure

As we notice from this listing, a job consists of the `ops` field, short for opcodes, the job `id`, and three `timestamp` fields that indicate when the job passed from the various statuses we presented earlier. This job, consists of an opcode list with a single element, and represents an instance create operation as indicated by the `OP_ID` field, i.e, `OP_INSTANCE_CREATE`. Every opcode also has its own timestamps, as the job, so that the user can be informed with more details about the exact time every single opcode passed from the various statuses. The rest fields presented, are related to the specific opcode, to make the reader better understand how Ganeti stores and handles the parameters we give in a commonly used job like the creation of an instance. These are the iallocator algorithm, the hypervisor chosen, and the opportunistic locking choice for the lock retrieval, a topic that we 'll cover later in Section 3.3.4.

Besides the above fields, we distinguish two important fields that will help us to better understand how a job executes by the Ganeti processor; the `priority` and `depends` fields.

The job `priority` is an integer number; the lower the number the higher the opcode's priority is. This is a very helpful attribute in job queue handling, because in cases we want to run a job like an emergency shutdown as soon as possible, we want to overcome factors that could delay us. The priority range is [-20..19], and jobs submitted without priority assigned the default zero value. To avoid starvation, a job can change its own priority after a certain amount of retries, or a certain amount of time. One interesting thing is that opcodes also have their own priorities. So, the job priority is the same as its first unprocessed opcode. This behavior, combined with the fact that the job processor returns the job back to the queue after each opcode completion, means that if there are opcodes of higher priority submitted in the meantime of a job execution, these will first try to acquire their locks and as result the job that was `Running` will go to the `Waiting` status again. That behavior makes the job queue structure a lot more versatile.

The `depends` field of a job, is an optional property which defines dependencies on other jobs. Clients can submit jobs in the right order and proceed to wait for changes made to them. The master daemon will take care of everything, Section 3.3.3 covers that topic. Jobs waiting for dependencies are in the `Waiting` status. In Listing 3.3, we present a simple example of job dependencies:

```
# First job
{
  id : 1365,
  ops : [
    { ''OP_ID'' : ''OP_INSTANCE_REPLACE_DISKS'',
      ''depends'' : null, ... },
    { ''OP_ID'' : ''OP_INSTANCE_START'',
      ''depends'' : null, ... },
  ],
}

# Second job
{
  id : 1532,
  ops : [
    { ''OP_ID'' : ''OP_INSTANCE_FAILOVER'',
```

```
      ''depends'' : null, ... },
  ],
}

# Third job, depending on success of previous jobs
{
  id : 1690,
  ops : [
    { ''OP_ID'' : ''OP_NODE_SET_PARAMS'',
      ''depends'' : [
        [ 1365, [''success'']]
        [ 1532, [''success'']]], ... }
  ],
}
```

Listing 3.3: Job dependency diagram

The job queue must be consistent between the master node and the master candidates, just like the cluster configuration files. Failures to replicate a job to other nodes will be only flagged as error in the master daemon log if more than half of nodes fail to copy it, otherwise the failure will be ignored and the operation will continue normally. This relies on the fact that the next update for already running jobs, will retry the update.

Now we will present the job execution procedure from a high-level view; the *"Life of a Ganeti job"*, from the time the user submits it till its completion:

1. Client submits the job. The appropriate opcode or a list of opcodes if the job consists of multiple opcodes, will be built in the client side. The opcode contains all the available information that Ganeti needs to execute the operation. For example, an *OpInstanceCreate* opcode contains the name of the instance, the os_type, the hypervisor, or instance-related parameters such as the beparams, the hvparams, the nicparams, and so on.

2. Then the list of the opcode{s} will be sent via the LUXI protocol in the master daemon, who will generate a new job identifier depending on the value of the serial file, and it will assign it to the job. Then the master daemon writes the job to his local disk and replicates it to the master candidates. The job must be copied successfully at half of the candidates at least, otherwise the operation will fail. Then, the identifier is returned to the client using the LUXI protocol again.

3. After the job_id is returned to the client, the master daemon builds the job object named `_QueuedJob`, and adds a new task to the workerpool. The task is the job object and the workerpool is a heap queue. The tasks are ordered in the heap queue in respect to the job's priority primarily, and if the priorities match, to an increasing number which denotes their incoming order.

4. As soon as a new task is added to the heap queue, a pool of job queue workers with currently 25 threads will be notified for new arrivals. Those threads wait for new jobs to arrive. If all threads are busy, the job will have to wait until one of them become available. The first worker finishing its work will grab it. Otherwise one of the waiting threads will pick up the new job.

5. When a job assigned to a worker it is time for the job to start its execution. The worker does not know nothing about the opcodes that the job contains. He just passes the opcode to the Ganeti's processor who dispatches them to the appropriate Logical Units, the *LUs* in short. There is a Logical Unit for each Ganeti opcode which knows how to deal with it. The LU is the part of Ganeti which finally executes the operation which will modify the cluster's state. The rest responsibilities of a worker thread include the appropriate handling of the job queue lock, the notification of other threads when it finishes its work, and generally taking care of the job's smooth execution.

6. If the user chooses to wait for job status updates and does not make use of the `-submit` flag, he waits by calling a waiting RPC function. The mechanism underlying the waiting function is an inotify manager who responds to events happen in the job file located to disk. In this case, log messages may be shown to the user depending on the job. The user can also cancel the job while it is waiting in the queue.

7. The client can also archive the job, which then moved to a history directory called *archive* ,i.e., the default path is the `/var/lib/ganeti/queue/archive` directory. This can be done in order to speed up the queue handling, because by default, jobs in the archive directory do not touched by any function.

3.3.3 Ganeti Daemons

We have already made a few references to the Ganeti daemons in previous sections. Now we will talk in more details about the internal structure of Ganeti, and particularly the set of daemons that it is divided into. Ganeti consists of a growing number of daemons. Each of these deals with a specific task that the cluster has to face, and communicates with the rest using a variety of protocols. Specifically, as of Ganeti version 2.7, we have four daemons. The situation is as follows:

Master Daemon
The master daemon runs on a single node only, the master node. Currently is written in python and deals with every cluster operation. It is the Ganeti's heart because it is responsible for the overall cluster coordination. Without it, no modification can be performed on the cluster. This is the reason why it is the most heavy loaded daemon of all. It receives the commands given by the clients, either through the Command Line tools or the Remote API, parses them, and executes the appropriate operation. Creates, and manages the jobs that will execute those commands, handles the locks, and ensures that race conditions will never occur. It is also responsible for managing, and maintaining the cluster configuration files, updating them when it is necessary, and replicating them to the master candidates, in addition to the job queue. Each job is managed by a separate python thread. The basic python threads which managed by the master daemon are presented below:

- *The main I/O thread*: It is a single thread. The `masterd` is build around this thread. It accepts connections in the master socket and setups/shutdowns the other thread pools.
- *The job queue worker threads*: This pool consists of 25 threads, each of which executes the jobs submitted by the clients. They are long-lived threads and are initialized during the daemon startup.

- *The client worker threads*: The client worker pool contains 16 threads. They handle the connections in the master socket, one thread per connected socket, parse LUXI requests, and send data back to the clients. They are also being built during daemon startup.
- *The RPC worker threads*: This is not actually a pool like the above two categories. The thread size depends on the RPC call; single-node or multi-node. They interact with nodes using HTTP based RPC calls.

The `masterd` keeps some interaction paths for the communication with the rest Ganeti daemons. More specific, the interaction between the Command Line tools which are located in the master node, and the RAPI daemon is done with a custom protocol called LUXI. LUXI is a UNIX-socket based protocol of JSON-encoded messages. The UNIX socket permissions itself will determine the access rights. The LUXI API allows both job related operations, and cluster query functions.

The communication between the master daemon and the rest node daemons is done through RPC calls, using HTTP{S} simple PUT/GET of JSON-encoded messages. Communication between master and nodes is protected using SSL/TLS encryption. Both the clients and the server must have the cluster-wide shared SSL/TLS certificate, and verify it when establishing the connection by comparing fingerprints. For highly-traffic commands like image dumps, or low level commands such as restarting the `node-daemon`, a simple SSH protocol is used. The master node must share the cluster-wide shared SSH key with the rest nodes of the cluster.

During startup, the `masterd` will confirm in coordination with the node daemons that the node it is running, is the master node of the cluster, indeed. This is done via a voting procedure where all the nodes take part, even the offline ones. For successful confirmation the `masterd` has to get half plus one positive answers. When the `masterd` receives a SIGINT or SIGTERM signal, it stops accepting new jobs, and prepares to shut down as soon as the jobs that are currently running finish their execution. At the meanwhile, it still answers to LUXI requests. Pending jobs are re-added to the queue in `Queued` state after the daemon restarts. If a hard shutdown requested the cluster may be leaved in an inconsistent state.

The current Ganeti daemon structure suffers from many performance problems caused by the various protocols involved in interaction between daemons, and by the many python threads that are created which increase lock contention, log pollution and memory usage. This is the reason why from version 2.9, Ganeti daemon subdivision will change to improve the current situation.

Node Daemon

The `noded` runs on all the nodes of a cluster. It is also written in python, and it is responsible for receiving the requests made by the `masterd` over RPC, and executes them using the appropriate backend tool ,e.g., hypervisors, DRBD, LVM. It executes almost all operations that modify the node's state, like creating disks for instances, activating disks, starting/stopping an instance and so on. If a `noded` stops, the `masterd` will not be able to talk to this daemon, but the instances will not be affected.

Rapi Daemon

The `rapid` is written in python, and runs automatically on the master node only. By default, listens on TCP port 5080 and uses SSL/TLS encryption. Both those

parameters can change via command line. Ganeti supports a Remote API protocol which is JSON over HTTP, designed over the REST principle, for enabling communication with external clients, to easily retrieve information about the cluster state or modifying it. The `ganeti-rapid` waits for requests issued remotely through that protocol. Then, it forwards them via the LUXI protocol to the master daemon to deal with them.

`Rapid` reads its users and their rights from a file on startup, which is usually located under the `/var/lib/ganeti/rapi/users` path. Changes to that file will be loaded automatically. Most query operations are allowed without authentication. Modification operations though, require authentication in order to be executed.

Configuration Daemon

The configuration daemon is written in Haskell and runs on all master candidate nodes, since the configuration exists only on that group of nodes. This daemon is used to answer queries related to the configuration of a cluster. It makes sure that we have a highly-available and very fast way to query cluster configuration values. The `config.data` is reloaded automatically from disk every time it is updated. The requests are made through an HMAC authenticated JSON-encoded custom protocol over UDP, and meant to be used by parallel querying all the master candidates, or a subset of them, getting the most up to date answer by comparing the value of the `config.data`'s serial number, named `serial_no`. The queries are answered from a cached copy of the config which it keeps in memory, so no disk space is required in order to get an answer. Queries are also contain a "salt" which they expect the answers to be sent with, and clients are supposed to accept only answers which contain salt generated by them. The configuration daemon answers simple queries such as:

- master node
- master candidate, offline nodes
- instance list, primary nodes
- cluster info
- job list, and more

In Ganeti 2.7 we can also disable the `confd` during build time using the `-disable-confd` flag, if it is not needed in our setup. The `confd` serves both network-oriented queries about the static configuration, and local UNIX socket queries about the current status of the system including live data configuration. To answer queries of the second category the daemon has to communicate with the node daemons through RPC calls. In next Ganeti versions, it is intended those two functionalities to be separated into two different daemons, for simplicity and security reasons.

Finally, we have to mention that there exists a log file per daemon model, which are by default stored under `/var/log/ganeti` directory. Those log files are:

- The master-daemon.log, for the MasterD.
- The node-daemon.log, for the NodeD.
- The rapi-daemon.log, for the RapiD.
- The conf-daemon.log, for the ConfD.

3.3.4 Ganeti Locking

We have already covered the most major Ganeti parts. The last, but not least, part we will cover is the Ganeti *Locking* library and the way it is implemented. Locking libraries are vital for every project, affecting the overall performance. They must preserve data coherency, prevent deadlocks and thread/job starvation. Ganeti Locking library has passed through many stages but still improves and extends its features. In earlier Ganeti versions (*1.x*), there was a single global cluster lock for most operations, which made inevitable the execution of parallel operations. In Ganeti *v2.0* a complete redesign of the locking library has been made, which allowed the parallel execution of multiple operations. The locking library was also drastically improved in version *v2.1*, but the last major change was made in *v2.3* when the job priorities was firstly introduced. A feature called *Opportunistic Locking* was added lately, at *v2.7*, which also improved the parallel execution of some operations, mainly the instance creations. Below we will present the current Ganeti Locking library and how it is working *"under the hood"*.

The Locks

Locks are represented by objects of `locking.SharedLock` class. These locks are declared by the Logical Units located in the `cmdlib.py` module, and are acquired by the Processor which is found in the `mcpu.py` module, with the aid of the Ganeti Locking library, in `locking.py`. There are several locking levels which must acquired in specific order. These levels are the following:

1. Cluster level or BGL from Big Ganeti Lock.
2. Instance level.
3. Node allocation level or NAL.
4. Nodegroup level.
5. Node level.
6. Node resource level.
7. Network level.

These locks must be acquired in an increasing order. Each lock has the following possible statuses:

- *Unlocked*, anyone can grab the lock.
- *Shared*, anyone can grab the lock but in shared mode only.
- *Exclusive*, only one can hold the lock.

Besides the order in which the locks acquired, there are some extra rules which must be preserved:

- *Cluster level*, resides the Big Ganeti Lock, or BGL. It is the first lock which must be acquired before performing any operation in the cluster. Can be acquired either in shared or exclusive mode, but acquiring it in exclusive mode is discouraged and should be avoided.

- *Instance level*, resides the instance locks. They have the same name as the instances they protect, and are created when a new instance is added to the cluster. They are acquired as set, which means that if we need more than one instance locks we must acquire them at the same time. Internally the locking library acquire them in alphabetical order.
- *Node level*, resides the node locks and have the same names as the nodes they protect. They are also acquired as a set, and internally acquired in alphabetical order. We should first acquire all the instance level locks that reside in a node, before we acquire the node lock itself. Ofcourse, before the node locks, we should already have the BGL acquired, preferably in shared mode.
- *Node Resource level*, it is used for node resources protection, as it name reveals, and should be used by operations with possibly high impact on the node's disks.
- *Node Allocation level*, this lock is similar to the BGL in the sense that it has its own level and there is only one. It must be acquired after the instance locks and before the nodegroup locks, and used for instance allocation related operations. As a rule-of-thumb, NAL must be acquired in the same mode as the node and/or the node-resource locks. It blocks instance allocations for the whole cluster and can be acquired either in shared or exclusive mode. OpCodes doing instance allocations should acquire it in exclusive mode. When an Opcode blocks all or a significant amount of the cluster's locks, it should be acquired in shared mode. The NAL lock should be released when the set of acquired locks for an opcode reduces to the working set, to allow allocations to proceed.

Besides the above levels, we also have the `ConfigWriter` lock which is shared among those functions that read the `config.data` file, and acquired exclusively by functions that modifying it. This extra lock level allows the `config.data` replication to the master candidate nodes using the `rpc.call_upload_file` call, without holding the node level locks since the RPC function caller already holds the config lock in exclusive mode. This have the advantage that the config distribution can run in parallel with other cluster operations.

Similarly to the `ConfigWriter` lock, exists the Big *Job Queue* lock. It is used from all classes involved in the queue handling. Job queue functions acquiring it can be safely called from the rest of the code, because the lock is released before leaving the job queue again, something that prevents deadlocks. Unlocked queue functions must only be called from those functions, which have already acquired the lock beforehand.

Ganeti Locking Library

As we have already mentioned, locks in Ganeti are represented by objects. The basic class which implements a lock in Ganeti is the `SharedLock` class located in the `locking.py` module. All locks needed in the same level must be acquired together. So, a class is needed to take care of acquiring the locks always in the same order, thus preventing deadlocks. This class is the `locking.LockSet` class, a container of one or more `SharedLock` instances, which provides an interface to add/remove locks, to acquire, and subsequently release any number of those locks contained in it, distinguished by name. As this class is beyond the scope of this document, we will not present it further. In this section we will focus in the `SharedLock` class, to understand the Ganeti's approach to its locking requirements.

`SharedLock` class implements a shared lock. Multiple threads can acquire the lock by calling `acquire(shared=1)`. Exclusive acquirers should call `acquire(shared=0)`.

Since Ganeti first introduced job priorities in *v2.3*, the internal structure of `SharedLock` class also changed to support them. All pending acquires for a lock with different priorities is contained in a heap queue similar to the worker pool structure, named `__pending`. The heap queue does automatic sorting, automatically taking care of priorities. For each priority there is a single plain list ([]) of pending acquires. This is a normal in-order list of conditions [1] to be notified when the lock can be acquired. Shared acquires are grouped together by priority and the condition for them is stored in a separate dictionary of shared acquires called `__pending_shared`. There is also a dictionary called `__pending_by_prio` which keeps references for the per-priority queues indexed by priority for faster access.

When the lock is released, the code locates the list of pending acquires with the highest priority waiting. Due to the heap queue behavior, this is the first element in the structure. The first, zero indexed condition of the list is notified. Once all waiting threads receive the notification, the condition is removed from the list, the code processes the second condition and so on. When the list of conditions is empty it is removed from the list, and the list of conditions of the second priority in the heap is processed. In Listing 3.4, we present a possible state of the internal queue from a high-level view. Conditions are shown as waiting threads. Assuming we have no timeouts or other modifications, for simplicity reasons, the lock will be acquired by the threads in the following order (concurrent acquirers in parenthesis):

thread-Ex1, thread-Ex2, (thread-Sh1/thread-Sh2/thread-Sh3), (thread-Sh4/thread-Sh5), thread-Ex3, thread-Sh6, thread-Ex4, thread-Ex5

```
{
  (0,  [exc/thread-Ex1, exc/thread-Ex2,
        shr/thread-Sh1/thread-Sh2/thread-Sh3]),
  (2,  [shr/thread-Sh4/thread-Sh5]),
  (10, [exc/thread-Ex3]),
  (33, [shr/thread-Sh6, exc/thread-Ex4, exc/thread-Ex5]),
}
```

Listing 3.4: Structure of the `SharedLock` class

Locking Granularity

With the current locking policy, each Logical Unit acquires/releases the locks it needs; this means that locking is at the Logical Unit level. Ofcourse, each LU has its own locking requirements. Logical Units declare their locks and then execute their code with the appropriate locks held. In Listing 3.5, we present how the Ganeti Processor with the aid of the Logical Units executes an OpCode from an abstracted point of view, which pays more attention to the lock handling.

[1] A condition variable in Ganeti is a bit different from the Python's built-in `threading.Condition` class. It uses POSIX pipes in addition to the operating system support on timeouts on file descriptors (see `select(2)`). All clients of the condition use `select` or `poll` to wait for notifications. In a higher level-of-view a condition variable has `acquire()` and `release()` methods that call the associate lock methods. Also has a `wait()`, `notify()` and `notifyAll()` methods. Threads waiting for a particular change of state call `wait()` repeatedly until they see the desired state. Threads that modify the state will call `notify()` or `notifyAll()` when they change the state in a desired way for the waiting threads.

Opportunistic Locking

The last major change in Ganeti locking library was made in v2.7, when firstly introduced the *Opportunistic Locking* feature. The motivation behind this change was the need of more instance creations in a shorter amount of time. As of Ganeti *v2.6*, instance creations acquire all locks when an iallocator algorithm was used, causing a lot of lock congestion on node locks when someone tried to create many instances at once. This situation can become worse when we are waiting for DRBD synchronization between disks, if we choose the `drbd` template for an instance. As a result, even on big clusters with multiple nodegroups all instance creations were serialized. The main objective was to speed up instance creations in combination with an iallocator even when the cluster's balance is sacrificed in the process. The cluster can be rebalanced latterly, by using external Ganeti tools ,e.g., `hbal`. So, the opportunistic locking reduces the number of node locks acquired for instance creations, causing many creation operations to run in parallel. More specific, instead of forcibly acquiring all node locks for creating an instance using an iallocator, only those locks available will be acquired, and the iallocator algorithm will run on those nodes we have succeeded to acquire their locks.

```
1  def ExecOpCode(opcode):
2    # Depending on the opcode given get the appropriate LU class instance.
3    lu = lu_class(opcode)
4
5    # Purely lock-related functions.
6    # Update all the opcode parameters to their canonical form,
7    # (e.g. user passed names are expanded to the internal lock/resource
8    # name). Then known needed locks are declared.
9    lu.ExpandNames()
10
11   # While most of LUs declare their locks at ExpandNames time, sometimes
12   # there is the need to calculate some locks after having acquired the
13   # ones before, because we can't know which resources we need before
14   # locking the previous level.
15   lu.DeclareLocks()
16
17   ## At this point every function is called with the appropriate lock held.
18
19   # This method checks the prerequists for the execution of this LU.
20   lu.CheckPrereq()
21
22   # This method implements the actual work, the opcode execution.
23   lu.Exec()
24
25   ## All acquired locks released, locks declared for removal are removed.
```

Listing 3.5: OpCode execution path

Chapter 4

Ganeti and NoSQL

In Chapter 3, we discussed about Ganeti's architecture; we covered extensively its most basic components and all the prerequisites needed for someone to get familiar with this tool. In this chapter, we will discuss how Ganeti's current design impacts its performance and its scalability, and then we will provide a design solution for limiting some of those issues and make the tool even more suitable for cloud environments.

Specifically, section 4.1, discusses in a few words the current document's objective and how we are going to succeed it. Section 4.2, provides a detailed view of the configuration and job queue storage, in addition to the main issues resulting from those design choices. The options which lead us to the product choice are discussed in section 4.3, while section 4.4 covers in more details the tool we chose to overcome those performance issues, and more specific the Apache CouchDB database. Section 4.5 finally, provides a detailed presentation of our software design.

4.1 Objective

Ganeti has been evolved since first introduced, and has become a mature software tool for managing the low level VM of big clusters. In version 2.7, many new features like the opportunistic locking where introduced, but many scalability and performance issues are still arise from the current design.

In the current chapter, we will introduce a different approach for handling the Ganeti's configuration and job queue storage, using a NoSQL database as the backend storage layer, which will attempt to remedy some of the performance and scalability issues that exist in Ganeti version 2.7.

4.2 Background

While Ganeti v2.7 is usable, it limits the flexibility and the performance of the cluster. The current design for handling the configuration data and the job queue storage, in addition to how it replicates those files among the master candidate nodes, are some of the main reasons of those limitations. In the current section we will analyze this design, and we will extensively discuss about the main issues arisen from it.

More specifically, Section 4.2.1 presents analytically the configuration data form, in more details than the previous chapter, while section 4.2.2, concentrates on the job storage. Section 4.2.3, points out the most important performance issues that arise from the current design, both for the configuration file and the job queue storage.

4.2.1 Cluster configuration data

In section 3.3, we saw that the Ganeti's cluster configuration database is stored in a single file, the `config.data` file, on the master node filesystem. In this section, we will dive into the internal structure of the `config.data` file, which will lead us to the conjecture that the current configuration management is imperfect and suffers from scalability problems mainly on bigger clusters.

The configuration data uses JSON format, consisting of key/value pairs. The keys that the configuration file consists of, are a combination of Ganeti specific object collections, and default JSON objects of name/value pairs. In detail, there are five Ganeti objects namely: `Cluster`, `Node`, `Instance`, `Nodegroup`, and `Network`. From these objects the `cluster`, `nodes`, `instances`, `nodegroups`, and `networks` attributes are composed of. The default name/value pairs are the `serial_no`, `version`, `ctime`, and `mtime`. Ganeti configuration objects provide the appropriate functions for serializing, de-serializing, and handling them in a safe way, in order to by easily handled by external parties. They also provide recursive checks for their derived classes, and are also responsible for handling appropriately any attribute error that will arise.

The configuration file is represented internally by a `ConfigData` object, which actually is the topmost-level configuration object. The `cluster` attribute contains all the available information that the cluster needs to operate normally like the master node name, the master ip, the default hypervisor chosen during cluster initialization, the candidate_pool_size, and more. The `nodes` attribute, contains all the nodes that the cluster consists of. Each distinct `Node` object, contains information about the corresponding node such as its name, its primary and secondary IP, the node role information, and more. The `instances`, `nodegroups`, and `networks` attributes, contain relevant information for the instances that the cluster contains, the nodegroups which the user created, and the networks that exist in the cluster, as their names denote.

Besides the Ganeti specific objects, `config.data` also contains information about four general attributes. That are, the `serial_no`, which is an increasing number denoting the number that the `config.data` has been modified since it has been created, and it is used for consistency checks in case when some candidates are stalled in the middle of a configuration update, or in order to find the most recent answer when used by the configuration daemon ,i.e., `confd`. The `version` attribute, contains the current Ganeti version, and the `{c/m}time` timestamps contain the exact time when the cluster was created and modified, respectively.

Listing 4.1, briefly presents the `config.data` internal structure with its most basic key/value pairs. Due to its size, most of its attributes and values have been intentionally removed for simplicity reasons.

```
{cluster: {'candidate_pool_size': 3,
           'cluster_name': 'ganeti.cluster.com',
           'default_iallocator': 'hail',
                        . . .
           'hvparams': { ... },
           'master_ip': '192.168.0.206',
           'master_netdev': 'eth0',
           'master_node': 'node.example.com'}
 instances:
    {'inst1': {'admin_state': 'up',
               'ctime': 1383218076.653934,
               'disk_template': 'plain',
               'hvparams': {'vnc_bind_address': '0.0.0.0'},
               'hypervisor': 'kvm',
               'name': 'inst_name',
                           . . .
               'nics': [{'mac': 'aa:00:00:60:23:f9', 'nicparams': {}}],
               'os': 'snf-image+default',
               'osparams': {'img_format': 'diskdump',
                            'img_id': 'debian_base-7.0-1-x86_64',
                             . . .
                            }
               'primary_node': 'node.example.com'}}
 nodegroups:
    {'default': {'alloc_policy': 'preferred',
                 'name': 'default',
                       . . .
                 'uuid': '1c580c22-8a14-4f78-8416-0e0a7eac827f'}},
 nodes: {'node.example.com':
    {'ctime': 1380633061.361784,
     'drained': False,
     'group': '1c580c22-8a14-4f78-8416-0e0a7eac827f',
     'master_candidate': True,
     'master_capable': True,
                 . . .
     'name': 'node.example.com',
     'offline': False,
     'primary_ip': '192.168.0.3',
     'secondary_ip': '192.168.8.8',
     'vm_capable': True},
 networks : {},
 ctime: 1380633038.861877,
 mtime: 1383320564.876588,
 serial_no: 10,
 version: 2070000}
```

Listing 4.1: Structure of the `config.data` file

4.2.2 Job storage

Jobs are stored in the filesystem as separate, individual files using JSON format just like the `config.data` file. The choice of storing each job in its own separate file was made for a number of reasons. The most important of them are summarized below.

In chapter 3, we saw that a job can change its status many times during its execution ,e.g., Queued, Waiting, Running, and more. Moreover, an opcode, so as the job, passes from several execution phases like acquiring the needed locks, running the iallocator algorithm, and so on. The user should have access to all that information when requests it, either for debugging reasons, or simply for monitoring the execution path of a job. Those information is saved in the job object as a list of log messages in the `log` attribute. It is obvious that a job is modified many times during its execution. The choice of using a file per job, is based on the fact that a job changes quite often, and a file that can easily and atomically be (over)written facilitates this behavior. Furthermore, a file can be easily replicated to the master candidate nodes. The replication is done atomically for every single file with a multi-node `Remote Procedure Call (RPC)` call with a timeout of about a minute. In addition, a consistency check in the job queue across master candidate nodes through a third partie, can very easily be implemented, since all job files should be identical.

It is also interesting to see how a job is internally represented by Ganeti. Jobs are stored in the filesystem, but until the job completes its execution there is also an in-memory representation of it. Any modification to the job object is flushed on the disk, and then replicated to the candidate nodes. That in-memory representation of a job is a python class definition called `_QueuedJob`. This class contains attributes and implements all the appropriate methods needed for the smooth execution of a job. The most important attributes that a `_QueuedJob` object contains are: the job `id` field, the `queue` object on which the job belongs, the `ops` field which is a list of the job's opcodes where the `_QueuedOpCodes` objects are encapsulated, and three timestamps the `received`, `start`, and `end`, providing information about the time when a job was received, started and finished its execution respectively.

4.2.3 Caveats

We extensively discussed about the design of Ganeti. While it scales quite well for small clusters, it does not as the number of nodes grow, and some drawbacks start to appear. This is mainly due to the fact that several documents are shared among the nodes of the cluster. The `config.data` file and the job queue need to be present, and replicated, to all master candidates for reliability reasons mainly, and the `ssconf_` static configuration files have to be replicated to all the nodes of the cluster, as well. In addition, the `confd` which is present on all the master candidate nodes, might need to have access to the configuration data file even if the master node is down. Due to those needs, an increase in the size of the cluster will bring up the configuration management imperfections that Ganeti suffers of. In particular:

- Some operations like `instance-(add/remove/rename)`, `network-(add/remove)`, or `node-(add/remove/offline/drained)` are the most commonly used operations in a Ganeti cluster. Some of these operations like the instance related ones, grow in frequency as the cluster grows. We would ideally want the time needed in order

to those operations to complete, be steady. This does not apply at all, due to the fact that all these operations need to contact all the nodes of the cluster in order to update the `ssconf_` files, so they become significantly slower.

- Any other operations that do not affect the `ssconf_` files will contact, at least, all the master candidate nodes for two reasons. The first one is to inform the configuration data files for the new updates been made, and the second one is to synchronize the job queues among them. As the number of nodes in a cluster grows, we expect the number of jobs to grow as well. While the number of candidate nodes is constant, an increase on the jobs will have impact on the cluster performance. This is due to the fact of an overloaded master daemon which will affect the whole cluster performance, because besides the growth in the number of jobs it has to deal with, it also has to supervise the replication process of those files among the candidate nodes, manage the locking, an so on.

- The candidate pool size is not affected as the cluster size increases. It is a constant number independent of the nodes of the cluster. Ganeti though, interacts with the candidate nodes every time it updates one of its configuration and job queue files in order to maintain the file consistency among them. The replication procedure is the main reason that prevents Ganeti from scaling. It is handled and monitored exclusively by the master daemon, and is a procedure that breaks the scalability because many should-be-fast operations are slowed down by replicating the changes to remote nodes, thus waiting with locks held on remote RPC calls to complete.

- Another issue arisen from the increase in the number of jobs, is the *config lock*. Any job that modifies the cluster state must exclusively acquire that lock before apply its changes to the cluster, and so as to the configuration file. The single config lock becomes a bottleneck, when a huge number of jobs is in execution and try to acquire it. In addition, the lock will not be released until the modification have been successfully replicated to the master candidate nodes, something that increases the congestion on that lock.

- The configuration file is a JSON formatted file which needs to be serialized before it is flushed to disk. The time needed to serialize it, is quite small in smaller clusters and it can be ignored. Ganeti though, is a tool used by big open source projects, such as Synnefo [1], which means it is mainly used in bigger clusters. Providing an example, a cluster with around *1.000* instances, will have a `config.data` file of about *2.5 MB* in size. The serialization cost starts to be a bottleneck when the configuration file enlarges, even from sizes at around *1 MB*. This claim will be justified in Chapter 5. The de-serialization cost from disk is also raised respectively, but it does not affect the overall cluster performance at all, because the `config.data` de-serializes only at the master-daemon startup. Then it exists at the master node memory, and its disk version is updated from there.

- Besides the serialization cost, more factors are affected when the configuration data size increases. These are the time needed to flush the changes to disk, in addition to the time needed to distribute those modifications to the master candidates. These two extra costs, in addition to the serialization time discussed in the bullet above, slows down the cluster operations, reduce the overall performance of the cluster and

[1] `http://www.synnefo.org`

make even the quick commands last several minutes, instead of some seconds in order to complete.

- The current configuration management groups all the cluster's attributes in a single JSON file ,e.g., instances, nodes, networks, and more. This design choice, in combination to the global config lock, forbids any type of concurrent update to distinct Ganeti objects, and makes every modification access to the configuration file be serialized. There are various cases when that restriction reduces the cluster performance. An example is when a client wants to add an instance to the cluster, while another one tries to create a new network. While the clients modify two distinct objects of the cluster with an indirect relationship, they can not make the updates concurrently, but they have to wait instead. It would be very convenient for Ganeti and its users to allow concurrent updates when they do not affect other cluster operations, because it would have a positive impact in the cluster's throughput.

- Moreover, in case of a faulty update on the configuration file, there is no way to roll back the changes made in it and return to a previous state. Even if we keep backups of the configuration file and want to revert a modification, we should alter the whole configuration file and not only the section that we modified, increasing the probability of causing a breakage in the cluster configuration state. This attribute could be very useful in case when the `config.data` breaks due to a faulty-update on it, and we want to recover it painless.

4.3 Choice of product

Our design solution aims to address some, and not all, of the above issues. It replaces the Ganeti configuration and job queue storage with a NoSQL distributed, document-oriented database. There are plenty of NoSQL choices in the database market which would meet Ganeti's requirements, such as MongoDB [2], or Riak [3]. Some of the criteria for deciding which one was the best fitted NoSQL solution for Ganeti were the replication style, the reliability provided, the available API, and how the database handles reads and writes. For our implementation we chose *Apache CouchDB* [4] as our backend database.

Apache CouchDB (acronym of *"Cluster Of Unreliable Commodity Hardware"*), is an open source database built on the Erlang OTP platform [5], a functional, concurrent programming language, and a development platform too. Erlang was developed for real-time telecom applications with an extreme emphasis on reliability and availability using lightweight "processes" and message passing for concurrency. It is an ideal solution for a database server due to its robustness and concurrent nature. CouchDB is a database that focuses on being *"a database that completely embraces the web"*, as its developers promote. It is a NoSQL document-oriented database, using JSON format for storing its data, a simple RESTful API based on HTTP, and a powerful query server using Map/Reduce techniques that is written in JavaScript, by default, but there are also servers available for nearly any language we can imagine [6]. It was created in April 2005 by Damien Katz, and since

[2] `http://www.mongodb.org/`
[3] `http://basho.com/riak/`
[4] `http://couchdb.apache.org/`
[5] `http://www.erlang.org`
[6] `http://en.wikipedia.org/wiki/CouchDB`

February 2008 is part of the Apache foundation. In July 2013, the CouchDB community merged the codebase for BigCouch, Cloudant's clustered version of CouchDB, into the Apache project. The BigCouch clustering framework is prepared to be included in an upcoming release of Apache CouchDB [17].

The main reasons we chose CouchDB for our modifications to Ganeti, are presented in the current section. We do not claim it is the best fitted database solution for Ganeti or the only one, but without lost of generality it is a representative NoSQL database which will show us how Ganeti reacts with a different underlying storage solution. The criteria which were evaluated are the following:

- *Free Software,* CouchDB is an open source project which keeps up with the current open source Ganeti policy.
- *Licensing,* CouchDB is licensed under the Apache License, Version 2.0 [7] which does not affect Ganeti's license or model. Ganeti will remain a separate software, which connects through Apache licensed libraries to CouchDB.
- *Bindings,* Ganeti is written in both python and Haskell languages. CouchDB provides Haskell bindings which are available on Hackage, the Haskell package library, and a variety of Python libraries for working with CouchDB.
- *Document Storage,* In order to avoid big changes in the current configuration and job queue storage we want a solution that would fit this design. CouchDB stores its data as documents; one or more key/value pairs using JSON format, that fits the Ganeti's needs.
- *Replication model,* CouchDB is a peer-based distributed database system. Its replication process synchronizes two copies of the same database. If you change one document of a database, the replication process will propagate these changes to the second database. This is very similar to the master-slave replication procedure used by Ganeti. In addition, the BigCouch merge in the CouchDB project will give us a native clustering support which could later provide different design solutions for the Ganeti data handling.
- *Simplicity,* CouchDB comes with good documentation in the form of books, presentations, blog posts, wikis, and a strong and active community which makes it much simple to be installed and configured. In addition, the RESTful HTTP API which uses, is quite straightforward and simple, and it does not requires much effort to learn using it.
- *Security model,* CouchDB comes with a simple reader access and update validation model to protect who can read and update documents, that can also be extended to support custom security models. In addition, every CouchDB database instance can have one or more administrator accounts. These accounts come with specific privileges and user credentials in order to secure the access to selected databases and documents. Validation functions are written in JavaScript, and can be used as documents are written to disk. If the documents pass the validation criteria, the update is allowed to continue, otherwise the update is aborted and an error message is returned to the client.

[7] `http://www.apache.org/licenses/LICENSE-2.0`

- *Resource Usage,* CouchDB is designed from the ground up to service highly-concurrent use cases. There is not fixed RAM, CPU or disk space needs for CouchDB. It is flexible enough to run from a smart phone to a cluster. Apparently, more RAM is better, because CouchDB works completely through file I/O, delegating caching to the operating system, the filesystem cache. CouchDB makes the assumption that disk space is cheap, so it does not take great care of it. The good news are that some operations like database *compaction* reclaim a lot of disk space. CouchDB is written in Erlang, so the more CPU in a server, the most *beam* processes [8] can be created. CouchDB (or Erlang) take great advantage of this resource. Summing up, memory and disk have the great "pain", as a result faster disks and more memory will be handy and will increase the database overall performance.

- *Debugging,* CouchDB maintains a log file where every single operation, or event made to the CouchDB server is recorded. The debugging level of the log file can be modified, and the user can define how verbose and detailed the logging will be. We can choose from a very informative log file, where every HTTP header, external process communications, authorization operations, and much more information is recorded, to a status where any debug message is disabled. Ganeti also maintains a set of log files to record its updates, and CouchDB will just insert one more log file to the already existent.

- *Reliability,* CouchDB comes with a fault-tolerant storage engine that puts the safety of the data first. As it name denotes, is build for Clustering On Unreliable Commodity Hardware and the main goal is to provide data integrity, high-scalability and reliability in a fault-prone environment. The fact that CouchDB is written in Erlang, a concurrent, functional programming language with an emphasis on fault-tolerance reinforces the succeed of the data safety goal. Its internal structure is fault-tolerant, and failures occur in a controlled environment and are dealt without letting single problems cascade to the whole system, but are isolated in single requests. For CouchDB specifically, if an operation fails, we will never end up in a state with partially updated objects, or corrupted objects that was previously written successfully to the server. It provides a total reliable storage engine, which we will extensively present in section 4.4.

- *Recovery from failures,* The current CouchDB design does not provide automate recovery from failures. The recovery procedure will be handled by Ganeti with the administrator help, as it currently is. Various commands like the `master-failover`, or the `redist-conf` will be extended to meet up with the new needs. The CouchDB replication procedure, along with the CouchDB tools, will make those operations relatively painless.

- *Backups,* CouchDB stores each database in a separate file in the disk, as we are extensively cover in later sections. We can take backups of a database file, silently and without stopping the database by simply running a `cp` Unix command ,i.e., `cp db.couch /mnt/backup`. We can store periodically flat-file copies of the database files on the master and master candidate nodes, and create a method to "re-init" the Ganeti status from those flat-files during disaster-recovery.

- *Backwards compatibility,* We would like to have a way to easily converting from the current Ganeti storage management, to CouchDB. We decided to support both

[8] `http://www.erlang.org/documentation/doc-5.8.3/doc/efficiency_guide/processes.html`

CouchDB and file configuration as different storage engines, with different limitations for each case. Maybe it is not the simplest solution since it requires to convert a fairly great amount of code, but the future ability of further expanding the underlying storage options is a tradeoff that reinforces that approach.

4.4 Apache CouchDB

In the previous section, we mentioned the factors why we chose CouchDB for our implementation, depending on current Ganeti needs, keeping also in mind that we do not want a choice that will introduce important Ganeti design changes. In the current section we will discuss extensively about Apache CouchDB and its main characteristics and features, in order to proceed with the detailed design section.

CouchDB is a document-oriented, distributed, schema-free NoSQL database, using views for aggregating and reporting on documents in a database. For a complete overview of CouchDB's technical information there is a well structured documentation [9]. Let's review some of the basic elements of CouchDB:

Schema-Free
Unlike SQL databases which are designed to support highly-structured data, with CouchDB no schema is required. New document types can be added at will, alongside with the old ones. CouchDB is designed to perform on document-oriented applications with large amounts of semi-structured data.

Document-Oriented
Documents are the primary unit of data in CouchDB. Each document is a JSON formatted object and consists of any number of named fields and attachments [10]. Field values can be strings, numbers, dates, or even ordered lists and associative arrays. Documents also include metadata that are maintained and used by the CouchDB server. An example of a document in CouchDB would be a contact document as shown in Listing 4.2. In that document, "Type" is a field that contains a single string value "Contact", "Email" is a field containing a list of two values and so on.

A CouchDB database is a flat collection of documents, each having a unique identifier named `_id`, and a revision/version number named `_rev`. The version number is a special field in CouchDB with great importance, about which we will talk later in this chapter. The underlying data structure used to store the database files is a B+tree structure. CouchDB implementation is a bit different from the original B+trees; while it maintains all the important properties, it adds an append-only design [11], along with a *Multi-Version Concurrency Control*. That append-only variation of the original B+tree structure, trades a bit of (disk) space for speed. A B+tree is an excellent data structure for storing huge amount of data for fast retrieval. The B+trees are very shallow but wide data structures. The leaf nodes contain the actual data in an ordered manner, while the intermediate nodes contain indexes/pointers to

[9] `http://kxepal.iriscouch.com/docs/dev/contents.html`
[10] Documents in CouchDB can have attachments just like an email. For creating an attachment, we need to provide a file name, the MIME type and the base64 encoded binary data. Its even possible to have multiple attachments for a single document.
[11] `http://guide.couchdb.org/editions/1/en/btree.html`

the nodes beneath them. While other tree structures can grow very high, a typical B+tree has a single-digit height, even on millions of entries. *Jan Lenhard*, one of the core CouchDB developers, said during the *Berlin's Buzzwords 2013* conference [12] that a B+tree node in CouchDB has a size of about *60.000* entries. That actually means that even on billions of entries in a database, we will have a tree depth of 6, at most. This is very interesting for CouchDB particularly, where the leaves nodes of the B+tree are stored in a slow medium such as a hard drive. CouchDB does not make use of a built-in cache layer, but it uses the operating system's cache instead. Due to the small height of the structure, the filesystem cache keeps the upper nodes of the tree cached, so reading, or writing to a document requiring only a few seeks to disk on the final tree node, making it a quite fast data structure for both read and write requests.

```
{
    _id : ''couch_doc'',
    _rev : ''4-33d3d5926a1986207dde6e9adc4c9006'',
    Type : ''Contact'',
    Name : ''Name'',
    Email : [{''Home'' : ''foo@bar.net'', ''Work'' : 'foo@bar-work.net'}],
    Phone : [{''Home'' : ''+81 00 0000 0000''}],
    Tags : [''foo'', ''bar'', ''foo-bar'']
    Address : []
}
```

Listing 4.2: Document sample in CouchDB

Views

CouchDB is a schema-less database. However, for some applications a kind of structured data may be required. In order to deal with that problem, CouchDB integrates a view model using JavaScript for the view description. Views are also a useful tool for many other purposes like document filtering based on specific fields, extracting and presenting data in a special order, building indexes among documents to find a value that resides in them, and generally performing all sorts of calculations on the databases data.

CouchDB views are stored inside special *design documents*, and a design document can contain any number of views. A view is actually a map function of a map/reduce system. All map functions have a single parameter `doc`, which corresponds to a single document in the database. A simple view example which checks whether the database's documents have a date and a title field is shown in Listing 4.3. When we query the view, CouchDB takes the source code and runs it on every document in the database our view was defined. We query the view to retrieve the view result. Because the view runs on all documents of a database, it would take a lot amount of time to run it, if it should traverse the whole database every time we query it. Instead, a view runs on all documents only the first time it is queried; if a document is updated the map function will only run to recompute the new keys and values

[12] Berlin Buzzwords is a Germany's conference that focuses on the issues of scalable search, data-analysis in the cloud and NoSQL-databases

for the updated document. Views in CouchDB are stored in separate flat files just like the databases, using B+tree data structure as well. Initially, the view file is empty because no index has been built yet. Views are being built *lazily* when the first query is made. The next view query will incrementally update the not updated view indexes.

```
function(doc) {
  if(doc.date && doc.title) {
    emit(doc.date, doc.title);
  }
}
```

Listing 4.3: View function in Javascript in CouchDB

ACID properties

CouchDB is a database, so every transaction should ensure the ACID (Atomicity, Consistency, Isolation, Durability) properties. CouchDB provides ACID semantics, and in this part we will examine carefully each of those properties.

- *Atomicity*, refers to the ability of database to guarantee that either all the tasks of a transaction are performed or none of them are. Each transaction is said to be atomic in case when one part of a transaction fails, the whole transaction fails. CouchDB database modifications follow the "all or nothing" rule, ensuring that property.
- *Consistency*, is the property that ensures that any transaction will bring the database from one valid state to another, according to some defined rules. The valid state does not necessarily guarantee correctness of the transaction in all ways the application programmer might have wanted, but that the database will remain consistent even if the transaction succeeds, or fails. For distributed systems, as CouchDB is, the system is either strongly consistent or has some form of weak consistency, also referred as eventual consistency. CouchDB is an eventual consistent database, ensuring that the database will eventually reach at a consistent state. The *MVCC* method ensures that each client sees a consistent snapshot of the database from the beginning to the end of the read operation. The latest version is sitting somewhere in the cluster. Older versions are still out there and eventually all nodes will see the latest version.
- *Isolation*, refers to the requirement that other operations cannot access or see the data in an intermediate state during a transaction. This constraint is required to maintain the performance as well as the consistency between transactions in a database. Thus, each transaction is unaware of another transactions executed concurrently in the system. It ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially. Documents updates in CouchDB (create, delete, modify) are always serialized on disk. In addition, the concurrent update of the same document will result in two new documents, where none of the clients that modify the same document is aware of the other client existence.
- *Durability*, refers to the guarantee that once the user has been notified of success, the transaction will persist, and will not be undone. This means it will

survive system failure, and that the database system has checked the integrity constraints and will not need to abort the transaction. CouchDB uses by default the *fsync* system call, to ensure that a transaction have reached the disk before declaring it as successful. It also gives the ability to the user to loose that property and increase the write performance by using intermediate buffers, but by default it always "fsyncs" for every transaction. In addition, another transaction will never overwrite any changes made by a previously successful transaction, due to the append-only model followed. So, it cannot corrupt anything that has been written and committed to disk already.

In order to understand even better how CouchDB ensures the above properties, we are going to give an overview of some more technical, but important, CouchDB features:

CouchDB implements a form of *Multi-Version Concurrency Control (MVCC)* [13], instead of locks. Requests are run in parallel, making excellent use of the CPU, but writes are always serialized on disk. Database readers will never have to wait for writers or other readers on the same document. Each reader sees a consistent snapshot of the database from the beginning till the end of the operation. We mentioned earlier that every document always contains besides the `_id` field, a `_rev` field as well. This is the document's version, because in CouchDB documents are versioned. If we want to modify a document, we create an entire new version of it and save it after the old one. So, we end up with more than one versions of the same document, depending on how many times we modify it. The version is a composition of two values ,i.e., `''23-a2a33fdabad1376f58a12ea0ff4b''`. The first one is an increasing sequential number, which denotes how many times the file was modified. The second part of the number after the dash, is a hash composition of the document's contents plus the sequential number of the revision. If we found two documents, in different databases, with same `_id` and `_rev` values its not necessary to look at the contents of each file to know that are identical, we already know they are. In that lockless update model we may end up with two clients updating the same document. In that case, a conflict error will be produced, where two versions of the same document will exist. There is a conflict mechanism used by CouchDB to resolve those errors, and the user can also involve in the conflict resolution. There are three possible states after a conflict detection, which are explained in the `Eventual Consistency` section 4.4, and ensure that the database remains consistent even if sometimes the application should involve in that procedure.

Committing is the process of updating the database file to reflect the changes requested. It is a CouchDB process with great importance. It is not needed in order to use CouchDB, but it will help us to deeper understand the CouchDB design, and how the ACID properties are preserved. CouchDB uses a B+tree data structure for both the "storage" and the "view" needs. In that paragraph we will focus on the storage part. Every database file consists internally of a number of components. The most important of them follow:

- *B+tree, by_id_index.* It is a B+tree structure that uses the document ID as the index key. This index stores the mapping from document IDs to their positions on disk. Is is mainly used to lookup documents by their ID. The data on disk

[13] `http://en.wikipedia.org/wiki/Multiversion_concurrency_control`

it points to, contain the list of revisions, along with the document's revision history.

- *B+tree, by_seqnum_index.* It is a B+tree structure that uses the sequence number as the index key. A new sequence number is a monotonically increasing number, and is generated every time a document is created, deleted, or modified. This index stores the mapping from the update sequence number to the document's position on disk. This B+tree actually answers the question *"what happened since?"*, and is very useful to keep track of the last point of the replication synchronization or the last point of view index updates, by the *compaction* operation, and more.
- *Header.* The file header contains the pointer to the roots of the above two B+tree structures, some metadata needed by the CouchDB server like the database name, or size, and a checksum to ensure the file's integrity.

CouchDB database files are purely append-only. This means that all document updates like create, delete, and modify happen in a purely append-only mechanism. All writes occur at the end of the file, and old versions of documents are never overwritten, or deleted when new versions come in. Either inserting or deleting a document, the database file still grows only at the end. More specifically, CouchDB uses a kind of copy-on-modified approach. This means that the update is not happening in-place, but after we located the B+tree node that contains the document to be modified, we copy it over, make the appropriate modifications, and append it at the end of the file. After that, the parent B+tree nodes should also be informed to point to the new location. A modification to the parent node is triggered, which will also cause a new copy of the parent node, and so on all the way back to the root node of the B+tree. Finally, the file header must be modified to point to the new root node location. That means that every update will trigger 1 write to the document and *logN* writes to the B+tree nodes, where N is the B+tree height. A graphical representation of this procedure from a high-level of view is shown in Figure 4.1, which was taken from a blog post by Ricky Ho titled, "NoSQL patterns" [14]. Notice that the update of a data slot causes the creation of the *green* nodes, while the *yellow* ones will be removed as soon as no one uses them, after a compaction operation.

The update mechanism also forbids partial updates; either the update will succeed or will fail completely, there is nothing in between. The file footer is the last region that is updated and appended at the end of the file during a transaction. It is the last 4K of the database file, and is actually the database header we explained above. To declare an operation successful the footer must be written twice. It is separated in two identical chunks of 2K each. The committing process occurs in two phases. During the first phase, CouchDB appends any changes in terms of document data and their associated indexes to the file. After recorded the new file's length, it records it to the first 2K of the file footer. Those updates are synchronously flushed to disk. In the second phase it just copies the first 2K of the footer, over the second 2K of the footer, and flushes again. Only when both footers are flushed to disk CouchDB declares the transaction as successful. If a failure happens during phase 1, the partially flushed updates are simply forgotten. Incomplete writes and index updates simply appear as garbage data at the end of the file, and are ignored on database open. If a failure happens during the header committing (phase 2),

[14] `http://horicky.blogspot.gr/2009/11/nosql-patterns.html`

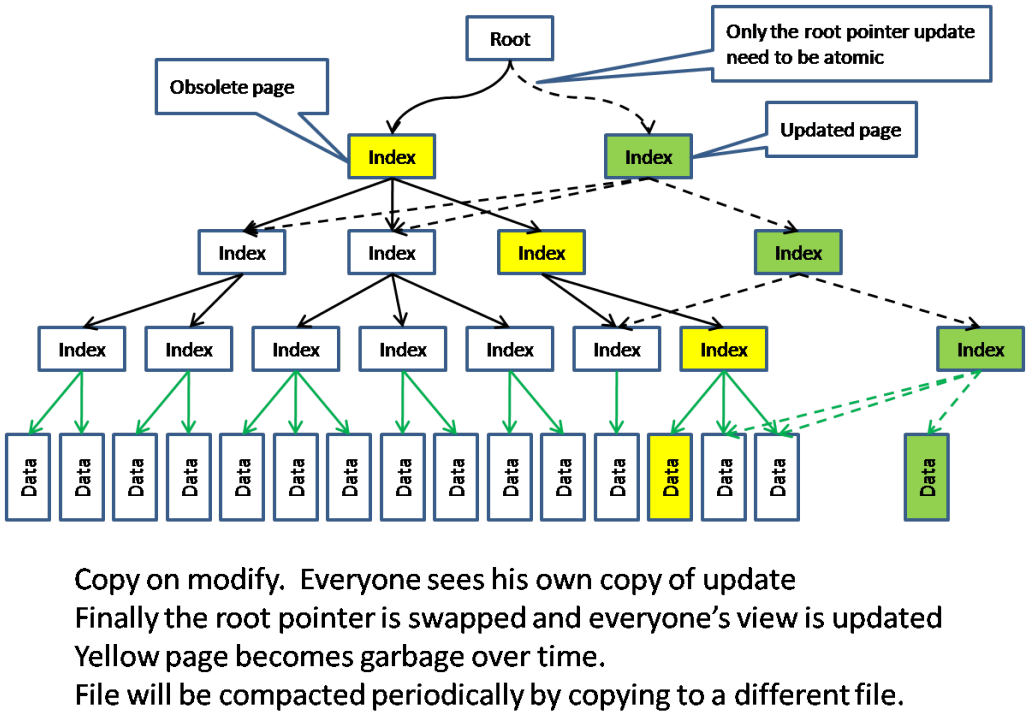


Figure 4.1: The power of B+trees in CouchDB

CouchDB start reading the file backwards [15]. If the first 2K is corrupted, CouchDB replaces it with the second identical 2K footer. The same happens if the second 2K are corrupted. If the header is intact then the data are intact as well. Checks never happen in the document data or their associated indexes, but only in the database header. With this design data are never lost, and data on disk are never corrupted.

This append-only design results in very fast updates, and query processing. Provided that the B+tree nodes are in system memory, they require the minimal seeks possible. A tradeoff of this scheme is the disk space needs. CouchDB is built with the assumption of cheap disk space, but since every document update causes a whole new copy of the B+tree indexes, care should be taken for limiting the disk space needs. CouchDB answer is the *compaction* operation. Compaction compresses the database file by removing unused sections created during updates. Old revisions of documents are also removed from the database, though a small amount of meta data is kept for server related needs. It is manually triggered per database and retrieves a great amount of disk space. Technically, the compaction process opens the file and reads the *by_seqnum_index*. It traces the B+tree all the way to the leaf node and copy the corresponding document content to the new file. The database remains completely online the entire time and all updates and reads are allowed to complete successfully. The old file is deleted only when all the data has been copied and all users transitioned to the new file.

[15] To differentiate data from headers, CouchDB appends a single byte every 4K of the database file. If the byte value is 0x01, everything preceding is database header. If the value is 0x00 then document data precedes.

Distributed Updates and Replication

Maybe the most powerful CouchDB's feature is the simplicity of replicating databases among different servers in the web. Replication is an incremental, fast, and one way process involving a *Source* and a *Target* database. The aim is very simple; synchronize the independent replica copies of the same database, between the source and the target nodes. All active documents should co-exist after the replication finished, and the deleted documents should also be deleted in both databases. During replication, only the last version of each document is copied from the source to target database, along with the document's revision history. Previous revisions are only accessible via the source database. Not all documents replicated over and over again, replication process continues from the last replicated documents. CouchDB replication protocol is not something magical, but a simple agreement of its public HTTP API in a specific way. The replicator is actually a separate, independent Erlang application with its own processes, where processes are CouchDB client workers with some logic on synchronizing documents between two databases. The CouchDB replication framework comes with many features and can be modified depending on the distributed model we want to follow. We can chose between *Master-Master* replication, the common *Master-Slave* replication, and also *Filtered* replication which managed by Javascript functions so that only particular documents fulfilling specific criteria will be replicated.

Before we extensively present the CouchDB Replication Protocol, we should explain some terms that we will confront at a later point. It is important to know that CouchDB saves every replication in a separate special database called `_replicator`. A replicator object is a normal JSON document just like all the other CouchDB documents. In Listing 4.4, we see a replication document where the replication is set to `continuous` which means that it will replicate as soon as new documents appear in the source database. The `create_target` attribute means that if the database does not exists in the target node it will be created, the `_id` is the identifier provided by user to the replication which is different form the `replication_id` which internally assigned by the CouchDB server. The `_replication_state` shows the current replication status, while the `_replication_state_time` is a Unix timestamp which denotes the time when replication was set. Finally the `source` and `target` are the databases which involve in the replication procedure. Now we are ready to proceed with the replication algorithm.

```
{
  _id : ''rep_from_A_to_B'',
  source :  ''http://aserver.com:5984/foo'',
  target :  ''http://bserver.com:5984/foo_a'',
  create_target : true,
  continuous :  true,
  _replication_id :  ''c0ebe9256695ff083347cbf95f93e280'',
  _replication_state :  ''triggered'',
  _replication_state_time :  1297971311
}
```

Listing 4.4: Replication document in CouchDB

CouchDB Replication Protocol

The CouchDB protocol is a synchronization protocol between two peers over HTTP using the RESTful CouchDB API. Anyone familiar with Git [16], a well-known distributed source control system, actually knows how replication works. Replicating is very similar to the *push* and *pull* methods used in distributed source managers like Git. Once a replication task is posted or a replication document is created in the `_replicator` database, the *couch_ replicator* process is started. This process watches the whole replication procedure which we are extensively explain below:

1. Firstly, *Replicator* (the process responsible for the replication) verifies that both the Source and Target peers/databases exist. This is done via a `HEAD /db` request in both databases. If the Target does not exist and the `create_target` field is set to True, an additional `PUT /<target>` request will be produced.
2. *Replicator*, retrieves basic information from both Source and Target, in order to get some important for the replication fields like the `update_seq`. Each update of a document during his life generates a serial sequence number and that Update Sequence gives us powerful information about the modifications made in a specific update, or a range of updates. This is done via a `GET /<db>` request.
3. Next, a unique identifier must be generated by the Source and assigned to the replication. That replication ID is very useful in order to find and resume previously interrupted replications, and identify each separate replication process.
4. The replication ID generated in the previous step, is saved in a special non-replicating document interface, the `_local` document on both peers via a `PUT /<db>/_local/<unique_id>`. This document will contain the last sequence ID, `last_seq` field, from the previously run replication. The `last_seq` is also mentioned as a *Checkpoint* for the replication process.
5. Using the replication ID, *Replicator* will retrieve the replication log history from both Source and Target via a `GET /<db>/_local/<unique_id>`. Then it will compare the two logs in order to find common ancestry. If there is not common ancestry, or if there are not any replication logs, it means that the replication is triggered for first time and an error will occur during that step. This will not affect replication because it is only an optimization to speedup the replication by not re-replicating already copied files.
6. *Replicator* then listens to the `_changes` feed from the Source's database. The `_changes` feed is actually a list of changes made to the database's documents. The *Checkpoint* can be used as input to the `since` option of that call, in order to retrieve changes immediately after the sequence number given. `feed, style, heartbeat,` and `filter` are some more parameters that can be used while listening to the changes feed. Only the list of current revisions changed will be returned, and not the whole revision tree of the changed documents. An example request for the `_changes` feed is:
   `GET /<source>/_changes?feed=normal&style=all_docs&since=last_seq`
7. Collect a group of documents/revision ID pairs from the changes feed and send them to the Target via a `POST /<target>/_revs_diff` request. The Target's response will contain only those pairs that are missing from its database.

[16] `http://gitscm.com/`

8. Fetch all the documents contained in the response from the previous step, from the Source database. A `GET /<source>/<doc_id>?revs= true&rev=<revision>` request will be made for each document. There are also some optimization options which can be used like `?atts_since`, but will not be further presented. Every document fetched is put in a local stack for bulk uploading in order to utilize the network's bandwidth effectively. When all documents are fetched they will be uploaded at once to the Target database using `POST /<target>/_bulk_docs` request.
9. After the batch of changes uploaded to the Target, the Target must ensure that every single bit is on persistent storage (on disk). The request is: `POST /<target>/_ensure_full_commit`.
10. Then, Source and Target update their replication logs and the *Checkpoint* value, so next replication will resume from that point. The request is as in a previous step: `POST /<db>/_local/<unique_id>`

If the replication feed is set to continuous, the *Replicator* will listen to the `_changes` feed until someone cancels it. When new changes encountered, the replication process will repeat from step 7. *Replicator* does not have to run either on Source or Target servers. It could run from anywhere with read access to the Source's database and write access to the Target's database. However, it is nearly always run by either the Source or the Target server.

Eventual Consistency

In Chapter 2, we discussed about the relational databases constraints and some compromises that a NoSQL database have to make, in order to achieve better performance and scalability, mainly in distributed environments. CouchDB loosens the consistency checks that a traditional database would make, and makes it really simple to build distributed applications with huge performance improvements that would scale, by sacrificing immediate consistency. Maintaining consistency between multiple database servers is a common and complex problem with many books devoted to its solution. Sharding, master/slave replication, multi-master replication, and many more techniques are used to deal with it.

CouchDB's way of address that problem is an incremental replication procedure along with a MVCC, which provides eventual consistency to the nodes involved. CouchDB operations stay within the context of a single document. Incremental replication is a process where document changes are periodically synchronized between different servers. The level the nodes will interact and communicate is decided by the user who sets up the replication. We can have from a cluster with independent nodes, up to a cluster where all nodes communicate and synchronize their databases. When two or more databases are synchronized from both directions, we may face a condition when the same document has been modified from both databases. This is a conflict, and CouchDB system comes with automatic conflict detection and resolution. What it does, is detect the document both changed, flag it as a conflicted document, and then decide which one version of the two is the "winning" one. The winning version saved as the most recent version of the two, and the losing version also stays in the database as a previously edited version. The two databases make exactly the same choice in order to achieve consistency between them. The user can decide between three possible options. Leave the CouchDB choice as it is, revert

it, or merge the two versions and generate a new one with the appropriate changes made.

4.5 Detailed Design

The changes we made, can be split into three main entities:

- *Core Changes*, that affect the software design.
- *Feature Changes*, which do not have impact on the design but facilitate some cluster wide operations.
- *Interface Changes*, user-level (*CLI*) and Remote-API level (*RAPI*) changes, which enable the features added.

4.5.1 Core Changes

One main core change will be extending Ganeti's design to support additional solutions for the configuration and the job queue storage. A use case will be Apache CouchDB along with the current disk implementation, an approach that aims to increase the job queue throughput and allow Ganeti to scale even better. Furthermore, the configuration data file will be divided into its most heavily used components, in order to increase the resource utilization in bigger clusters.

Besides these major changes, another "core" change which will not be visible to the users, will be abstracting a few python modules, and more specifically those responsible for the job queue and configuration storage management, in order to provide support for alternative storage solutions. This will allow future flexibility in defining additional drivers by moving away from the current static Ganeti approach which complicates and actually prohibits additional storage options.

Module Abstraction

The modules related to the configuration and job queue storage management that will be re-written in a more generic form, are the `config.py`, `jqueue.py`, and `jstore.py`. In Python terms, we will convert the `{config/jqueue}.py` modules to packages, while the `jqueue.py` module will remain a single entity, with the appropriate changes applied. A package is simply an extension of the module mechanism to a directory. A detailed overview of those modules and how they restructured follows:

- ***config.py***

Overview

The `config.py` module will be transformed into the `config/` package. The configuration related code will be split into smaller modules. The `config/__init__.py` file will contain the imports of the various sub-modules of the package, in order to expose their functionality to the clients that will make use of it. It is a necessity those

abstraction changes be invisible to the rest Ganeti code and to the user as well. Any new storage type that is created and added, should not disturb the existing code. The most reasonable step was to use polymorphism and to create a common interface for all those types, which would separate the rest Ganeti functionality from the knowledge of the specific type it uses. The solution we adopted was to force the creation of new objects occur through a common *Factory* [17], rather than to allow the creation code be spread all over Ganeti's code. With this approach, every new type designed is silently added to the factory and the new feature is available. The factory along with its auxiliary method and the appropriate lookup table for the configuration storage types are also contained in the `__init.py__` file. The relevant code is presented in Listing 4.5.

```
1  # Lookup table for configuration storage types
2  _CONFIG_DATA_TYPES = {
3    constants.CFG_DISK: default.DiskConfigWriter,
4    constants.CFG_COUCHDB: couch.CouchDBConfigWriter
5    }
6
7  def GetConfigWriterClass(name):
8    '''Returns the class for a configuration data storage type
9
10   @type name: string
11   @param name: Configuration storage type
12
13   '''
14   try:
15     return _CONFIG_DATA_TYPES[name]
16   except KeyError:
17     msg = 'Unknown configuration storage type: %r' % name
18     raise errors.ConfigurationError(msg)
19
20
21 def GetConfigWriter(name, **kargs):
22   '''Factory function for configuration storage methods.
23
24   @type name: string
25   @param name: Configuration storage type
26
27   '''
28   return GetConfigWriterClass(name)(**kargs)
```

Listing 4.5: Factory method for the configuration storage objects

The next task was to identify the features and functions that should be extracted to each sub-module of the package. The `config.py` file will be renamed to `config/base.py`. Only the should-be-common classes and methods for all the imple-

[17] `http://www.itmaybeahack.com/book/python-2.6/html/p03/p03c03_patterns.html#factory`

mented drivers will remain in that module. The rest class methods and functions that will not be contained in the base module, will be contained in a newly created file named `config/default.py`, which will provide the default disk storage type functionality for the configuration data file.

Technical Details

The `config/base.py` file, will provide the basic interface to the rest configuration drivers. The "globally" needed objects, from classes to functions and variables will appear in that module. The rest drivers will inherit [18] the base functionality, which they have to extend depending on their specific needs.

In the previously single configuration module, all the needed functionality were contained in a single class interface named `ConfigWriter`. This interface will be transformed to a more generic form, named `_BaseConfigWriter` that will implement only a subset of the original class methods. We can define a sort of rule about the functions that will be implemented by the base module, and those by each driver. We can group the configuration methods to those that modify the cluster state, and those that simply query it. The second group of methods will be implemented by the base module, while the first one will not. Providing an example, a method that modify the cluster state like the `AddInstance`, is presented in Listing 4.6. It is an empty method, that raises a `NotImplementedError` exception in order to indicate to the derived classes that it is required to be overridden. In addition, some method's definition were needed to be modified, in order to allow them to accept an arbitrary number of arguments depending on the driver's needs. The Python's `*args` and/or `**kargs` special syntax, facilitated those definition modifications.

```
1 def AddInstance(self, instance):
2   '''Add an instance to the configuration file.
3
4   '''
5   raise NotImplementedError()
```

Listing 4.6: Implementation of the `base.AddInstance` method

The default disk driver class methods will be implemented by the `config/default.py` module. We will not analytically present any of the functions of that module because they correspond to the default Ganeti methods. For a detailed comprehensive view of the code, refer to the following Github link [19]. Listing 4.7, presents a part of the disk constructor method of the `DiskConfigWriter` class, which shows how the base functionality is inherited via the Python `super()` call, before any disk specific changes been made.

[18] Because Python is a dynamically typed language, it does not really cares about interfaces and types. All it cares about is applying operations to objects. The interface and inheritance keywords have different meaning in Python from other Object-Oriented languages such as C++ or Java, where it is often to inherit from a common interface. In Python, and as in our implementation, the only reason to inherit is to re-use the code in the base class.

[19] `https://github.com/dblia/nosql-ganeti`

```
1 from ganeti.config import base
2
3 class DiskConfigWriter(base._BaseConfigWriter):
4   """Disk storage configuration type
5
6   """
7   def __init__(self, cfg_file=None, ... ):
8     super(DiskConfigWriter, self).__init__()
9
10    if cfg_file is None:
11      self._cfg_file = pathutils.CLUSTER_CONF_FILE
12    else:
13      self._cfg_file = cfg_file
14
15                . . .
16
17    self._OpenConfig()
```

Listing 4.7: Constructor of the `DiskConfigWriter` class

- ***jqueue.py***

Overview

The `jqueue.py` module that implements the job queue handling will also be re-written in more generic form. The new package will be named `jqueue/`, apparently. The `jqueue/__init.py__` file, will make all the appropriate package imports, and it will also contain the corresponding factory for the job queue storage needs. The factory is implemented in the same sense as the relevant one for the configuration package. The previously global job queue module will be renamed to the `jqueue/base.py` module, and the driver functionality for the default disk storage type will be implemented by the `jqueue/default.py` file, similarly to the configuration package.

Technical Details

The "rule" we defined for the configuration package about the methods that should remain in the base module and those that will not, similarly applies for that package. The `_JobFileChangesWaiter` class, will be re-written in a more generic way, and will be renamed to the `_BaseJobFileChangesWaiter` class. This class implements by default an *inotify* [20] manager using *Pyinotify*, a Python module for monitoring filesystem changes [21]. Different drivers will apply alternative methods for notifying about the events happen in a job queue object, so this class will be overridden by every driver. Similarly, the `_WaitForJobChangesHelper` class, which is a wrapper

[20] `http://en.wikipedia.org/wiki/Inotify`

[21] Pyinotify relies on a Linux Kernel feature (merged in kernel 2.6.13) called inotify. inotify is an event-driven notifier, its notifications are exported from kernel space to user space through three system calls. Pyinotify binds these system calls and provides an implementation on top of them offering a generic and abstract way to manipulate those functionalities. (*From Pyinotify Project's Wiki page*).

over the `_JobFileChangesWaiter`, will be abstracted and renamed to the `_BaseWaitForJobChangesHelper` class. The last class which must be transformed, is the `JobQueue` class, which is responsible for the job queue management. It will be renamed to `BaseJobQueue`, and it will only implement the common methods for the individual drivers.

The `jqueue/default.py` module, inherits and extends the base module functionality depending to the disk driver needs. The `_DiskJobFileChangesWaiter`, `_DiskWaitForJobChangesHelper`, and `DiskJobQueue`, are the corresponding implementations of the above-mentioned class transformations. This module will also contain the default `_JobChangesWaiter` class, which is needed by the file notification mechanism only. In Listing 4.8, we present a small part of the constructor method of the `DiskJobQueue` class.

```
1  from ganeti.jqueue import base
2
3  class DiskJobQueue(base.BaseJobQueue):
4    def __init__(self):
5      super(DiskJobQueue, self).__init__()
6
7      # Initialize the queue, and acquire the filelock.
8      self.jstore_cl = jstore.GetJStore(''disk'')
9      self._queue_filelock = self.jstore_cl.InitAndVerifyQueue()
10
11     # Read serial file
12             . . .
13
14     # More initializations
```

Listing 4.8: Constructor of the `DiskJobQueue` class

- ***jstore.py***

Overview

In Listing 4.8, and more specifically in line 8, we observe that an instance of the jstore class is created. This is due to the abstraction of the `jstore.py` module, which is the last one that must be transformed. Many auxiliary handling functions for the job queue are located in this module. In contrast to the previous modules, we did not convert this one to a separate package, mainly due to its small length.

Technical Details

The generic base jstore class will be renamed to `_Base`, where all the original methods like the `ReadSerial`, or the `ReadVersion`, will be contained. The `FileStorage` class overrides them in order to provide the disk type functionality. The appropriate factory method was also created and added to that module, so that the user can chose silently the desired jstore object.

Configuration Data

We already discussed about the main reasons why the current configuration data structure prevents Ganeti from achieving better resource utilization, particularly in bigger clusters. Ganeti is build for low-level VM management, so the most commonly used operations are those related to instance level modifications. Network and node related operations are also used quite often by cluster administrators. In bigger clusters where the single config file grows, a lot of congestion is observed and many should-be-fast operations, consume too much time in order to complete. The approach that is followed by the CouchDB driver, will try to remedy this limitation. In this section we will analytically discuss about the solution we designed, and how we applied it in the CouchDB driver.

Database Structure

We decided to separate the `config.data` file into its most heavily updated objects. Our claim was: *Why should we flush the whole configuration file to disk, when we just want to update a single field, like modifying an instance parameter.* By dividing it properly, we could save a lot of operational time for the cluster. The CouchDB approach of handling the documents in a database facilitates that thought, and in fact gave us the opportunity to apply it in the CouchDB driver. All the changes we will make, are referred to the on disk representation of the configuration data. The in-memory representation should, and will, remain as it is, for compatibility reasons with the rest of the code.

A database in CouchDB is a collection of documents, so we decided to separate the five primary components of the configuration file that are shown in Listing 4.1, into five distinct databases. As a result, we are introducing five new entities for the configuration storage; the `instances`, `nodes`, `nodegroups`, `networks`, and the `config_data` databases. With the new design, when we will want to add an instance, for example, we will add a single document in the `instances` database. This practically means that only few kilobytes will be serialized at a time instead of the whole configuration data file, which can become bigger than *2 MB* in bigger clusters. The detailed structure of the `config.data` file and a complete database presentation follows:

- *instances,* containing one document per instance object, and indexed (`_id`) by the instance name.
- *nodes,* containing one document per node object, and indexed by the node name.
- *nodegroups,* containing one document per nodegroup object, and indexed by the uuid of the group.
- *networks,* containing one document per network object, and indexed by the network name.
- *config_ data,* containing one document in total, indexed as `config.data`. It will contain all the Ganeti `cluster` information, as well as the `{c/m}time, serial_no,` and `version` fields. The `instances`, `nodes`, `nodegroups`, and `networks` fields will remain in this document for compatibility reasons, and will contain an empty dictionary as a value.

Each of the objects above, must be updated to contain two extra necessary fields for the CouchDB needs; the `_id` and `_rev` fields. Since CouchDB has access control

per database and even on document level, we will reserve the right to create private, and public databases, that will contain information which can be shared among the cluster administrators or third parties respectively.

Implementation Details

Before we proceed to the detailed design of the CouchDB driver, we have to mention some internal technical details of the driver from a high-level point of view. While the `config.data` object will remain in memory as a unified entity, like it currently is, the on-disk representation will be totally different, as we explained above. With this approach all the modifications we will make will be invisible to the rest Ganeti code, and none function definition will have to change.

Some aspects of that transformation must be clarified. The first one concerns the configuration loading from disk to memory. When the configuration loads from the CouchDB server must be unified, in order to construct the single Ganeti `objects.ConfigData` object, and achieve the compatibility we want to with the rest of the code. This is an easy task to do, and will introduce a quite small overhead during Ganeti reload, because Ganeti reads the configuration file from disk only when it starts, or reboots. The fact that Ganeti rarely reloads, makes that overhead negligible. The second aspect refers to the way that the updates will be flushed to the appropriate database. The previously globally used `_WriteConfig` function will be converted and extended to dispatch any updates to their appropriate databases. Finally, the last modification concerns the replication procedure, which will be handled exclusively by the CouchDB server, removing this heavy task completely from Ganeti.

While the CouchDB driver has been tested, and it seems to work smoothly, it brings an important change to the Ganeti configuration design. Since most of the Ganeti operations update more then one configuration object in a single call, we have to make more than one distinct updates in different databases. If we want to add an instance, for example, we also have to update the `serial_no`, and the `mtime` fields of the config object that are located in a separate database. Most of Ganeti operations that modify the state of the cluster follow that update policy. While the cost of making two sequential updates of small objects is negligible comparing to the whole flushing of the configuration file to disk, the problem lies in the fact that the ACID property of a configuration transaction changes, since we may face a situation of a hardware failure where only one of the two updates has been completed. In the first version of the CouchDB driver, we do not deal efficiently with that case but it is intended to be fixed in a later version. Currently, we firstly update the global configuration fields, that are the serial number and the modification time of the file, and later the main object of the operation, ,i.e., an instance, a node, a network, or a nodegroup. If the failure occurs between the two updates, we will have a configuration file with an increased serial number by one, and an updated modification time, while the actual update have not been performed, because the operation failed. We believe that this is a small tradeoff we have to bear with for the first version of our driver in order to achieve better resource utilization.

CouchDB Driver Design

Now we are ready to proceed with the detailed CouchDB driver design. The complete

code of the CouchDB configuration driver, as the rest code of the project is hosted at Github [22]. The top-level configuration management class, will be inherited and extended by the `CouchDBConfigWriter` class. Table 4.1, presents the interface of the cluster configuration on CouchDB driver behalf. A presentation of the most important methods of the driver follows:

CouchDBConfigWriter Class
`__init__(self, offline=False, accept_foreign=False)`
`IsCluster()`
`AllocatePort(self)`
`AddNodeGroup(self, group, ec_id, check_uuid=True)`
`_UnlockedAddNodeGroup(self, group, ec_id, check_uuid)`
`RemoveNodeGroup(self, group_uuid)`
`AddInstance(self, instance, ec_id)`
`_SetInstanceStatus(self, instance_name, status)`
`RemoveInstance(self, instance_name)`
`RenameInstance(self, old_name, new_name)`
`AddNode(self, node, ec_id)`
`RemoveNode(self, node_name)`
`MaintainCandidatePool(self, exceptions)`
`_UnlockedAddNodeToGroup(self, node_name, nodegroup_uuid)`
`_UnlockedRemoveNodeFromGroup(self, node)`
`AssignGroupNodes(self, mods)`
`_OpenConfig(self, accept_foreign)`
`_UpgradeConfig(self)`
`DistributeConfig(self, node, replicate)`
`_WriteConfig(self, db_name=None, data=None, feedback_fn=None)`
`SetVGName(self, vg_name)`
`SetDRBDHelper(self, drbd_helper)`
`Update(self, target, feedback_fn, ec_id=None)`
`AddNetwork(self, net, ec_id, check_uuid=True)`
`RemoveNetwork(self, network_uuid)`
`_BuildConfigData(self)`
`_ClusterObjectPrepare(config_data)`

Table 4.1: Interface of the `CouchDBConfigWriter` class

- `__init__()` method:

This is the constructor method of the CouchDB driver. The relevant code is presented in Listing 4.9. The code is quite straightforward; we create five new local class variables, one for each database we created, before we call the `_OpenConfig` method which will construct the global `ConfigData` object. These variables are instances of the `couchdb.client.Database` class, actually a representations of the databases on the CouchDB server.

[22] `https://github.com/dblia/nosql-ganeti`

```
class CouchDBConfigWriter(base._BaseConfigWriter):
  '''CouchDB configuration storage type

  '''
  def __init__(self, ... ):
    super(CouchDBConfigWriter, self).__init__()
                  . . .
    # CouchDB initialization
    # Setup the connection with Couch Server for all databases
    self._hostip = netutils.Hostname.GetIP(self._my_hostname)
    ip = self._hostip
    port = constants.DEFAULT_COUCHDB_PORT

    # Get database instances
    self._cfg_db = utils.GetDBInstance(CLUSTER_DB, ip, port)
    self._nodes_db = utils.GetDBInstance(NODES_DB, ip, port)
    self._networks_db = utils.GetDBInstance(NETWORKS_DB, ip, port)
    self._instances_db = utils.GetDBInstance(INSTANCES_DB, ip, port)
    self._nodegroups_db = utils.GetDBInstance(NODEGROUPS_DB, ip, port)

    self._OpenConfig()
```

Listing 4.9: Constructor of the `CouchDBConfigWriter` class

- `_OpenConfig` method:

The `_OpenConfig` method, is very similar to the default disk method. The main difference lies in the way that the document is loaded from disk. The default `utils.ReadFile` method, is replaced by the `_BuildConfigData` method, which is responsible for the unification of the configuration databases to a single object. Listing 4.10, contains the relevant code. It was a challenge to collect all the documents from all databases in a small amount of time. The view mechanism provided by CouchDB, gave us the solution. The special `_all_docs` view, combined with the `include_docs` attribute set to `True`, returns a listing of all the documents in a database, ordered by their `_id`. We query that view on each database and construct the separate dictionaries of the instances, nodes, networks, and nodegroups of the cluster. Then we combine them to build the unified `ConfigData` object.

```
def _BuildConfigData(self):
  '''This function unifies the config.data from its sub- components,
  because we do not want to change the in-memory represantation of it.

  '''
  # Get config.data from the db
  raw_data = self._cfg_db.get(''config.data'')

  # nodes dictionary
  nodes = {}
```

```
11  view_nodes = self._nodes_db.view('_all_docs', include_docs=True)
12  for row in view_nodes.rows:
13    node = row['doc']
14    nodes[node['name']] = node
15
16  # instances dictionary
17  instances = {}
18  view_insts = self._instances_db.view('_all_docs', include_docs=True)
19  for row in view_insts.rows:
20    instance = row['doc']
21    instances[instance['name']] = instance
22
23  # nodegroups dictionary
24        . . .
25
26  # networks dictionary
27        . . .
28
29  # build the unified config.data object
30  raw_data['nodegroups'] = nodegroups
31  raw_data['nodes'] = nodes
32  raw_data['instances'] = instances
33  raw_data['networks'] = networks
34
35  return raw_data
```

Listing 4.10: Implementation of the configuration unify method of CouchDB

- `_WriteConfig` method:

This method does not contain significant changes comparing to the default one, and it is unnecessary to provide its code. However, we have to mention two important differentiations from the default implementation. The `_DistributeConfig` call have been completely removed, because CouchDB follows a different approach for the document replication, and it was not necessary any longer. In addition, we do not have to make a `json.dump` call to serialize the `config.data` object before we flush it to disk. Instead, we call the `utils.WriteDocument` function which is responsible for all the write requests to the CouchDB server. We will present the relevant CouchDB `utils` module in the upcoming paragraphs.

- `AddNode` method:

Besides the primary methods for the configuration management that we presented earlier, we also chose to present the `AddNode` method. This is maybe the most representative method of the CouchDB driver, because it denotes the way we handle the writes, and how we setup the replication between the master candidate nodes. It is also a method of two object updates on different databases in a single operation. If we carefully take a look in Listing 4.11, we observe that if the node is a master candidate, we call the `_UnlockedReplicateSetup` function from the `utils/couch.py`

module. This call enables the replication between the master and the new candidate node, and we do not longer have to worry about it. It will replicate any new changes made to the master databases to the candidate nodes database respectively. The last modification concerns the `cluster` dictionary updates in the `config_data` database. As the configuration is in memory as a single `ConfigData` object, we surely do not want to flush the whole object to disk. Instead, we call the `_ClusterObjectPrepare` method which actually clears the `instances`, `nodes`, `nodegroups`, and `networks` dictionaries before flushing the updates of the `cluster` object to the `config_data` database.

```
1  @locking.ssynchronized(_config_lock)
2  def AddNode(self, node, ec_id):
3    '''Add a node to the configuration.
4
5    '''
6    # In-object updates
7    node._id = node.name
8    node.serial_no = 1
9    node.ctime = node.mtime = time.time()
10   self._config_data.nodes[node.name] = node
11   self._config_data.cluster.serial_no += 1
12
13   # Write the cluster object to the 'config_data' database.
14   data = _ClusterObjectPrepare(self._config_data)
15   self._WriteConfig(db_name=self._cfg_db, data=data)
16
17   # Write the node object to the 'nodes' database.
18   data = objects.Node.ToDict(node)
19   self._WriteConfig(db_name=self._nodes_db, data=data)
20
21   # Enable continuous replication if node marked as MC.
22   if node.master_candidate:
23     for db in constants.CONFIG_DATA_DBS:
24       utils.UnlockedReplicateSetup(hostip, node.primary_ip, db)
```

Listing 4.11: Implementation of the `AddNode` method of CouchDB

Job Queue

A job is the only way to modify the cluster state in Ganeti. During its lifetime, a job interacts many times with the disk, as it passes from the several states of its execution. The interaction involves the information of the on-disk representation of the job, to correspond to the latest updates. The policy of Ganeti obliges every local update of a job to be spread among the master candidate nodes. That necessity, creates a bottleneck and reduces the throughput of the job queue and consequently the execution of jobs, specifically when many jobs are sent concurrently to the master I/O thread for execution.

Our objective, is to take advantage of the CouchDB replication process, and write a driver without the need to replicate jobs inside Ganeti. We believe that splitting

some of the work ,i.e., the replication task, to a separate thread, not tied with the Ganeti code path, will increase the job queue throughput. This actually means that in bigger clusters with a lot of congestion due to the number of clients who concurrently talk to the master daemon, the job execution will also be fasten up because jobs will go to the `Queued` state earlier than currently are and the waiting threads will grub them sooner too. In addition, the timeouts happen to the LUXI server due to a heavy loaded master daemon will be limited.

Database Structure

The structure of the database does not differ a lot from the original disk representation. We will create two new databases, named `queue` and `archive`, for the job queue and the archive directory respectively. In more details:

- *queue*, containing one document per job, and indexed by the job identifier.
- *archive*, containing one document per archived job, and indexed by the job identifier too.

The `_QueuedJob` class, which corresponds to the in-memory representation of the job, already contains an `id` parameter which will be used for the CouchDB index needs ,i.e., `_id` parameter. We will add an extra `rev` field that will be used as the job's version, and will be `None` by default if the job have not been written to disk yet. Otherwise it will contain the last revision number of the job.

CouchDB Driver Design

The document containing a job, will have three fields; the common `_id`, and `_rev` fields and a new one named `info`, which will contain the `_QueuedJob` instance. The same will apply for the documents in the archive database, but with the addition of an extra field named `archive_index`, which corresponds to the original classification of the archived directory per *10.000* jobs. With a simple query of a view in the archived database, we could easily fetch the desired jobs from the given range.

For the CouchDB driver needs, three main classes should be created; the `_CouchDBJobFileChangesWaiter`, the `_CouchDBWaitForJobChangesHelper`, and the `CouchDBJobQueue` classes. We are going to present the main attributes of each one of those classes. For deeper investigation, the code is hosted as Github [23]. The `CouchDBJobQueue` class interface, the related class for the queue management for the CouchDB driver, is presented at Table 4.2:

We are about to present the methods of the `CouchDBJobQueue` class with the greatest importance. The relevant code will be presented, where it is necessary.

- `__init__()` method:

From the constructor method, we distinguish the creation of two new local class variables; the `self._queue_db` and `self._archive` variables, containing the relevant `couchdb.client.Database` instances, similarly to the configuration driver.

- `_UpdateJobQueueFile method:`

This is the method that writes a job to the database. Actually it is a wrapper method over `utils.WriteDocument`, which is the responsible method for the write

[23] `https://github.com/dblia/nosql-ganeti`

CouchDBJobQueue Class
`__init__(self, context)`
`_InspectQueue(self)`
`AddNode(self, node)`
`RemoveNode(self, node)`
`_UpdateJobQueueFile(self, data, job)`
`_RenameFilesUnlocked(self, arch_jobs, del_jobs)`
`_NewSerialsUnlocked(self, count)`
`_GetJobIDsUnlocked(self, archived=False)`
`_GetJobsUnlocked(self, archived=False)`
`_LoadJobUnlocked(self, job_id)`
`_LoadJobFromDisk(self, job_id, try_archived, writable=None)`
`_UpdateQueueSizeUnlocked(self)`
`SetDrainFlag(self, drain_flag)`
`UpdateJobUnlocked(self, job, replicate=True)`
`WaitForJobChanges(self, job_id, fields, prev_job_info, prev_log_serial, timeout)`
`_ArchiveJobsUnlocked(self, job_list)`
`ArchiveJob(self, job_id)`
`AutoArchiveJobs(self, age, timeout)`

Table 4.2: Interface of the `CouchDBJobQueue` class

requests, as we said in the configuration driver section. If a job is written for the first time, `_rev` field is None, we create a document with a empty `_rev` field. If a modification made in a job in the `queue` database, besides the data and the `_id` field, we should also provide the `_rev` field of the document we are about to change. This stands for all document updates in the CouchDB, because as we said CouchDB uses a *MVCC* policy, and the right document version must always be provided in order to avoid conflicts. The `rev` attribute we added in the `_QueuedJob` class, keeps the last revision number of the job and it is always updated with the most recent version of it, after each successful write. This speeds up the write requests, otherwise we should first fetch the document from the database to get the right revision, before we update it, which would introduce a great overhead to the whole operation. The same policy used for all the documents in the `queue` database, like the `serial` and `version` files.

- `_Get{Job/ID}sUnlocked methods:`

These two methods return all the jobs from the `queue` database, and the `archive` one if requested, and all the job identifiers respectively. Consequently, we want a fast way to retrieve all the documents of those databases. We made use of the view mechanism again. We defined a view which returns only the jobs of the database where it runs, and not other documents like the `serial` or the `version` ones. The fact that a view runs only in the newly inserted documents and not in those already ran, speeds up that operation. The view, as we said in the Apache CouchDB 4.4 section, must be defined inside a `_design` document. In our case, this is the `_design/queue_view` document, and contains a single view named `jobs`. This document is presented in Listing 4.12. If we query the view with the `_included_docs` option set to `True`, it

will return us a list with all the documents of the database. If that value is set to `False` we will get only the job IDs and no other information.

```
{ ''_id'' : ''_design/queue_view'',
  ''language'' : ''javascript'',
  ''views'' : {
    ''jobs'' : {
      ''map'' :
        ''function (doc) {
          var q; q = doc._id.indexOf('_');
          if ((doc._id != 'serial') && (doc._id != 'version') &&
              (q != 0)) {
            emit(doc._id, doc)
         }}''
    }
  }
}
```

Listing 4.12: View in CouchDB for job retrieval from the `queue` db

- **`AddNode method:`**

This method is responsible to enable the replication tasks for the `queue` and `archive` databases, between the master and the new candidates. To enable/disable the replication, we make a call to the `utils.UnlockedReplicateSetup` method, just like the relevant method from the configuration driver.

- **`_RenameFilesUnlocked method:`**

This is the last method we are going to present from that class. It is used for the archival of the jobs of the queue. This method may need to archive hundreds of jobs at a time, so we made use of the bulk update feature of CouchDB, which updates the given list of documents using a single HTTP request. After we collected the documents to be archived, we simply pass them as input to the `update(documents, **options)` method, which performs the bulk update.

- **`Notify Manager:`**

A challenge we faced during the job queue driver design, was the implementation of the notifying manager we will use while waiting for changes in a job. We can not make use of the inotify manager because the jobs will be at the database and not at the filesystem. We decided to use the CouchDB `_changes` feed, where all the changes made to a database are recorded. This feed is also used by the replicator process, a topic that we have extensively covered earlier in this chapter. Two classes involve in the waiting procedure. The `_CouchDBWaitForJobChangesHelper` and the `_CouchDBJobFileChangesWaiter` classes. Those two classes with the aid of the default `utils.Retry` function, provide the desired feature.

The `_CouchDBWaitForJobChangesHelper` class is almost identical with the original `_WaitForJobChangesHelper` class, and it will not be presented. we will focus on

the `_CouchDBJobFileChangesWaiter` class which actually implements the waiter. Unlike the default class, this one consists of two methods only; the `__init__()` and the `Wait` method. The `_changes` feed and the relative functionality provided by CouchDB have been used for the waiter implementation. The `_changes` feed contains every single modification made on the database. It would be a great waste of resources to search the whole feed for a modification of a single job at a time. This is why we created a filter function, which only searches for changes in the job ID we want to. This filter function is named `job_id`, and is contained in a design document just like the view functions. The relevant document is presented in Listing 4.13. Another decision we have to make, was the way we will poll for results in the `_changes` feed. The most appropriate choice was the `longpoll` feed with a timeout to close the feed if nothing has changed. It is a very efficient form of polling, which avoids the need to frequently poll CouchDB to discover nothing has changed. It does not run any requests if nothing changed, but as soon as a result appears the HTTP connection between CouchDB and the Ganeti client closes. The timeout that was selected is identical to the one used by the default inotify condition. The last parameter of the `_changes` feed we have made use of, was the `since` parameter. By providing the last sequence number to that parameter, we are waiting for new notifications only, ignoring the old ones. The sequence number refers to the number of the updates that have been made to the database, because any new update generates a unique sequence number. The `last_seq` value corresponds to the upcoming update. As presented in Listing 4.14, if an event happened (`have_events['results'] == True`), we return the relative `_QueuedJob` object to the client. Otherwise, if the result returned is `False`, the `utils.Retry` function will re-poll repeatable to the desired job until an event happens.

```
{ ''_id'' : ''_design/filter'',
  ''language'' : ''javascript'',
  ''filters'' : {
    ''job_id'' :
      ''function (doc, req) {
        if (doc._id == req.query.id)
        { return true; }
        else { return false; }}''
  }
}
```

Listing 4.13: Filter function in CouchDB

```
class _CouchDBJobFileChangesWaiter(base._BaseJobFileChangesWaiter):
  def Wait(self, timeout):
    '''Waits for the job to change.

    @return: Tuple of ('Polling', result) format. If the timeout
             expires result is False, otherwise a new _QueuedJob
             object with the new data returned. 'Polling' is used
             to distinguish this case in utils.Retry function.
```

```
9
10    '''
11    assert timeout >= 0
12    result = False
13    timeout = int(timeout) + 1
14    have_events = self.db_name.changes(filter='filter/job_id',
15        id=self.job_id, feed='longpoll', include_docs=True,
16        since=self.since, timeout=timeout * 1000)
17    if have_events['results']:
18      try:
19        data = have_events['results'][0]['doc']
20        raw = json.loads(data['info'])
21        result = _QueuedJob.Restore(self, raw, writable=False,
22                                    archived=None)
23      except Exception, err:
24        raise errors.JobFileCorrupted(err)
25
26    self.since = have_events['last_seq']
27
28    return ('Polling', result)
```

Listing 4.14: Waiting manager function of CouchDB on the job-ID given

Utility module for CouchDB

In the configuration and job queue CouchDB drivers, we made a lot of references to the relevant `utils/couch.py` module. The creation of such a module was essential, in order to make the main code of the CouchDB driver clearer, and also separate the main functionality of the driver from the utility functions needed, for the interaction with the CouchDB server. In Table 4.3, we provide the interface for the `couch.py` utility module:

utils/couch.py module interface
`URIAuth(user_info, reg_name, port)`
`URICreate(scheme, auth, path='', query='', fragment='')`
`DeleteDB(db_name, host_ip, port)`
`CreateDB(db_name, host_ip, port)`
`GetDBInstance(db_name, host_ip, port)`
`UnlockedReplicateSetup(host_ip, node_ip, db_name, replicate)`
`MasterFailoverDbs(old_master_ip, new_master_ip, db_name)`
`WriteDocument(db_name, data)`

Table 4.3: Interface of the utility CouchDB module

The `URIAuth` method, creates the authority value within a URI, while the `URICreate` method, returns a general universal resource identifier. The `{Create/Delete}DB`, `GetDBInstance`, and `WriteDocument` methods, have a quite straightforward usage. A special mention have to be made for the `UnlockedReplicateSetup` and the `MasterFailoverDbs` methods.

The `UnlockedReplicateSetup` method is responsible for enabling/disabling the replication tasks between the master node and the master candidates. During a server restart, we want the replication tasks to survive and continue from where they left of. In order to achieve that, we create a special replication document in the `_replicator` database. Those document persist a server restart and can only be modified by the database administrator. This is what this function does. Depending on the value of the `replicate` parameter, it creates or removes the desired replication document. The replication document contains four fields; the `source` URI, pointing to the master node, the `target` URI, pointing to the master candidate, the `create_target` attribute set to `True`, to create the database in the target node if it does not exists, and the `continuous` attribute set to `True` to make the replication process run forever. A continuous feed stays open and connected to the database until explicitly closed, and changes are sent to the client as they happen ,i.e., in near real-time.

The `MasterFailoverDbs` method, is called in case of a master-failover. In that case the replication documents from the old master node, are moved to the new master node. The replication documents must be removed from the old master, otherwise we will end up with a bi-directional replication process with unknown results. We should also take care of cases where the master CouchDB server is down, when the master-failover is requested. In that case, we re-create the replication tasks to the new master candidate, and as soon as the old master is up again, it is informed that it is not longer the master node, and the cluster administrator should run a `redist-conf` command, which will remove the unwanted replication tasks from the old master.

Summing up, we presented the *Core* modifications made to the Ganeti code, to support multiple driver solution for the configuration and job queue management. Changes have also been made in other Ganeti parts, but will not be further presented, because are out of the scope of that document. For more details, and further investigation on the full changes list, refer to the above-mentioned Github link.

4.5.2 Feature Changes

The main feature-level changes will be:

- the extension of `LUClusterRedistConf` functionality.
- global cluster-level parameter.

Redistribute Config

Current State

Currently, `LUClusterRedistConf` triggers a copy of the configuration file to all master candidates and of the ssconf files to all nodes. It also distributes every additional file which is part of the cluster configuration such as the certificate files. This is a call which should not normally needed, but in some cases like an upgrade of the Ganeti software, or if the `verify` call complains about configuration mismatches, must be run to "re-synchronize" the cluster status.

Proposed Changes

With CouchDB driver, we may end up with such configuration mismatches in the following scenario. If the current master node fails, and the CouchDB server can not be contacted, a `master-failover` must be run. When the old master becomes available again, he will be informed by the Ganeti master voting procedure that he is no longer the master node of the cluster. The problem lies in the remaining replication tasks, that where never removed during the `master-failover` operation, because the old master node was unreachable. The `redist-conf` which would normally update its configuration values, will be extended to also remove the remained replication tasks, in order to avoid a bi-directional replication. Such solution is implemented with an additional function call to the `LUClusterRedistConf` class, named `_RemoveRemainedReplicationTasks`. This function will search about remaining replication tasks in the candidate's `_replicator` databases, and will simply remove them if there are any. The Ganeti replication tasks have a specific `_id` format ,i.e., `from_`*`source`*`_to_`*`target`*, and can be identified from different replication tasks for other purposes.

Global Cluster parameter

Current State

Currently, the configuration data and job queue are stored to disk, by default. There is no alternative storage solution provided. It would be nice for Ganeti users to have the ability to chose between several cluster-wide parameters the type of the underlying storage solution they would like to use.

Proposed Changes

The pluggable modular driver transformation of Ganeti's base components which we attempted, enables that feature. We added a new cluster-level parameter, which will modify the underlying storage type of Ganeti per will. This parameter will be kept into the `cluster` dictionary of the configuration data, which allow us to create generic instance "classes" for the configuration and job queue storage handling. The default value will remain the disk storage.

4.5.3 Interface Changes

There is a single area of interface changes to expose the designed solution:

- *Command Line Interface*-level changes, *CLI.*

Command line changes

The new Ganeti feature we designed, introduces modifications in the way CLI arguments are handled. The command `gnt-cluster` will be modified and extended to allow setting, and changing the default parameter of the underlying storage type. The `init` command will be extended and a new command line argument will be added named `-backend-storage`, or `-S` for abbreviation. The default value will be **''**`disk`**''**, but user can also choose **''**`couchdb`**''** as a second alternative choice.

The generic syntax of the cluster init command is presented in Listing 4.15.

```
$ gnt-cluster init \
    --enabled-hypervisors=%hypervisors% \
    --secondary-ip=%secondary_ip% \
    --no-ssh-init \
           . . .
    --backend_storage=%disk|couchdb%
    %CLUSTER_NAME%
```

Listing 4.15: Extension of the `gnt-cluster init` operation

Chapter 5

Performance Evaluation

In this Chapter, we will evaluate the performance of the new CouchDB driver comparing to the default disk implementation, that is currently used by Ganeti for its storage requirements. Good benchmarks are non-trivial; each driver is different, and different use cases need to tune different parameters. In next sections, we will try to illustrate the benchmarking methodology of the diagrams we are about to present, before we proceed to the detailed explanation of our results.

The structure of this chapter is the following. Section 5.1 provides details about the hardware and software on which we conducted our benchmarks, while section 5.2 concentrates on the methodology behind our measurements; the main factors we have taken into account in order to decide which were the most interesting and applicable for Ganeti fields, were we should evaluate our driver's performance. Finally, section 5.3 contains the actual performance evaluation of the CouchDB driver. In each of its subsections concentrates on a specific field of interest for the driver evaluation, which will subsequently lead us to the final conjecture about how the driver responds to real-world workloads.

5.1 Specifications

To evaluate our software tool, it was required to setup a Ganeti cluster where our benchmarks would run. We decided to setup our testing cluster into a virtualized environment. More specifically, we chosen *Synnefo* [1], an open source cloud software, to host our testing environment. A bunch of reasons lead us to this decision. Firstly, as Ganeti is a software tool for managing clusters of physical nodes, it is not a facile task to obtain, setup, and maintain physical machines for the cluster requirements. On contrast, using a virtualized environment makes it quite easy to add and remove nodes from the cluster at will, without interacting with any physical processes that would be running in a physical machine. It provides a complete isolated environment where no one else can intervene with our work. In addition, keeping up-to-date snapshots of our virtual nodes makes disaster recovery quite a bit easier, and our cluster can be recovered in just a few seconds in case of a hardware or a software failure. Moreover, it makes incredibly simple and fast to modify the underlying hardware we use for our VMs, through the hardware abstraction it is provided.

[1] `http://www.synnefo.org/`

On the contrary, every virtualization solution comes with an additional overhead in terms of computation, networking, and I/O operations. The overhead incurred by virtualization has been the focus of many performance studies in the past, including numerous of general-purpose benchmarks. A short review on those, indicates an overhead below *5%* on computation [3], below *15%* on networking [3, 2], while the parallel I/O performance losses due to virtualization has been shown to be below *30%* [21], respectively. Recently, there also has been occurred a burst on the research activity related to the performance of using virtualized resources in cloud computing environments [6, 16, 10], that provide additional metrics of the effects of using the cloud computing services for running many types of scientific tasks.

In our case, the testing environment has been setup in a *7-node* cluster, where each node was armed with a *24-core AMD Opteron(tm) 6172* processor at *2.10 GHz*, with *189 GB* of primary memory, and *3.7 TB* of storage, running on a *SMP Debian GNU/Linux* with *3.2.0-4* kernel in *64-bit* mode. The virtualization software used, was *QEMU 1.7.0* with the aid of the *KVM* kernel module. When used as a virtualizer, QEMU achieves near native performance by executing the guest code directly on the host CPU. The redundancy on the physical resources of the nodes we setup our cluster, provides 1:1 mapping between the CPUs and vCPUS, as for the primary memory too. This results to a minimum performance overhead because there is no overcommitment in the physical resources at all. On the contrary, we can not come through the overhead during I/O operations with the disk and the network usage, but keeping in mind that both of our drivers interact with the disk and make use of the network resources, that overhead is linearly applied to each of them and will not affect the final results. The specifications of each one of the VMs constituting our virtual *5-node* Ganeti cluster, where we conducted our benchmarks, are the following.

Component	Description
CPU	8 x QEMU Virtual CPU Version 1.7.0
RAM	8192 MB
Disk	80 GB

Table 5.1: Test-VM hardware specs

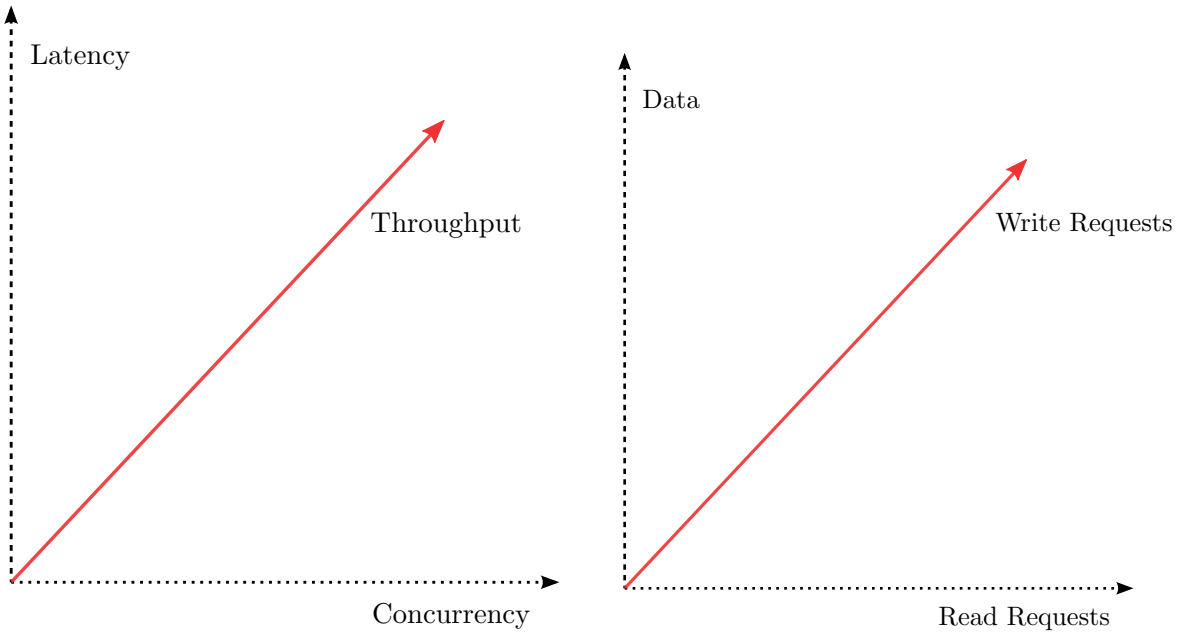
Software	Version
OS	Debian 7.1 Wheezy Base System
Linux Kernel	3.2.0-4-amd64

Table 5.2: Test-VM software specs

5.2 Benchmark methodology

Real benchmarks require real-world load. We will try to test our driver on real-world examples under situations when hundreds of clients try to interact with the master daemon concurrently, meaning that the masterd has to deal with them properly. Ganeti is a distributed software tool. So it is a premise to scale well and perform-fast, when used in real production environments with tens of nodes on each cluster.

There are a plenty of attributes affecting the performance of distributed systems and multiple "knobs" we could turn on to make a system perform better in one area, but affecting another area when doing so. A use case is the CAP theorem that was discussed in section 2.3.1. If we want our system to scale out, for example, there are three distinct areas to deal with; increased read and write requests, and data. In addition, reducing latency for a given system, affects concurrency and throughput capabilities. These two examples are graphically illustrated in Figure 5.1. Orthogonal to those attributes, there are many more factors that affect a system such as Ganeti, and more of the figures below can be drawed, that display different features such as reliability, simplicity, availability, and more. CouchDB is very flexible and gives us enough tools to create a system shaped to suit many, but not all, of our problems.



(a) Performance: Throughput, latency, or concurrency

(b) Scaling: read requests, write requests, or data

Figure 5.1: Compromises of distributed systems

Keeping the above in mind, the benchmarks we have executed can be roughly classified into four main categories, all of which having their own specific goals.

The first category, aims to expose the effect in the job submission rate, when the number of the master candidate nodes increases. In order to effectively measure the overhead that is introduced with an increase in the `candidate_pool_size` of the cluster, we will sent concurrently *errored* jobs to the master node, and then we will measure the rate that Ganeti adds them to the queue. We intentionally chose to sent errored jobs, because we simply want to observe how the job enqueue throughput of Ganeti responds in various candidate pool sizes, and not any other factors that may be affected by that type of measurement. The behavior we will observe, will also indicate a suitable size of master candidate nodes, to conduct the rest of our tests.

The second category, is a performance comparison test between the two driver implementations; the default disk implementation that Ganeti currently uses, and the CouchDB

one. The comparison concentrates on the job submission rate, by sending concurrently *errored* jobs of various sizes similarly to the previous category, but using a constant value of master candidate nodes instead, the one that was determined previously. The aim is to measure the performance against a tough workload, explain the differentiations, if any, that are observed on both of the drivers, and expose the factors that have the greatest impact on each of our drivers.

The third category deals solely with the configuration data management. We measure the performance of the configuration related write requests, on several file sizes, and in clusters with various candidate pool sizes, to examine the bottlenecks on having a single configuration file on big production clusters mainly. We will investigate all the factors that are delaying the update of the configuration file, on both of the drivers, and then we conduct a comparison test among them.

Lastly, in the forth benchmark category, we position the CouchDB and the disk driver in a real-world scenario; an attempt to create a bunch of instances and compare the total execution time of those jobs on each implementation. We aim to present the overall performance of each driver, and explain any differentiations that may arise.

5.3 Evaluating CouchDB

The overview of the benchmark methodology we provided, points out the various situations along with different metrics and workloads where we tested our implementations, in order to measure accurately their performance. The abovementioned categories, are presented in more details in the upcoming sections.

5.3.1 Impact of the candidate pool size

This category is a sort of an introductory section for the rest of our tests. Ganeti maintains a set of master candidate nodes, those that also contain a copy of the full cluster configuration, i.e., configuration and jobs files. The existence of those nodes has a great impact in the overall performance of the cluster, due to the fact that each modification in a disk configuration file causes a copy of it file to the candidate nodes. Creating a cluster with no master candidates at all is a risky attempt, because in case of a master failure all the cluster information will be lost. On contrast, maintaining a lot of master candidate nodes is a redundant waste of resources, as modifications have to be replicated to more nodes. We would ideally want to find out the set of master candidate nodes that fits a production environment, in terms of having the less impact in the cluster performance, and reducing the probability of a cluster failure.

In order to decide which is the most appropriate candidate pool size, we proceed with the following scenario. We sent jobs concurrently to Ganeti with varying candidate node numbers, and we measure the rate in which jobs are submitted to the queue, for each case. This metric is the job enqueue throughput, and denotes the average number of jobs that are added to the queue per second. It is a representative metric for our purpose, because every new entry to the job queue will also be replicated to the master candidate nodes before the operation is declared as successful. Since we are just interested for the enqueue rate, we decided to submit jobs that will never be executed and will be declared

as *errored*. An example of those jobs is the modification of an instance that does not exist in the cluster. The jobs will be normally inserted to the queue and replicated to the candidate nodes, but when they will start their execution, they will immediately fail as *errored*.

Jobs have been sent to Ganeti in batches of `10`, `20`, `30`, `40`, `50`, `100`, `150`, and `200` jobs. We ran that benchmark in a *5-node* cluster consisting of `none`, `one`, `two`, and `four` master candidate nodes, and the whole procedure has been repeated ten times in total. Since Ganeti writes every information to filesystem and then distributes it to the candidate nodes, there is a lot of disk and network I/O interaction. As a result, we expect a short of deviation in our sample data values because there are external factors that may affect the performance. The "outliers" values that may arise should also be included in the final results, because are part of the Ganeti behavior. Consequently, we believe that the *mean* value of our distribution is the most appropriate metric for our case, because is a metric that represents the *central tendency* of the distribution by taking into account the whole data information.

The benchmark outputs are summarized in two figures. Figure 5.2, presents the total results in a normal *line-points* plot style, while Figure 5.3 concentrates on the heavier workload of our benchmark that are closer to a real-world environment, in a clustered *bar-graph* plot.

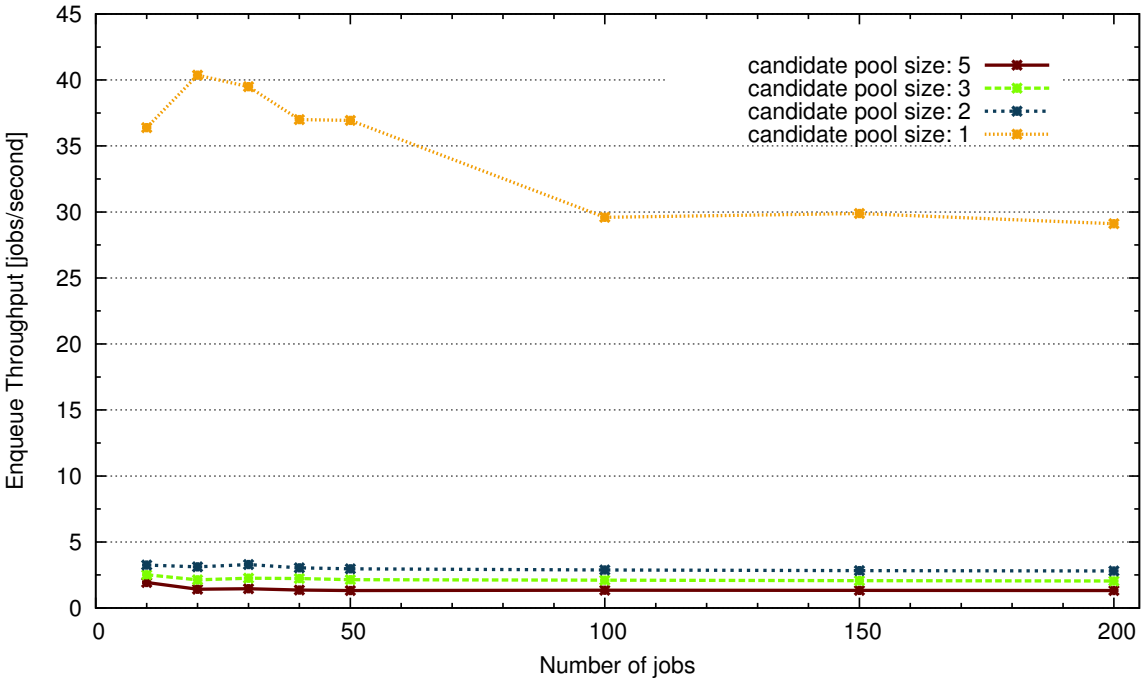


Figure 5.2: Job submission rate per number of candidates

Performance Analysis

There are several interesting points that we can conclude from these diagrams, which we will address below. The most obvious observation is the significant drop in the throughput performance, as we increase the master candidate nodes of the cluster. Furthermore, in case of no master candidate nodes, the distribution appears skewed characteristics and

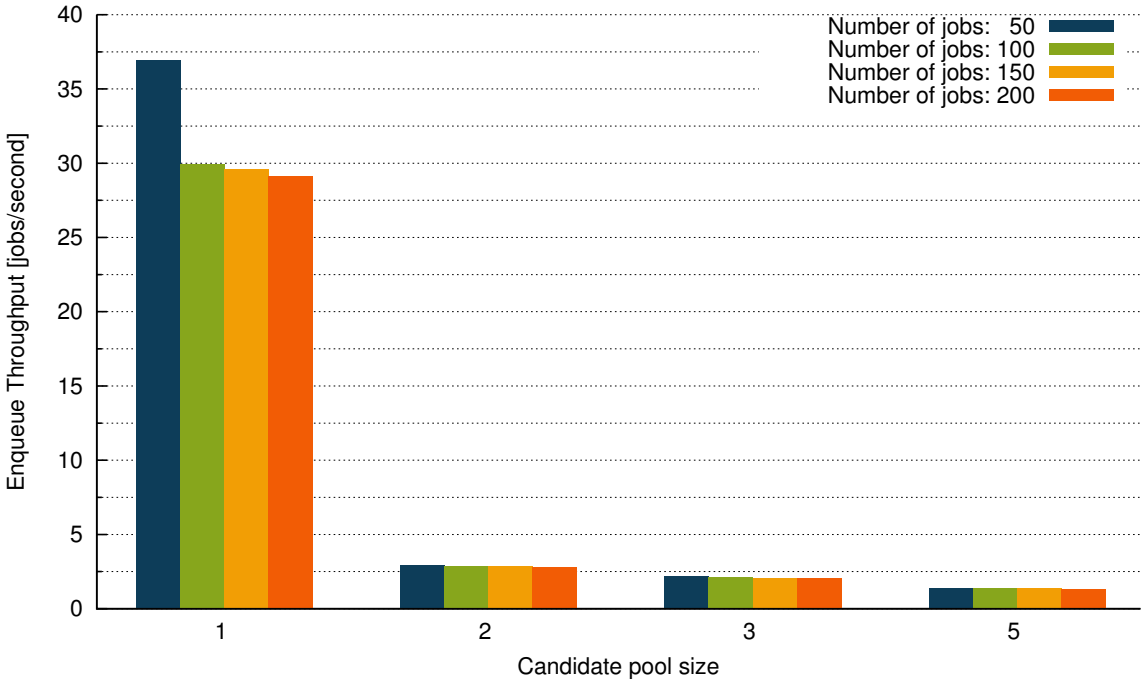


Figure 5.3: Job submission rate per number of candidates #2

visible differentiations in the throughput performance, something that is not observed when the candidate pool size is increased. A closer explanation of those points follows.

In both figures, there is an obvious relationship between the job submission rate and the number of master candidates that Ganeti maintains. As we stated in the Caveats section, i.e., 4.2.3, Ganeti writes jobs to disk and then concurrently replicates them to the master candidates. The performance slowdown of the replication process is displayed in these figures. With a single master candidate node, we have a drop in the throughput of about *8 times* comparing to the case of no master candidates. An additional increase in the candidate number, has no significant performance drop. This is an expected behavior because Ganeti uses a multi-node RPC call to update the files in the candidate nodes. This small dropdown is totally expected due to the checks that Ganeti does after each RPC call, to get informed about the success, or not, of the operation.

Another interesting finding from our results, are the "outliers" that were observed in the output sample data. When we ran the benchmarks in a cluster with no master candidates, the data tended to have a more sparse behavior than in a cluster with one, or more, candidate nodes. In order to make that variation visible to the reader, we calculated the *standard deviation* metric of our distribution, which shows how much dispersion from the average value exists, and we present it in Figure 5.4 to justify our claim. The question that may arise is: *Why the deviation exists only in the case of a cluster with no candidates.* The submission rate of Ganeti is the rate that the data are written to disk, and replicated to the candidate nodes. Obviously, the disk and network I/O are some of the factors that affect our results. As we know, Ganeti writes the jobs to its queue, so when a worker thread grubs a job for execution, it will subsequently update the job file in disk too. This causes a lot of congestion in the job queue lock both from the master thread and the job queue workers. The order that the workers grub jobs for execution causes those differentiations

in the throughput performance, in case of no master candidates. On contrast, when the cluster contains master candidate nodes, we have a totally smooth distribution with data around the mean value. This behavior is justified by the replication process of the job files to the candidate nodes. It is a quite time-consuming operation, that covers the rest operations that are executed, and actually determines the result's form. At this point, many types of measurements can be taken that will clarify those claims and probably will expose more, but are out of the scope of this document and this test category specifically, and we will not expand further.

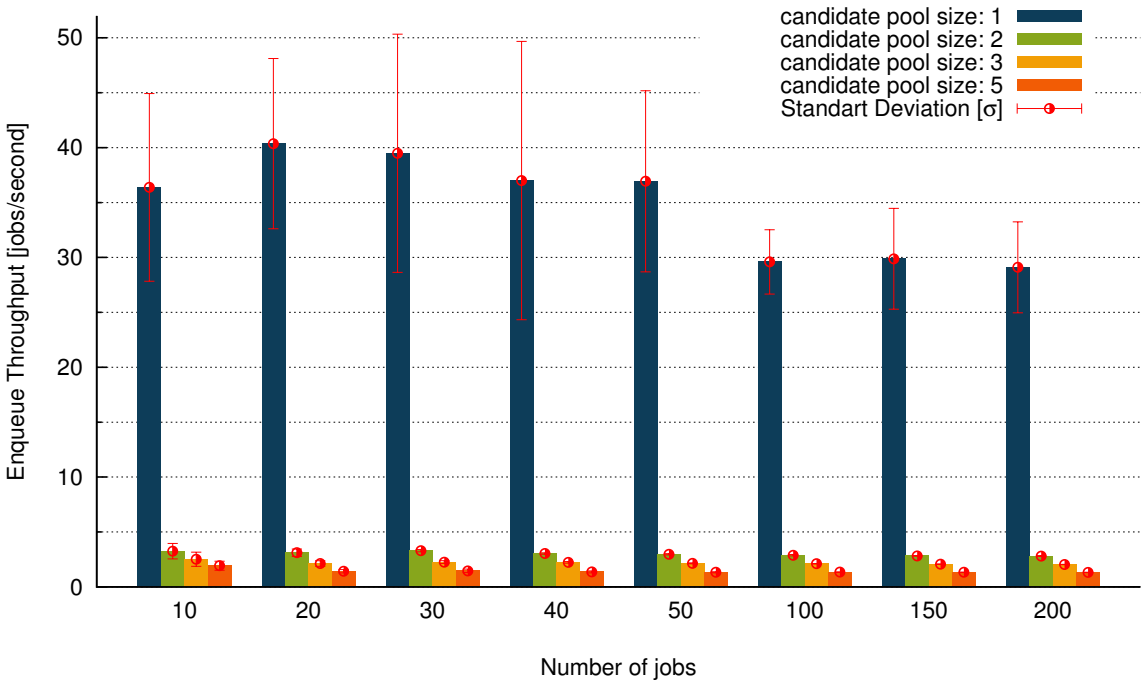


Figure 5.4: Standard Deviation [σ] of the job submission rate

Summing up, from the results we presented, we believe that having a cluster with two candidate nodes is the best choice to conduct the rest of our benchmarks. It is also a choice that reflects a real production environment because it provides the appropriate backup degree in case of a hardware failure, and also does not have great performance impact comparing to a cluster with a single candidate node.

5.3.2 Comparison of the job submission rate

To some extend, we already discussed about the job submission performance rate of Ganeti's default disk storage implementation, and we explained some of the drawbacks that start to appear as the candidate pool size increases. In this category, we make use of exactly the same metrics that we used in the previous one, with the difference that the tests are conducted in a cluster with a candidate pool size equal to *three* nodes, as we determined in the first benchmark category.

It is the first category where we actually compare the two driver implementations, and we explain the main factors that affect their performance. The results are summed up in two figures. Figure 5.5, contains the comparison of the job submission rate between CouchDB

and the disk storage type, while Figure 5.6 presents the comparison of the CouchDB driver performing under different socket options.

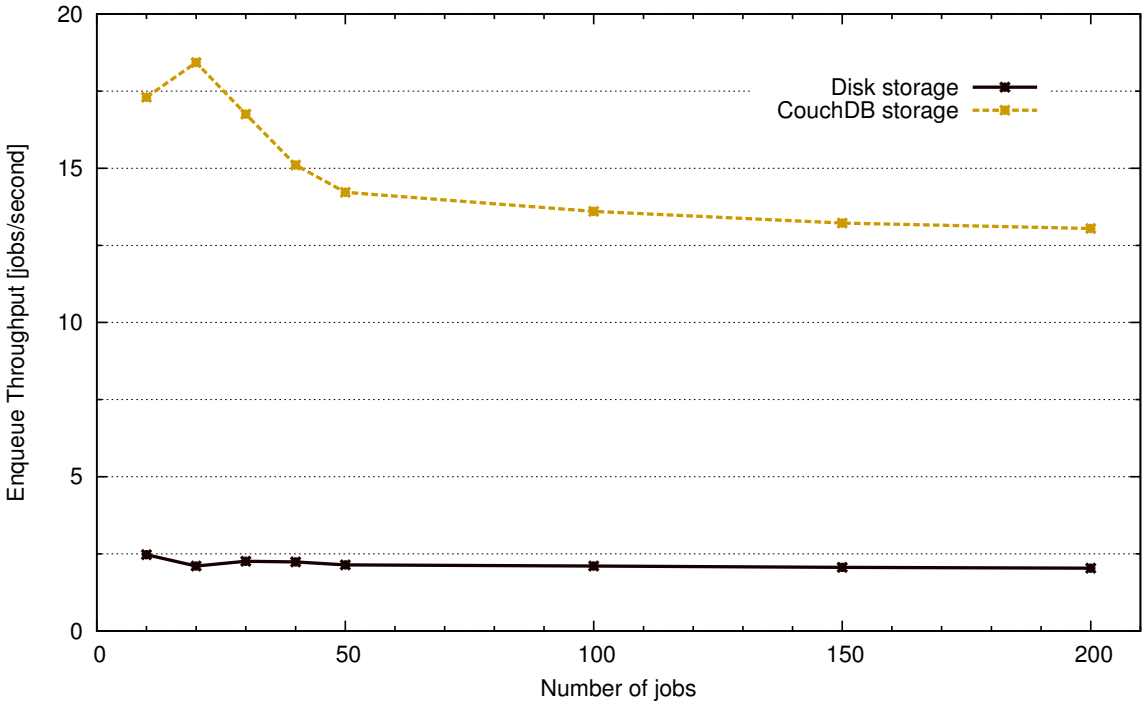


Figure 5.5: Comparison of the throughput performance

Performance Analysis

Before we proceed with the interpretation of the diagram results, we have to mention that the metrics we used, are exactly the same as in the first benchmark category, and the final values correspond to the *mean* value of every distribution too.

Figure 5.5, is the first performance comparison test that we made between the two peers. The initial results look very promising. In a *5-node* cluster with two master candidate nodes, we have a speedup in the job submission rate at about *7-times* in the CouchDB driver, comparing to the default disk storage implementation. The job submission rate has impact in the overall job execution duration, as we will show in the last benchmark category, and also reduces the timeouts happen in the LUXI server when many clients try to submit jobs to Ganeti. We extensively talked about the reasons that performance drops in a Ganeti cluster when jobs are saved to disk, in the first test category. Now we are going to have a closer inspection in the factors that prevent CouchDB from presenting similar behavior.

We discussed a lot about how Ganeti distributes the configuration files to the candidate nodes, and we presented the performance impact of this operation in the first category's figures. One of the main reasons we have chosen CouchDB for Ganeti, is the replication feature, as it was discussed in section 4.3. CouchDB is a database the replicates, and with this term we mean that its fundamental function is to provide a simple, fast, and convenient way to *synchronize* two or more CouchDB databases. Replication is handled completely by a separate process, external to Ganeti, which listens on changes to the *source* database, and replicates them to the *targets*. Obviously, the source refers to the master

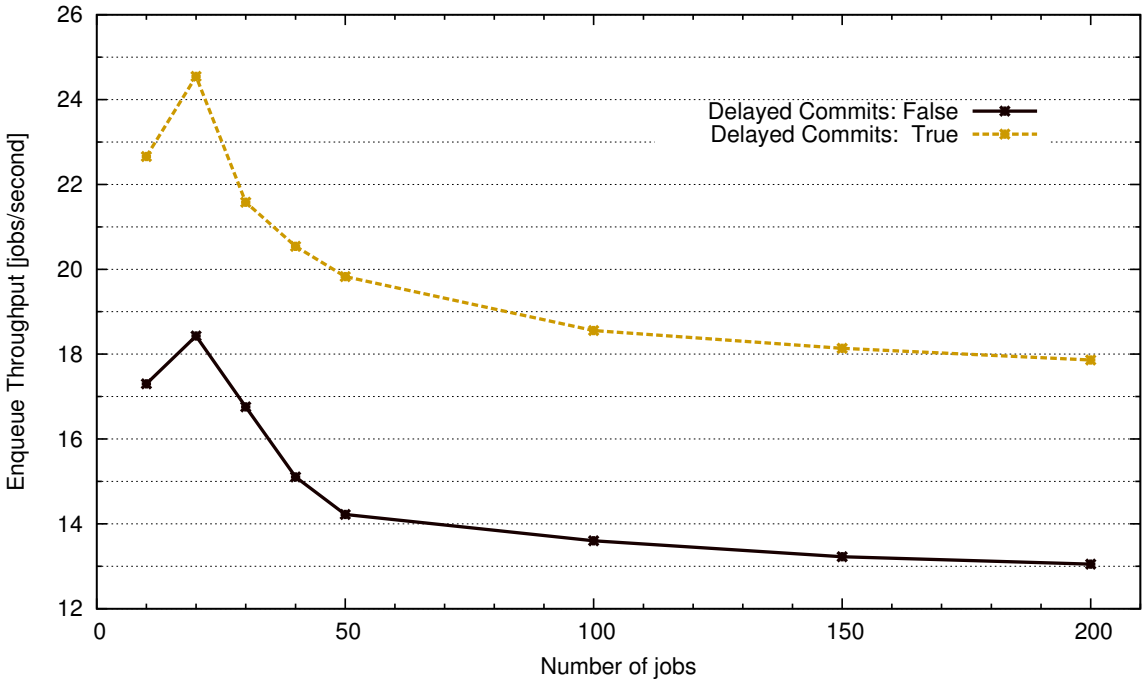


Figure 5.6: Throughput performance of CouchDB on various socket options

database, while the targets are the databases of the candidate nodes. The `replicator` process listens continually to the source's `_changes` feed, and a new modification to the source will immediately be replicated to the candidate nodes. In order to ensure safety, CouchDB makes an *fsync* call before a *201 Created* request is returned to the client. As soon as the nodes are "up" and running CouchDB will replicate, and there is no need to make extra checks similar to the RPC checks that Ganeti has to make to find out that the updates have reached a majority of the nodes, before declaring the operation as successful. Instead, it is sufficient to check that the CouchDB servers in the candidate nodes are accessible. This operation can be handled by individual clients, independent to Ganeti that will not affect the performance, and is an issue that is expected to be fixed in a future driver version, as we will discuss in the Conclusion, i.e., Chapter 6.

Besides the comparison test between the two implementations, we also make a comparison test for the CouchDB drive,r on various socket options that CouchDB provides, and have a great impact in the overall performance of the tool. Figure 5.6 shows this performance comparison test, on the `delayed_commits` attribute of CouchDB. When we set this attribute to `True`, it is observed a further increase in the throughput performance at about *9-times* comparing to the default disk implementation, and at about *35%* comparing to the CouchDB driver with this attribute set to `False`. Delayed commits is probably the most important CouchDB configuration setting for performance. When is set to true (the default), CouchDB allows operations to be run against the disk without an explicit *fsync* call after each operation. Fsync operations take time in order to complete, and calling them on each update limits the CouchDB performance for sequential writers. It is clear that setting this option to true, opens a window for data loss, because data are being kept in a write buffer and are fsync-ed after a certain amount of time, or when the buffer is full. Ganeti is an environment where we absolutely need to know when the updates have been received, so we set this attribute to false, by default. The aim of this test, is to expose

an important setting of CouchDB that could be enabled periodically, in several cases, like in a situation with an overloaded master daemon, and then disabled at will. It is up to the cluster administrator to measure the tradeoff between loosing some data in case of a hardware failure, and the "relief" that the performance improvement will bring to the cluster.

To achieve the results we presented for the CouchDB driver, another important configuration option must be modified, related to the TCP buffering behavior. This is the *nodelay* option which must be set to `True`, in order to disable the *Nagle's algorithm* [2], which introduces an additional delay when using keep-alive HTTP-connections. By setting this option to true, the *TCP_NODELAY* option is turned on for socket, which means that even small amounts of data sent to the TCP socket, like the reply to a document write request, or reading a very small document, will be sent immediately to the network. They will not be buffered hoping that it will be asked to send more data through the same socket in order to transfer them all at once. The main reason that this important option is disabled by default, is that the last releases of CouchDB ships with a more recent version of the HTTP server library *MochiWeb* [3], which by default sets the *TCP_NODELAY* socket option to false.

5.3.3 Comparison of the config.data performance

The CouchDB driver, besides the alternative storage solution that provides to the configuration files of Ganeti, also introduces a variation in the way it handles the `config.data` file, as it was extensively discussed in Section 4.2.1. The ultimate aim of this category is to compare the performance of the two alternative implementations of handling the configuration file, but before we reach to this point, we will investigate in deep all the factors that affect the configuration file performance, and that were discussed in Section 4.2.3.

In this category we will present three diagrams in total. The first two of them [5.7, 5.8], one for each driver, show the total execution duration of the `_WriteConfig` method, and all the sub-method calls that are been made. This is the responsible method for applying the changes of the configuration file to the permanent storage, and replicate them to the master candidate nodes. Every operation that modifies the cluster state calls this function to make the changes permanent. It is the most time consuming function related to the configuration file, and has a great importance in the performance of Ganeti, because is must be called with the `ConfigWriter` lock held in exclusive mode, which starts to become a bottleneck when a huge number of jobs is in execution. If we manage to reduce the time the lock is held by the workers, we will also reduce some of the congestion in the config lock. The last diagram [5.9], is the actual performance comparison plot between the two implementations.

The benchmarks of this section have been conducted on a cluster with a candidate pool size equal to `one`, `three`, and `five` nodes, respectively. In order to measure the performance of the `_WriteConfig` method, we intentionally increased the size of the `config.data` file, from `100 KB` up to `5 MB`. A cluster with about *2.000* instances has a configuration file of around `5 MB`, which corresponds to a real workload for a production environment. Our test concentrates on modifying a parameter of a single configuration object. We chose

[2] `http://en.wikipedia.org/wiki/Nagle%27s_algorithm`
[3] `https://github.com/mochi/mochiweb`

an instance object as a use case. Starting, restarting, or stopping an instance is a quite commonly used operation, that while it aims to modify a single field of the instance object, the whole configuration object is flushed to disk, and moreover the `ssconf_*` files are not affected; a variance that we do not want to take into account in this test category. The test was repeated *20* times for each pair, and we find it appropriate to make use of the *trimmed mean* value of our distribution. The trimmed mean is a method of averaging, that removes a small percentage of the largest and the smallest values before calculating the mean. This method aims to reduce the effects of the outliers on the calculated average, and stated as mean trimmed by $X\%$, where X is the sum of the percentage of observations removed from both the upper and lower bounds. In our case, we trimmed the mean by *20%*. The reason we did not calculate the normal mean value, is that we wanted to reduce the effects of the outliers that were observed, and to obtain a more accurate average performance overview, for both the implementations.

Performance Analysis

In the begging of our analysis, we will take a closer inspection on all the factors related to the performance of the write operation of the configuration file. An operation that affects the configuration state, passes from the following execution phases in general. Firstly, some preliminary checks are being made on the object that it is requested to change, and then the update of the in-memory representation of the `config.data` object follows. Then the `_WriteConfig` method is called, which flushes the updates to disk, and replicates them to the candidate nodes. This method consists of a number of time-consuming sub-method calls that affect the overall performance of any cluster operation that modifies the configuration file. These calls contain the verification of the config object for configuration errors, to maintain the consistency of the object, and is named `_UnlockedVerifyConfig`, the serialization of the config object in order to be prepared for applying the changes to disk, through the `serializer.Dump` call, and then the actual flushing of the in-memory object to disk, using the `utils.SafeWriteFile` function call. Finally, the modifications have to be replicated to the candidate nodes with the `_DistributeConfig` method. All these operations are affected from the size of the configuration file, and from the candidate pool size as well. Figure 5.7, extensively examines this behavior.

From Figure 5.7, the following conclusions can be made. The most time-consuming method, is the serialization of the configuration file. It is a totally independent cost from the candidate pool size, but it is 100% bounded with the size of the file. The file is stored in memory as a `ConfigData` object, a generic config object defined by Ganeti. In order to be saved to disk, it must be transformed to a string format, and this is the role of this method. The cost of the file replication to the candidate nodes is increases along with the number of master candidates, and the size of the file too. In the same sense, the verification check consumes a lot of time in bigger file sizes because it traverses the whole file, same as the time of the function that applies the changes to disk, but with the difference that even in bigger file sizes it does not consume a noticeable amount of time. From this diagram it is understood that in a cluster with three candidate nodes, even from a file of *1.0 MB* in size, it takes at about *1 sec* to complete a single operation. If we combine it with the congestion on the config lock, we can see the great impact of that delay in the overall cluster's performance.

It is obvious that a single configuration file comes with a number of disadvantages. Modifying a single field of the file requires the serialization of the whole config object and the

distribution to the candidates, as well. This approach reduces the cluster performance due to the increased operational cost that is implied. In Figure 5.8, we will present a different approach that the CouchDB driver introduces, by conducting the same test on the `_WriteConfig` method of the CouchDB driver.

In Figure 5.8, we observe a great improvement in the total execution duration of updating the configuration file. This differentiation can be justified by the different approach of CouchDB of handling the config object. The configuration file has been separated to its sub-components, as we extensively discussed in section 4.8. Modifying an instance, a node, and generally a single object of the configuration file, does not updates the whole object to disk, but only the single object we want to. Moreover, the serialization cost does not implies anymore. The transformation of an object before it is written to disk is a simple conversion to dictionary, and it is part of the `utils.WriteDocument` method. Since we convert only few kilobytes each time the cost is negligible. The distribution cost is also disappears due to the different approach of CouchDB on handling the replication process, as we already extensively explained. The verification cost is the only one that does not changes, since we continue to verify the whole in-memory configuration object.

The last diagram we are going to present in this category, is presented in Figure 5.9. It displays the total execution duration of an instance modify operation. It is a representative job, as all Ganeti operations modify a specific field of the configuration file. The same output would appear if we were modifying a node, a network, or a nodegroup. As we said in the *Implementation Details* section of the CouchDB driver, i.e., Section 4.8, every update causes two consecutive object updates to the CouchDB server. One for the cluster general information, and one for the object we want to change. In this figure, we present the total execution time for CouchDB, that is the aggregated execution duration of the two objects. Even with this overhead the difference of flushing the whole file to disk comparing to the single object that is updated is significant. For example, for a file of *2 MB* in size, we have at about *5-times* increase in the performance of CouchDB, while the gap is widening as the file size further increases.

5.3.4 Aggregate evaluation of the CouchDB driver

Up to this point, we have tested our implementation in a variety of situations that may occur in a Ganeti cluster. We also examined some of the main factors that limit Ganeti from scaling and achieving better performance. In this last benchmark category, we will attempt to measure the overall performance of our drivers in a real-world scenario. In order to explain the results of this section, we will also make use of the findings from the previous categories.

In a Ganeti cluster with *5 vm-capable* nodes, and three master candidates, we will concurrently submit jobs of *OpInstanceCreate* opcodes, in batches of `1`, `10`, `20`, `50`, and `100` jobs. We will measure the average time of the phases that a job passes through, and then the total execution duration from the first job that it is enqueued since the last that is completed. Since the `Running` times of the jobs are independent to the underlying storage layer that it is used, we will minimize it by creating instances with *1 GB* file disk, using the *–no-install*, *–no-start* options that disable the OS installation and the start-up of the instances respectively.

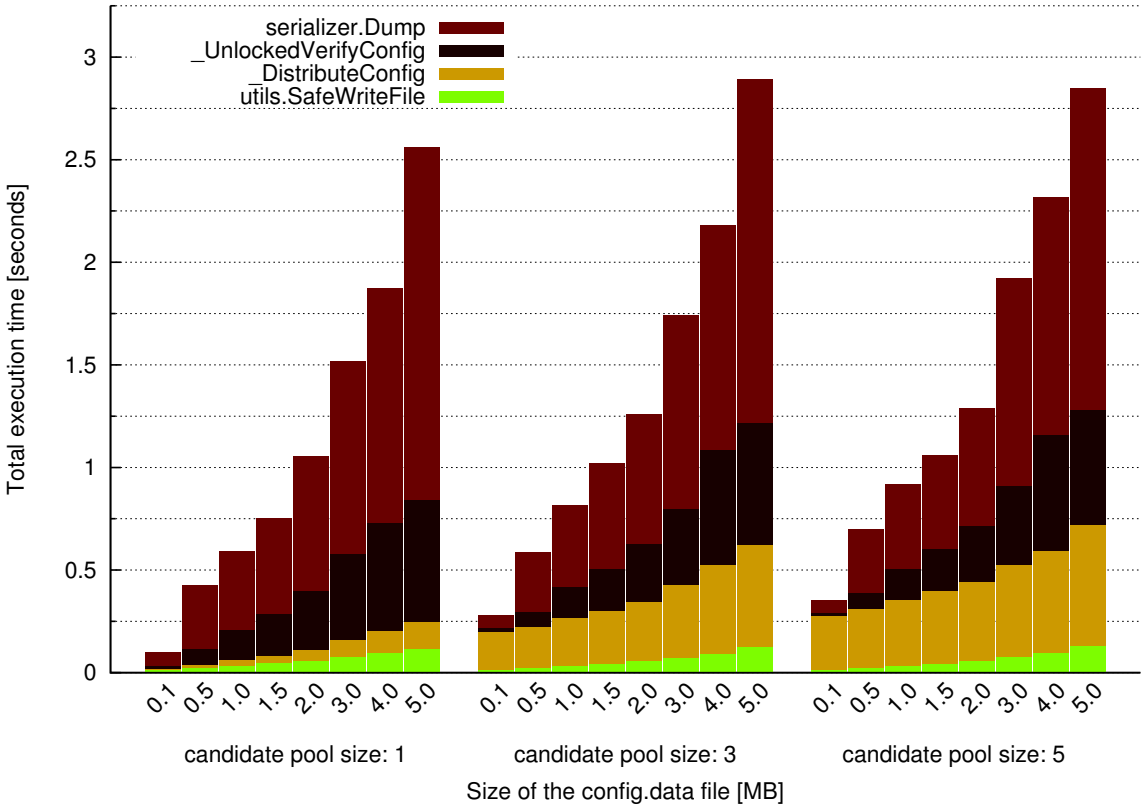


Figure 5.7: Performance evaluation of the default _WriteConfig method

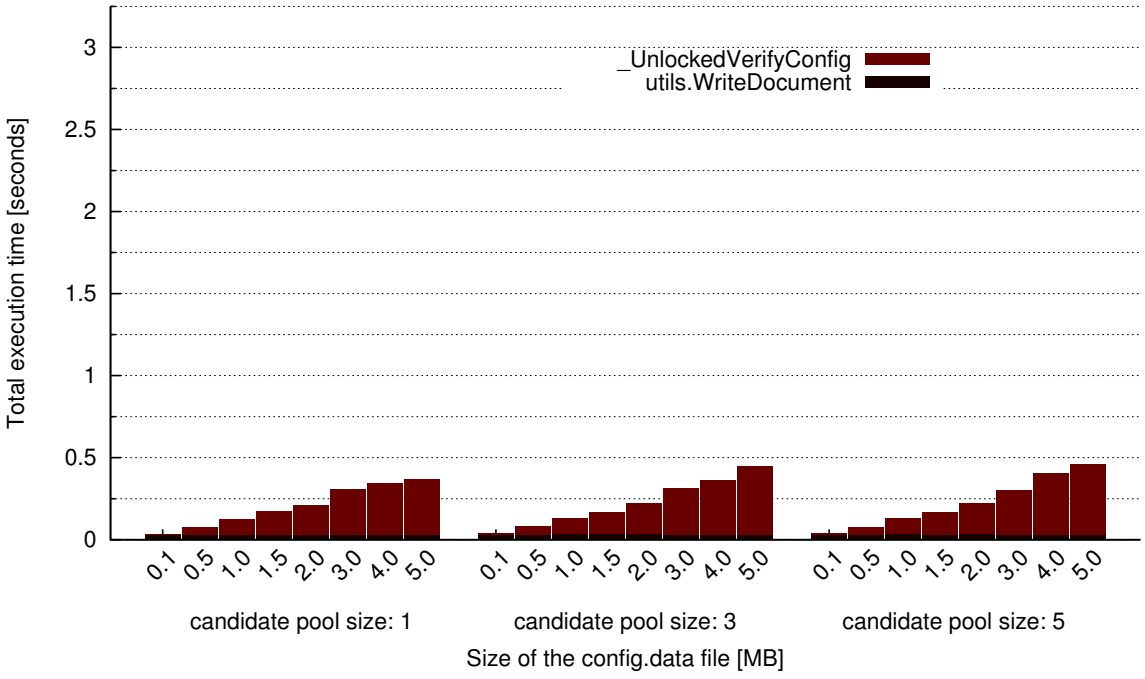


Figure 5.8: Performance evaluation of the _WriteConfig method of CouchDB

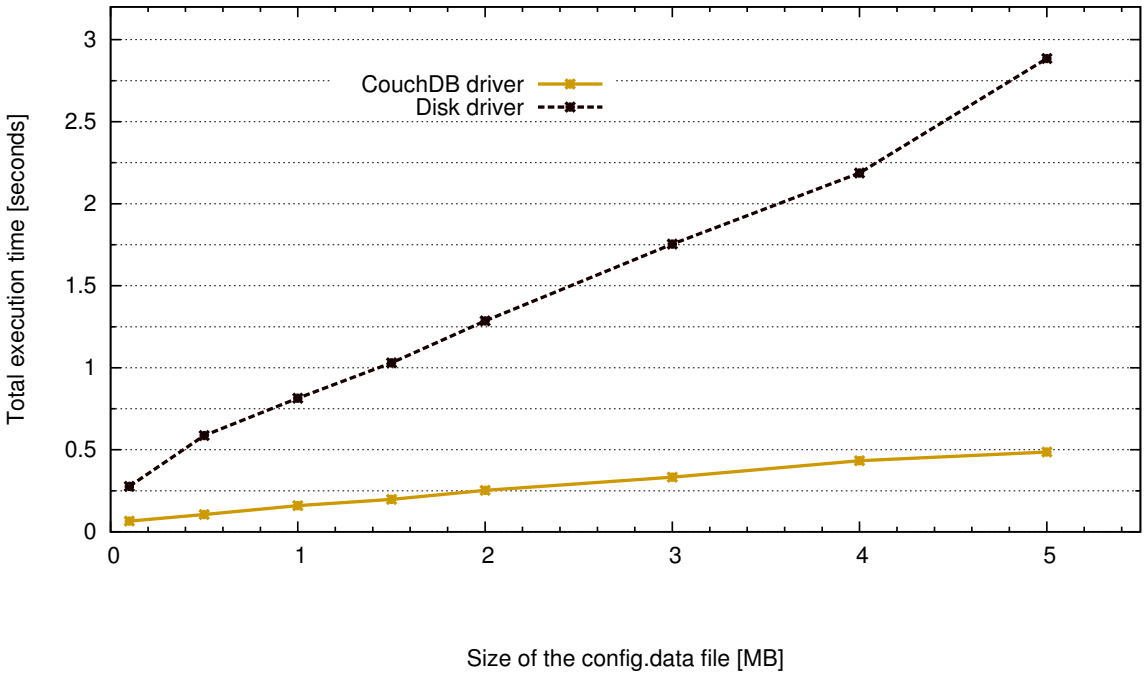


Figure 5.9: Comparison of execution performance for instance modify ops

Performance Analysis

Figure 5.10, displays the average duration of the execution phases of the *InstanceCreate* jobs we submitted. For a short reminder about the execution phases of a job, refer to Section 3.3.2.

What we observe from Figure 5.10, is a minimized *Running* time for reasons we already covered, while the most time is consumed in the *Waiting* phase. In this phase the jobs are waiting for locks, held by other threads that are in execution. The *Opportunistic locking* that it is used since Ganeti version *2.7*, improved the lock congestion in instance create operations, but since we create a lot of instances in a small cluster, it is a normal behavior. It is also observed that the average time of CouchDB in the *Queued*, and *Waiting* phase is quite smaller comparing to the disk implementation. We already covered the improved performance in the submission rate of CouchDB. The new finding, is the increase in the average *Waiting* performance time. This behavior can be justified by the increased job submission rate, as we presented in Figure 5.5. The worker threads, are waiting in the queue for new jobs to appear. As soon as a job is submitted in the queue, and a worker thread is available, it grubs it for execution. An increased job enqueue rate translates to workers that acquire their workload earlier. As a result, we have an immediate impact in the *Waiting* average time, due to the fact that the workers are idle for less time than they previously were, and the resources of the cluster are exploited more efficient than before. An immediate consequence of the increase in the performance of the *Queued* and *Waiting* time, is an increase in the total execution duration of the jobs. This induction is justified by Figure 5.11.

What we conclude from Figure 5.11, is that the CouchDB driver performs better under a heavy loaded environment. The performance gap between the two implementations widens as the number of jobs in the cluster increases. CouchDB is designed to service

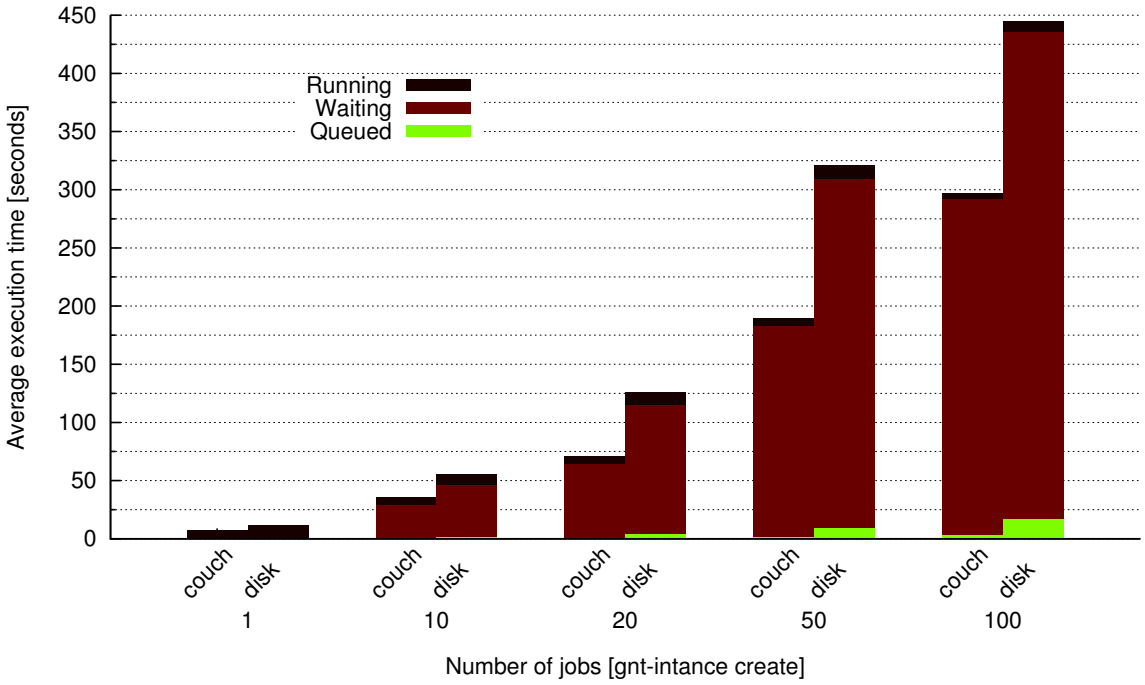


Figure 5.10: Comparison of execution performance for the phases of a job

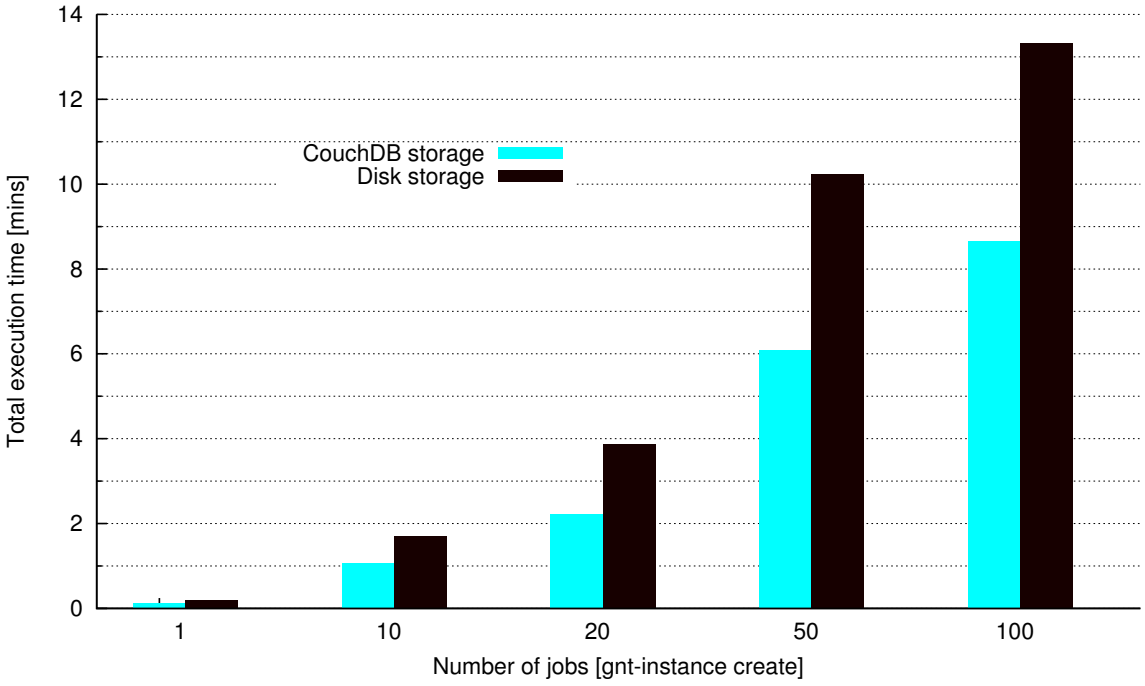


Figure 5.11: Comparison of the throughput performance for instance create ops

highly concurrent use cases, and perform under a heavy application load. The *Multi-Version Concurrency Control* that implements, makes CouchDB able to handle a high volume of concurrent readers and writers without conflicts to each other. As a result, there will not appear any performance gaps as the cluster workload is increased, and the requests will continue to be serviced efficiently.

Chapter 6

Conclusion

6.1 Concluding remarks

This thesis described the design decisions, the technical-issues, and the implementation details for replacing the Ganeti configuration and job queue storage engine with CouchDB, a distributed document-oriented database. The goal of this thesis was to examine if it was *feasible* to integrate a NoSQL database system in Ganeti, and measure the *efficiency* of this approach.

After introducing the necessary theoretical background information about Ganeti and all the related fields of interest, we presented the main drawbacks that Ganeti appears, and the options that were evaluated to chose a NoSQL system, and specifically CouchDB, to provide a solution to some of those issues. We continued with the design and implementation details of the new storage choice, and finally we evaluated our solution, in regard to two broad dimensions: performance, and scalability potentials. Special attention was given on being fully compliant with the current Ganeti requirements, such as maintaining the fault-tolerant attribute by putting the safety of the data first, security, and mainly not intervening with the parts of the Ganeti code that are not related with our implementation.

Looking back at what we have created, we can say that we also covered another need of Ganeti. The modular approach we followed for our design, supports both CouchDB and file configuration as different storage engines, with different limitations for each case. The current storage options can be easily extended due to the transformation that we made to the base configuration modules, and we could provide support for additional engines, such as MongoDB, or any other option that would fit the requirements of Ganeti. As far as the implementation complexity of the abstraction of some modules is concerned, we agree that we introduced an important amount of transformations to the code, but taking into account the extra features it introduces, and the fact that the code base is quite clear and straightforward, we can state that the benefits are greater than the cons, and the implementation can be consider *successful*.

Benchmark results were quite promising. Both the storage engines have been tested under various use cases and workloads. We measure the performance using a variety of metrics which we believe that correspond to a real-world environment. We have also studied the reasons that we consider responsible for the differentiations that appeared in the performance of the two engines. Based on this examination, we conclude that there are

good indications that the NoSQL approach will be able to give extra choices and provide additional support, to the single storage solution which is currently used by Ganeti.

We admit that there are more factors we should consider before trying this new approach to a real production environment. Since we are only in the first version of the CouchDB driver, and since some of the limitations we presented are fixed as of writing this chapter, and also considering the important performance gains we observed, we strongly believe that this work can become the basis for further work, and give the motivation for focusing on different approaches for the Ganeti storage engine. A further integration of this design in a demo environment, will help us gain the desirable feedback for more improvements, and perhaps will provide us the basis for new ideas and feature extensions.

6.2 Future work

The development of our implementation is far from over, as the tool has a lot of room for further improvements. Our tasks for the future, on some of which we are currently working on, will be classified into two separate categories. The short-term goals, and the long-term ones, which are the following:

6.2.1 Short-term plans

This section, mainly contains improvements of the current implementation, and the extension of the existing functionality.

Fixing configuration ACID issue
In Section 4.8, we discussed an issue that arisen from the transformation of the `config.data` file management. It is our primary priority dealing with this irregularity, and provide an efficient solution that will fix that issue, without affecting the overall performance of the tool.

Verification of the replication process
Currently, we do not make any kind of verification checks on the replication process. We do not have a way to ensure if a modification reached a majority of the candidate nodes, but we are based on the CouchDB server integrity. CouchDB will replicate, and all files will reach their destinations, as long as the server is running. We would like an external to Ganeti process, verify the status of the CouchDB instances, and warn the user for a possible failure. The current `ganeti-watcher` script, is a possible candidate that can be extended to provide the desired functionality, but more solutions can be discussed.

Improving the ssconf_* management
Ganeti maintains the `ssconf_*` set of files in all the nodes of the cluster. As a result, in an operation that modifies one or more of those files, the cost of applying the changes among the nodes of the cluster is aggregated. This cost is not negligible, and is one of the reasons that forbids Ganeti from scale. We could also add those files to the CouchDB server in a separate database called `ssconf`, and share this information only among the candidate nodes. Since, every node needs to have access to those files, we could give write permissions to the master node, while the rest nodes

will only have read permissions to the database. It is an important performance fix which is intended to be applied soon.

Import all cluster information
Currently, CouchDB hosts only the job queue, the archive directory, and the `config.data` file. The rest Ganeti information, such as the SSL certificates, the daemon related keys, or the abovementioned `ssconf_*` set of files, continues to be stored in the filesystem. We would like to import all those information in the CouchDB server, as well.

Backups for disaster-recovery
Ganeti keeps a backup for the configuration data file, as soon as a node is demoted from the master candidate role. CouchDB does not follow with this requirement. The code will be converted to follow that requirement, and subsequently it may be extended to keep flat backup files periodically, to add an extra layer of protection from hardware failures.

6.2.2 Long-term plans

This section contains our thoughts about extending Ganeti, improving its overall performance, and its ability to scale better in bigger clusters.

Improving candidate pool management
At this point, the CouchDB driver does not affect the management of the candidate nodes. Each node in the cluster has a running CouchDB server. As soon as a node is marked as candidate, the master node will extend its replication tasks to include this node as well. We are currently discussing another approach of managing the pool of candidate nodes. We could keep a set of hosts, even independent to Ganeti, that will be used for the candidate pool requirements only. These hosts will contain the live cluster configuration and the job queue information, which they will be shared. The master node will interact with those databases to store the cluster information, removing the requirement of maintaining the candidate pool exclusively inside Ganeti nodes. Using *Solid State Drives (SSDs)* in that small set of nodes is more feasible than in a whole cluster, and it will be really handy for CouchDB.

Clustering support
The above idea can be amplified by the clustering techniques that the NoSQL systems provide. Since the NoSQL systems are designed to serve large data sets, some additional features are introduced to provide redundancy and high-availability in any case. Clustering and auto-sharding are some of those features, that provide the ability to the NoSQL systems to store and share data across multiple machines, using efficient algorithms for handling the read/write requests, and support the data growth demands. Currently, CouchDB supports clustering using external applications such as *CouchDB Lounge* [1]. The merge of CouchDB that was announced with BigCouch, a Cloudant's clustered version of CouchDB, will bring soon all the clustering capabilities that currently CouchDB lacks of. Besides the facilitation in

[1] `http://tilgovi.github.io/couchdb-lounge/`

handling the set of candidate nodes, since they will be part of the cluster, with BigCouch merged in, CouchDB and Ganeti consequently will be able to replicate data at a much larger scale.

Extending storage choices

Abstracting the related to the storage management code of Ganeti, was our number one priority. We made it feasible to easily create additional storage engines for Ganeti. Moreover, the similarity among the NoSQL family that exists, can be exploited to design new driver solutions, such as `MongoDB` [2], compare their performance, and generally having the ability to choose among several storage options according to our application's needs.

Improving lock congestion

The NoSQL systems are using their own locking policy. CouchDB uses the `MVCC` method to handle read and write requests. MongoDB uses a `readers-writer`, or `shared exclusive` per database lock, to deal with concurrent readers and writers. We could improve the current locking situation and be based on the locking level layer providing by the NoSQL systems. The `ConfigWriter`, and the queue level lock are the first contenders to be removed, since any serialization to accessing the configuration file and the job queue would be unnecessary.

[2] `http://www.mongodb.org/`

Bibliography

[1] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. *SIGARCH Comput. Archit. News*, 34(5):2–13, Oct 2006.

[2] Aravind Menon, G. John Janakiraman, Jose Renato Santos, and Willy Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *In VEE '05: Proc. 1st ACM/USENIX International Conference on Virtual Execution Environments*, pages 13–23. ACM Press, 2005.

[3] Barham, Paul and Dragovic, Boris and Fraser, Keir and Hand, Steven and Harris, Tim and Ho, Alex and Neugebauer, Rolf and Pratt, Ian and Warfield, Andrew. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, Oktober 2003.

[4] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[5] G. Burd. NoSQL: An Overview of NoSQL Databases. April 2012.

[6] Deelman, E. and Singh, G. and Livny, M. and Berriman, B. and Good, J. The cost of doing science on the cloud: The Montage example. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12, 2008.

[7] S. Garfinkel. The Cloud Imperative. `http://www.technologyreview.com/news/425623/the-cloud-imperative/`, October 2013.

[8] Giuseppe DeCandia and Deniz Hastorun and Madan Jampani and Gunavardhan Kakulapati and Avinash Lakshman and Alex Pilchin and Swaminathan Sivasubramanian and Peter Vosshall and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In T. C. Bressoud and M. F. Kaashoek, editors, *IN PROC. SOSP*, pages 205–220, Oct 2007.

[9] C. D. Graziano. A performance analysis of Xen and KVM hypervisors for hosting the Xen Worlds Project. Master's thesis, Iowa State University, 2011.

[10] Iosup, A. and Ostermann, S. and Yigitbasi, M.N. and Prodan, R. and Fahringer, T. and Epema, D. H J. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):931–945, 2011.

[11] Koukis, Vangelis and Venetsanopoulos, Constantinos and Koziris, Nectarios. okeanos: Building a Cloud, Cluster by Cluster. *IEEE Internet Computing*, 17(3):67–71, May 2013.

[12] Koukis, Vangelis and Venetsanopoulos, Constantinos and Koziris, Nectarios. Synnefo: A Complete Cloud Stack over Ganeti. *;login*, 38(5):6–10, Oct. 2013.

[13] Markus BÖhm, Stefanie Leimeister, Christoph Riedl, Helmut Krcmar. Cloud Computing and Computing Evolution.

[14] Peter Mell, NIST. Big Data Tradeoffs: What Agencies Need To Know. `http://breakinggov.com/2012/11/12/big-data-tradeoffs-what-agencies-need-to-know-nists-peter-me/`.

[15] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[16] Ru Iosup and Simon Ostermann and Nezih Yigitbasi and Radu Prodan and Thomas Fahringer and Dick Epema. An early performance analysis of cloud computing services for scientific computing. *TU Delft, Tech. Rep., Dec 2008, [Online] Available.*

[17] Slater, Noah. Welcome BigCouch. `https://blogs.apache.org/couchdb/entry/welcome_bigcouch`, July 25 2013.

[18] J. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, May 2005.

[19] An Introduction to Virtualization, Amit Singh. `http://www.kernelthread.com/publications/virtualization/`.

[20] VMware. Understanding Full Virtualization, Paravirtualization, and Hardware Assist. November 2007.

[21] Weikuan Yu and Vetter, J.S. Xen-based hpc: A parallel i/o perspective. In *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on*, pages 154–161, 2008.