NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND
COMPUTER ENGINEERING
COMPUTING SCIENCE DIVISION
COMPUTING SYSTEMS LABORATORY

Resource Management of Virtual Machines in Cloud Environments

DIPLOMA THESIS

of

Apostolos N. Diamantis

Supervisor: Nectarios Koziris
Professor N.T.U.A.

Athens, September 2014

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Διαχείριση Εικονικών Μηχανών Σε Πλατφόρμες Υπολογιστικών Νεφών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Απόστολος Ν. Διαμάντης

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 26η Σεπτεμβρίου 2014.

.....	.....	.....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Νικόλαος Παπασπύρου
Καθηγητής Ε.Μ.Π.	Λέκτορας Ε.Μ.Π.	Αναπ. καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2014

.....

Απόστολος Ν. Διαμάντης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Το υπολογιστικό νέφος έχει εξελιχθεί ως ένας από τους πιο σημαντικούς τομείς της τεχνολογίας σήμερα. Το επόμενο βήμα σε αυτή την κατεύθυνση είναι οι συνομοσπονδίες νεφών, δηλαδή οργανισμοί που αποτελούνται από νέφη τα οποία διαχειρίζονται από διαφορετικούς παρόχους. Σε ένα τέτοιο περιβάλλον, οι χρήστες μπορούν να χρησιμοποιήσουν πόρους από οποιοδήποτε νέφος που ανήκει στη συνομοσπονδία, οπότε προκύπτει η ανάγκη επιλογής του καταλληλότερου παρόχου.

Σε αυτή τη διπλωματική, αναπτύξαμε ένα σύστημα που συλλέγει αυτόματα πληροφορίες από όλες τις εικονικές μηχανές μιας συνομοσπονδίας νεφών και τα αποθηκεύει σε μια κεντρική βάση δεδομένων. Έπειτα, δημιουργήσαμε ένα εργαλείο που, βάση των πληροφοριών αυτών, στοχεύει στην εύρεση του βέλτιστου συνδυασμού υπολογιστικών πόρων σύμφωνα με τις προτιμήσεις του χρήστη και τα κριτήρια επιλογής. Το σύστημα αυτό πρέπει να είναι κλιμακώσιμο, δηλαδή πρέπει να μπορεί να υποστηρίξει μεγάλο αριθμό από χρήστες και νέφη, ώστε να είναι σε θέση να λειτουργήσει σε ένα πραγματικό περιβάλλον. Οι σχετικές δοκιμές της υλοποίησής μας απέδωσαν ικανοποιητικά αποτελέσματα. Το σύστημά μας αυτή τη στιγμή υποστηρίζει νέφη που λειτουργούν με το λογισμικό Synnefo και μπορεί να επεκταθεί για να υποστηρίζει περισσότερα ακόμη λογισμικά νεφών.

Λέξεις κλειδιά

Συνομοσπονδία νεφών, διαχείριση πόρων, συλλογή πληροφοριών, συνάθροιση δεδομένων, κεντρική βάση δεδομένων, πρόταση νέφους, κλιμακωσιμότητα, python scripts, APEL, ~okeanos, Synnefo, Openstack, Kamaki

Abstract

Cloud computing has evolved as one of the most important areas of technology nowadays. The next step in this direction is cloud federations, i.e. unions of clouds that each is probably operated by a different provider. In such an environment, users can deploy resources from any of the clouds that belong to the federation, hence the need for a way to choose the most appropriate provider.

In this thesis, we developed a system that automatically collects information from all the VMs of a cloud federation and stores them in a central database. Then, we created a tool that ,based on this information, aims to discover the optimal combination of cloud resources according to the user demands and decision criteria. This system has to be scalable, in other words it has to be able to operate with a great number of users and clouds, so that it can work in a realistic scenario. We tested this aspect of our implementation and the results were satisfactory. Our system currently works with clouds running the Synnefo software and can be extended to support many more.

Keywords

cloud federation, resource deployment, information collection, data aggregation, central database, cloud proposition, scalability, python scripts, APEL, ~okeanos, Synnefo, Openstack, Kamaki

Acknowledgements

First of all, I would like to thank Professor Nectarios Koziris, who was the supervisor of my thesis and gave me the opportunity to get involved with ~~okeanos. I chose to conduct my thesis in the CSLab because of the courses that he taught, which interested and intrigued me a lot. Next, I want to thank all the members of the Computing Systems Lab, who were very helpful in many ways, not only concerning the thesis. At this point, I have to refer specifically to two people whose help was critical for my work. Nasia Asiki, who is a senior researcher of the CSLab, was my immediate supervisor. Nasia was always available when I needed her advice either at the lab, by mail or Skype and she guided me through the course of this project excellently. Georgios Goumas, who is a lecturer and a senior research associate at the CSLab, was also extremely helpful for the completion of my work.

Finally, I would like to thank my family for their support and guidance throughout the duration of my student life. Their advice, as well as the support of my friends and classmates, prepared a safe path for my future.

Table of Contents

List of Figures

Table of tables

Chapter 1

Introduction

Cloud computing is currently one of the fastest evolving and most popular areas of Computer Science and Information Technology. Compared to the traditional computing model, the cloud offers several benefits in terms of performance, scalability, reliability, productivity and independence. As far as the end user is concerned, cloud computing does not require from him to deal with infrastructure at all, but consists of a service providing resources such as storage, processing units, networks and applications. The provided resources can be tailored to his needs and can also be shared with other users, thus maximizing productivity. For example, a group of scientists can work on a computational project running in the same (virtual) machine in the cloud from different physical locations, without having to worry about the management or administration of the shared resources, while being able to dynamically modify them on demand, if needed. Their work is always updated and accessible to all the members of the team.

Apart from the scientific community, cloud computing is also a very strong tool for financial, accounting, administration and many other applications. Even entertainment is increasingly developing cloud characteristics, with content uploaded and ready to be shared across many different devices. In most cases, especially in the commercial ones, the provided quality of service is very critical. Moreover, the pay-as-you-go model is essential both for companies and for consumers, ensuring a flexible and more fair charging policy for any intended use.

The growing number of companies providing cloud-computing services has resulted in a heterogeneous cloud market, where users have the ability to select among a variety of offered services (e.g. VMs), pricing models etc. Additionally, a single provider cannot guarantee truly global coverage due to limitations of geographic presence, causing latency and communication issues, which have a negative impact to the quality of service provided. The solution to these problems is Cloud Federations, in other words unions of clouds. A cloud federation across different cloud resource pools allows user applications to run in the most suitable infrastructure environment. Moreover, such a federation allows a cloud provider to distribute workloads around the globe, move data between disparate networks and implement innovative security models for user access to cloud resources. Finally, it enables the client to choose the best cloud service provider in terms of price, availability and need of use.

The question that now arises is how to make more efficient, either for the cloud service provider or the client of a cloud federation, the decision of the cloud in

which the VM should be deployed. From the user point of view, one needs to know the price, the suitability and the quality of service of each cloud option for the intended use in order to make the correct choice. From the cloud providers' point of view, the load distribution and some more technical parameters and constraints (e.g. peaks in resource demand, possible maintenance issues) should be taken into account for this decision to be made. In most cases, a broker is used for the management of this data for all the federated clouds. If all this information is available to both parts and can be suitably processed, the user should be able to decide the best-fitted resources, but this is very complex to be achieved without the use of capable tools.

1.1 Thesis Motivation

The goal of this thesis is the implementation of a systematic way to aggregate in a central database real-time and historic information from the runtime execution of applications (e.g. statistics) and the cloud infrastructure specifications of multiple cloud infrastructures participating in a federation. The purpose of this is to be able to make efficient and fast decisions about the cloud solution that offers the best fitted resources.

The collection of all this data is a difficult task, mainly because of its vast amount. There can be thousands of VMs in a medium-sized cloud federation and each one has to provide its statistics. Also, the form of this data may differ according to the type of VM that sends it and their sequence has to be preserved. All the above requirements have to be fulfilled in a way that is automated, reliable and scalable and demands as few resources as possible.

After having all this data collected and stored in a central aggregation point, it must be properly processed in order to produce the optimal results. The system responsible for this processing has to be flexible and scalable, as the amount of data and the specifications are different each time. Moreover, quality and speed must be ensured. The best combinations of resources among different cloud providers have to be proposed according to the user demands, the QoS each provider offers and the historical information about actual performance metrics of their infrastructure.

Our motivation is to develop a system that is expandable, scalable and interoperable with different types of cloud infrastructures. This is why we exploited technologies and tools that can interact with different cloud software, such as OpenStack and OpenNebula (references). To showcase the developed system, our current implementation is based on Synnefo, an open-source cloud software used to power ~okeanos, which is GRNET's (Greek Research and Technology Network) cloud service for the Greek Research and Academic Community, providing Virtual Machines, Networks and Storage.

This implementation consists of two components, a client-side executable that needs to be installed in one of the user's VMs, collects all the data needed from all the VMs of his account and sends it to the second part, a central database in another VM, where an executable is responsible for the insertion of that data in the database and the proposition of the most suitable cloud for the deployment of a new VM. This VM, where the database exists, can either be one of the user's VMs (even the same that collects the data) in the scenario where the user makes the cloud choice, or a server-like VM that decides for every user of the federation.

The first part of the implementation is compatible with the Synnefo software, as it communicates through its API in order to collect the required information. It is easy, though, to add compatibility with any given cloud software, such as the open-source Openstack. The second part is independent of the Cloud Software and it only prerequisites MySQL and Python support. Since ~okeanos is a single Cloud, the tests that we ran included cloud sites instead of different clouds. However, the only thing that has to be changed in our project in order to fully perform in a cloud federation is the client-side data-mining executable, which has to be updated with the API of each cloud.

1.2 Thesis Structure

This thesis is organized in the following Sections:

Chapter 2:

> We provide all the necessary theoretical background for the concepts and entities discussed throughout this thesis and briefly discuss related work.

Chapter 3:

> We describe the architecture of every component of our implementation, its features and the criteria by which one cloud is preferred over another.

Chapter 4:

> We describe all the technical details of our implementation, including the software we used that was already available and the one that we designed. We explain why we chose that software and how we modified it in order to serve our goals. Finally, we outline what should be supplemented if support for more Cloud types is to be added.

Chapter 5:

> We test our implementation in a pseudo-Cloud Federation Environment, using Cloud Sites instead of separate Clouds, and evaluate its performance. Every decision derives from an objective function, the parameters of

which are discussed. We also observe its behavior in case of the current VM stats or their history.

Chapter 6:

We provide our conclusion remarks and examine the possibility of future extensions in order to include support for several cloud types and more advanced decision-making techniques.

Chapter 7:

Bibliography.

Chapter 2

Background

In this Chapter, we will cover the theoretical background that is required in order to make this document comprehensible to the reader. The concepts that will be explained are clouds, cloud federations and cloud brokers.

More specifically, Section 2.1 provides the definitions of cloud computing and describes its evolution and its characteristics, service models and deployment models. Section 2.2 describes cloud federations and explains the concept of brokering. Finally, Section 2.3 discusses similar approaches to our goal, demonstrating ways to propose the best matching cloud of a federation for the deployment of a new VM.

2.1 Cloud Computing

The growing importance of cloud computing in numerous aspects of the world of technology such as entertainment, business and research, have created the need for its specific definition. Given the diversity of its nature, there is no strict definition of cloud computing. However, according to the National Institute of Standards and Technology of USA (NIST) [1]:

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics**,** three service models, and four deployment models.

Below, we will discuss these characteristics, service models and deployment models as they are described by NIST. We will also point out the service and deployment models that are used in our project. But first, we will make a brief presentation of the historical evolution of cloud computing, in order to observe how the basic concept of resource sharing has taken advantage of the capabilities of the Internet and has resulted in what we call the cloud.

2.1.1 The evolution of cloud computing

One of the main ideas behind cloud computing is the resource provisioning from a shared pool. Having this in mind, the use of terminal computers in order to access the resources of a mainframe during the 50s can be considered as the starting point of the concept of cloud computing. Those terminals had no computing capabilities and their sole role was to connect their users to the central computer, where all of the processing took place [2].

The following decades came with the establishment of the personal computer, but also the concept of virtual machines was created. Virtualization made possible the simultaneous and parallel execution of operating systems in an isolated environment, which took the terminal-mainframe model to the next level. Moreover, the introduction of the Internet further enforced the client-server concept, giving an idea of what could follow.

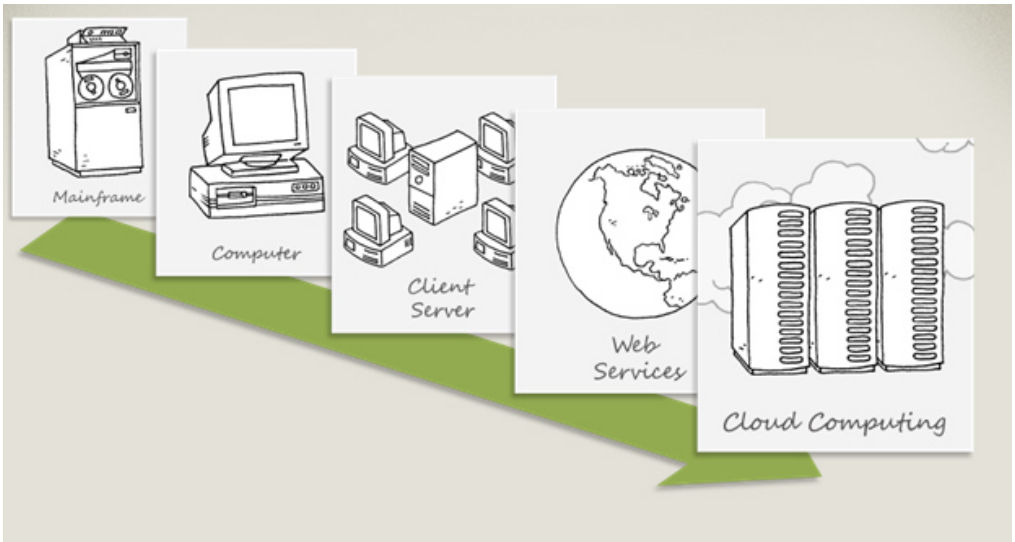

Figure 2-1: The evolution of cloud computing

In the 90s, telecommunication companies began offering Virtual Private Networks (VPNs), which were able to switch traffic in order to balance server use. It was the first time that the cloud symbol was used, marking the boundaries between the parts of the network for which were responsible either the providers or the users. The breakthrough of the World Wide Web established the concept of grid computing, where distributed systems worked under the same goal.

All these developments led to the formal introduction of the term of cloud computing in 2007, when Amazon presented Eucalyptus, the first open-source platform for deploying private clouds. Amazon used this software for the test version of its Elastic Computing Cloud (EC2). Since then, cloud computing has become a trending area of computer science and an increasing amount of cloud applications appear. Finally, as we probably approach a post-pc era, where smartphones and tablets surpass PCs in data and entertainment consumption,

cloud solutions guarantee that the end-devices are not burdened and limited by the storage and processing of data when someone else could take care of that [2].

2.1.2 The characteristics of cloud computing

The characteristics of cloud computing as described by the NIST [1] are the following:

- On-demand self-service:

The client can automatically modify the computing resources that he is offered, without the need of human act from the part of the provider.

- Broad Network Access

The services are available over the network for any supported device – client platform, such as mobile phones, laptops, tablets or workstations, and from whatever location. The only requirement is access to the Internet.

- Resource pooling

The total of the provider's resources of storage, processing, memory and network bandwidth are pooled and can be reassigned dynamically according to user demands. This multi-tenant model provides location independence and efficient use of the available resources, tailored to everyone's needs. Additionally, the pay-per use model is the user-friendliest, as one does not have to pay for resources that he does not use.

- Rapid elasticity

Resources can be elastically provisioned. The client can change the provided capabilities upon request and according to his needs without restraints and without affecting the operation of the existing resources.

- Measured service

The usage and provision of resources is monitored, controlled and optimized for both the provider and the user. This is possible by the use of metrics and statistics.

2.1.3 Cloud service models

According to NIST [1], the service model of a cloud can be one of the following:

- Software as a Service – SaaS

The user is provided with applications running on cloud infrastructure that he cannot modify and about which he does not need to know anything. These applications are accessible from a client interface compatible with many types of devices, most commonly a web browser.

- Platform as a Service – PaaS

The user is provided with the capability to create applications or application environments, using programming tools and services supported by the provider. He may have the possibility to conFigure the hosting environment, but he has no control over the cloud infrastructure.

- Infrastructure as a Service – IaaS

The consumer is provided with computing resources such as processors, storage and networks, on which he can deploy and run operating systems and applications. While he can modify the capabilities of the provided resources, he has no further control over the cloud infrastructure.

- Other models

On top of those service models, there are many subsets that are more specialized in specific needs and markets, such as Communication as a Service (CaaS) for telephony, Network as a Service (NaaS) for network optimizations and Database as a Service (DBaaS).

As is shown in Figure 2-2, the more flexibility that a service model provides a consumer with, the more control he has over the provided resources. For example, in the SaaS model, the user has little flexibility, as he is only capable of running specific applications. In the same time, he has the least possible control over his resources, being able to interact only with the environment of the application in use. In contrast, in the IaaS model, a user has full flexibility and control, as he can use his resources almost anyway he wants.

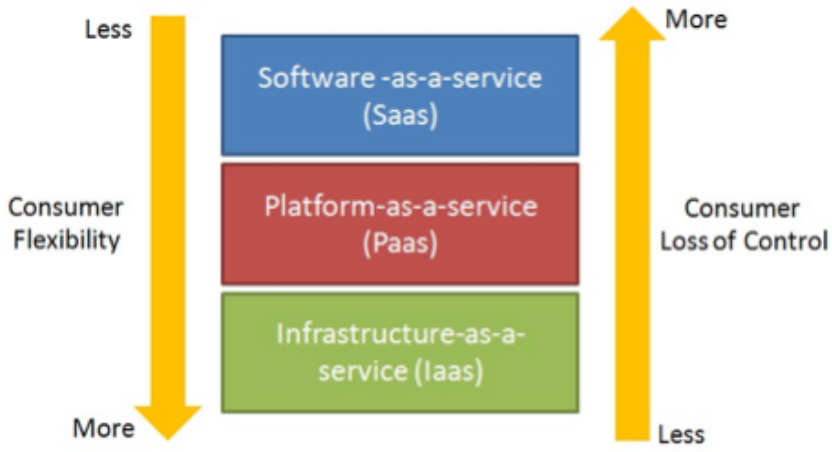

Figure 2-2: Consumer Flexibility and loss of control depending on the cloud service model

Our project addresses mostly the IaaS service model, as our main goal is to decide how resources will be distributed inside a cloud federation. In other words, we are dealing with Infrastructure. However, our system can also be viewed as a platform that provides tools to find the best-fitted resources for a deployment, so maybe a hybrid IaaS and PaaS model is the most suitable description.

2.1.4 Cloud deployment models

Finally, according to who operates the cloud infrastructure, there are different deployment models, as described by the NIST [1]:

- Private cloud

Private cloud is cloud infrastructure operated for exclusive use by a single organization with multiple consumers and it is owned and managed either internally or by a third party. Its location may be on or off the organization's premises.

- Community cloud

Infrastructure is shared by several organizations and supports a community that has shared interests. It may be managed by the organizations or a third party and usually exists on the premises.

- Public cloud

The cloud infrastructure is provisioned by a cloud service provider for open use by the general public over the Internet. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider.

- Hybrid cloud

This Cloud Computing model is a combination of two or more clouds (private, community, or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability.

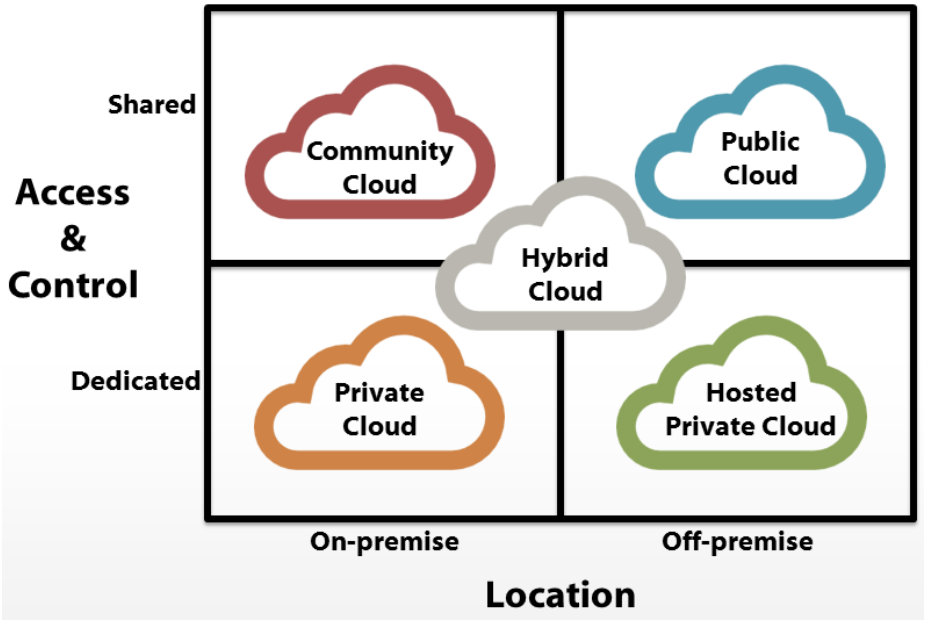

Figure 2-3: Deployment models distinction based on their characteristics

In Figure 2-3, we can see how the deployment models are differentiated according to the location of their infrastructure and their access and control. A private cloud, for example, has dedicated access and control, being administrated by a single organization, and depending on the location of its resources it may be hosted on or off the organization's premises.

Our system is developed to work with any cloud deployment model. Our current implementation is based on a public cloud (~okeanos (REFERENCE), but it is also designed to work in a Community cloud or a federation of public clouds. It also has the capability to be operated in private cloud environments, with only small modifications required.

2.2 Cloud Federations

Cloud federation is the result of interconnecting the cloud computing environments of two or more service providers for the purpose of load balancing traffic and accommodating spikes in demand [3].

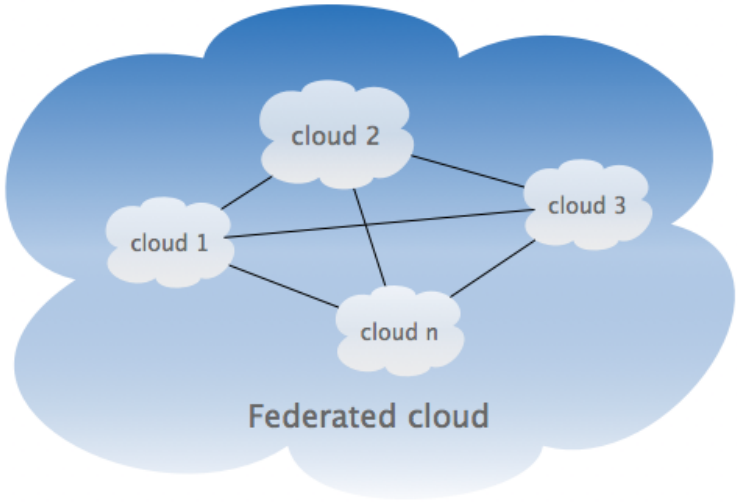

Figure 2-4: An example of a federated cloud, consisting of n clouds.

The concept of such a federation is that each single cloud does not have infinite physical resources or ubiquitous geographic footprint. Subsequently, if it saturates the computational and storage resources of its infrastructure, or is requested to use resources in a location where it has no footprint, it can still satisfy further user requests by providing resources from the infrastructure of other clouds of the same federation.

The cloud of clouds is an analogy to the Internet, which is a network of networks. Indeed, a service provider to which an endpoint is attached, will access or deliver traffic from/to source/destination addresses outside of its service area by using Internet routing protocols with other service providers with whom it has a pre-arranged exchange or peering relationship. Another analogy is the way mobile operators implement roaming and inter-carrier operability.

Cloud federations have many benefits for both the clients and the providers. Clients have virtually access to unlimited resources on demand and with full geographic coverage. Additionally, they are able to choose (when such choice exists) between providers offering the required services and resources, according to their criteria such as price, cloud type, performance demands or network preferences. Providers, on the other hand, are enabled to expand their geographic footprints and accommodate sudden demand spikes without having to invest in broader infrastructure. Moreover, they have the possibility to earn revenue from resources that would otherwise be idle or underutilized, while not compromising their quality of service (QoS) due to demand peaks [4].

2.3 Related work

The common practice in order to distribute resources inside a cloud federation is using a cloud broker. Below, we are going to describe what cloud brokering is and we will discuss some related papers and their approaches.

2.3.1 Cloud brokering

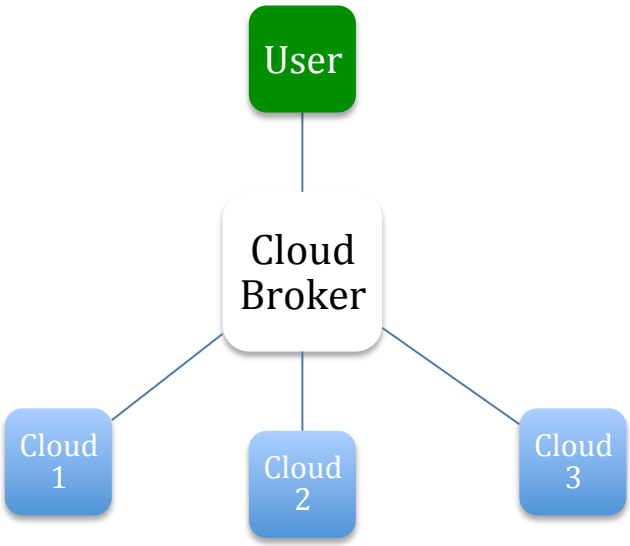

Figure 2-5: The cloud broker as an intermediate between the user and the federated clouds.

In a cloud federation environment, a user has many options of which to choose a provider of cloud services. Each one has different characteristics and very rarely can one be the most suitable in every case, even for the same client, as the client's requirements are not always the same. This choice becomes more difficult as the number of available clouds increases and it is no longer a simple task for the user to examine his options and choose the optimal. In fact, in most cases a user should not have to deal with this at all, meaning that the selection procedure should be transparent for him. Additionally, sometimes the optimal choice is related to factors that cannot be assessed by the user, such as the network topology or the resource constraints that a cloud may face due to a demand peak. Even for the providers, advertising their offer and comparing it to the others, which almost always is not even possible due to information that is not accessible to anyone outside the interested provider, is a job that should be done by someone else. Hence the need for cloud brokers.

The term cloud broker may refer to either a third-party business that acts as an intermediary between the client and the provider of a cloud computing service, or to a software application that automatically facilitates the distribution of work between different cloud service providers. In any case, the broker has access to all the relevant information of each cloud, such as its resource availability, its price scheme, its geographic location and its QoS and based on that and the user

demands, it can negotiate with cloud providers and propose the best solution or give the user an abstraction of all the info in order to let him make his own choice. After the choice is made, the broker contacts the provider and arranges the service [5].

In a cloud federation environment, the broker is a core entity of that federation, coordinating the distribution of resources of the common pool across its users. In addition to acting as an intermediary for the client-provider negotiations, the broker might also provide additional services facilitating the monitoring of all resources and the storage and safety of all user data. As each provider may use a different API, the broker must use a software adapter layer to deal with this.

Given the importance of the cloud broker, many papers have been published suggesting functionalities and principles of such brokers. The ones that we are going to discuss are "CloudCmp: Comparing Public Cloud Providers" [6], "A Coordinator for Scaling Elastic Applications Across Multiple Clouds" [7] and "Cloud Brokering as a Service" [8].

2.3.2 Related papers

The first paper [6] presents a comparison of existing providers in order to find a static way to propose the best fit according to the intended resource use. For the needs of it, various tests were performed on a set of commercial cloud services providers. Those tests included evaluation of the elastic compute cluster (CPU, memory and disk I/O), the persistent storage, the intra-cloud network and the wide-area network. The evaluation was based on running benchmarks and on the assessment of their finishing time, cost per benchmark and scaling latency.

The results produced, which were anonymous due to legal concerns, show that each cloud solution has its strengths and weaknesses. Taking the cost of each cloud into account, three case studies were deployed in order to find the most appropriate cloud for every case. These applications include a storage intensive e-commerce website, a computation intensive application for DNA alignment and a latency sensitive website, and each time the clouds' overall performance was different.

The brokering model presented in the second paper [7] embraces the concept of a cloud proposition (in this case selection) after a resource request. The main differentiating point is that the user only communicates with the cloud provider to which he belongs. If this cloud can provide the resources he demands, then it is the one that serves him. Otherwise, the cloud contacts the Cloud Exchange, which is the central component of this implementation, in order to place a request. Besides requests, the Cloud Exchange is also where the resource offers are made, and it is responsible for the start of negotiations between different clouds. Each cloud with spare resources sends such an offer to the Cloud Exchange, including the number of available VMs, their amount of memory and

number of cores, the computational capacity of each core and the hourly price of each machine.

After the Cloud Exchange finds a match for a request, he gets the two interested clouds in touch. These clouds now communicate through the Cloud Coordinator component of each one, which includes a negotiation engine. The Cloud Coordinator is also the component through which each cloud communicates with the Cloud Exchange. If the demands of the first cloud are not met, the offer of the second one may change in order to be accepted. When these demands are finally fulfilled, the resources of the second cloud are assigned to the user of the first.

Finally, paper [8] proposes a distributed cloud broker model, where each customer request is handled by a cloud agency that creates a dynamic broker specifically for this request. This request is called Call For Proposal (CFP) and contains both the list of resources to be acquired as an SLA (Service Level Agreement) Template defining the technical requirements for user's applications, and the broker policy to be enforced. The broker policy sets constraints and objectives on multiple parameters, such as the best price per time unit, the greatest number of cores, the best-accredited provider or the minimum accepted availability.

These constraints are divided into hard constraints and soft constraints. Hard constraints refer to the fact that the cloud offer must have the required condition; otherwise it is to be excluded. Soft constraints refer to desired requirements that can make a provider preferred with respect to another. For each parameter the user will eventually choose some constraints and define if they have to be hard or soft. Finally, the broker will use an objective function over all the proposals of the cloud providers that are matching to the CFP in order to find the most suitable.

Chapter 3

Architecture

The system that we developed proposes the cloud from which the resources demanded by the user should be deployed. This is achieved by examining the user requests and the information that is collected from the current infrastructure or its historic pattern. It can be designed to operate either per user or per cloud and we chose the per user approach.

In this Chapter, we are going to describe the architecture of our implementation, which consists of three main components:

- Collection of information
- Storage and organization of this information
- Data mining and decision-making.

Our goal was to make these components scalable, efficient and as platform-independent as possible. They can work either all together as a serial process or separately as stand-alone applications.

Figure 3-1: The three components of the implemented system.

3.1 Collection of data

This component collects information about the virtual machines of a cloud or a cloud site on a per-user basis. Its development is platform-dependent, since it needs to be integrated with the provided API of a specific cloud middleware in order to interact with the infrastructure and collect information.

In the case that the data is collected on a per user basis, a **Publisher** is responsible for collecting the information of all the VMs that belong to this user. This publisher can either be a special VM or it can be installed on a random VM of the user, as it generally presents very low resource consumption and does not affect the performance and load of a VM.

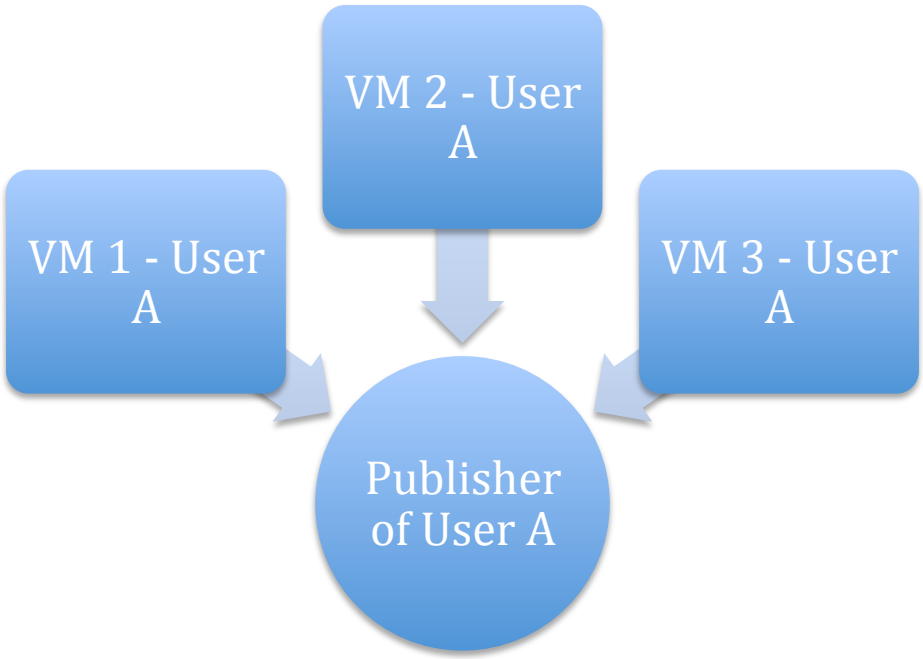

Figure 3-2: The publisher of User A gathers the information from all his VMs.

3.1.1 Connection to the cloud's identity manager

The Publisher connects directly to the cloud's identity management service for each cloud that the user has an account in. This service is responsible for all the user details including account info and quota limit. It also provides information about the user VMs (such as the unique ID of each one), so that the publisher can retrieve their stats and characteristics. The aforementioned connection requires that the API of this service is known and supported. In our case, the publisher is Openstack-compatible, which is an API well known for its compatibility, simplicity and reliability. A single publisher may support multiple APIs in order to be able to connect to all the clouds in which the user has an account.

The publisher requests all the desired information via method calls of the service's API that encapsulate HTTP requests to the appropriate server. This is repeated for every cloud that interests the user, in a repetition pattern that may be adjusted by either the user or the service provider. In most cases, the identity manager and the other servers will provide more information than what is needed, so the publisher is responsible for creating the records containing the appropriate data for each cloud.

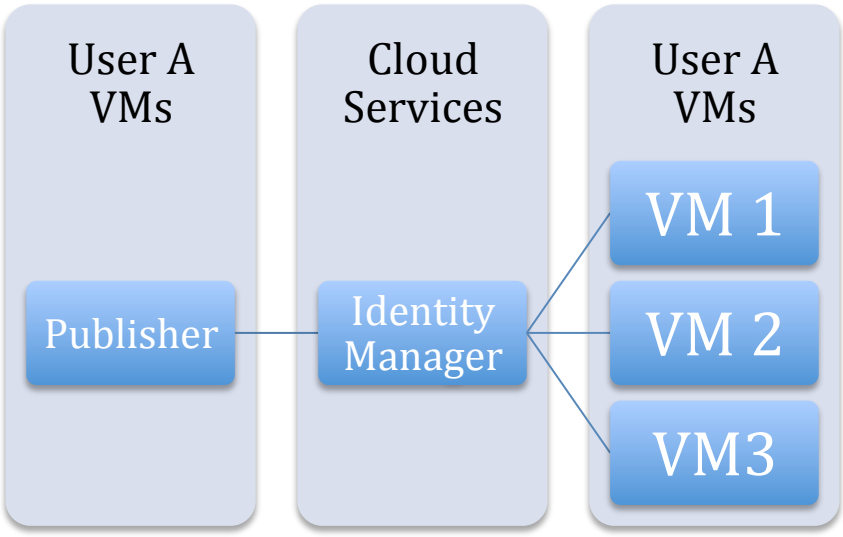

Figure 3-3: The publisher retrieves information from the Identity manager in order to gather data from the user's VMs.

So, as is pictured in Figure 3-3, the publisher, which is run in a user VM, requests information from the identity manager, which is a service provided by the cloud federation, in order to aggregate the required data of all the user VMs. This data is either provided directly from the identity manager's info (e.g. the quotas) or it can be retrieved from other sources thanks to this info. We will show in Chapter 4 how this is achieved.

3.1.2 Aggregation of data

Subsequently, the publisher creates one message file for each cloud containing a record for every VM that the user has deployed in that cloud. The key for each record is the VM's Universal Unique Identifier (**VMUUID**). The information per VM includes data concerning the user account in that cloud and the cloud itself, such as Local User ID, Local Group ID, cloud type, site name (cloud name), and the usage and limit of the user's resource quotas. Additionally, the VM configuration and statistic information are provided, such as CPU count and usage, memory, network type and speed and the image ID that operates the VM.

These files are stored in a queue, sorted by the time of their creation. Depending on the current policy, these files may be sent either instantly to the central database that accumulates all the data, or they may be kept and summarized by the publisher, who will later choose and send only some representatives ones. The sorting is very important both while storing and when sending the files, as possible inaccuracies can cause many inconsistencies to the service provided. So, in order to make them sortable, the files are named upon creation in a way that is easy for the queuing system to recognize.

In Figure 3-4 we can see an example of a VM record for a machine named "CentOs Apel Server" and with the VMUUID 213260, deployed in the cloud site "~okeanos" and belonging to the user with the ID 70c32715-cd13-4eaf-b923-d6a62952659f.

```
APEL-cloud-message: v0.2
VMUUID: 213260
SiteName: Okeanos
MachineName: CentOs Apel Server
LocalUserId: 70c32715-cd13-4eaf-b923-d6a62952659f
LocalGroupId: 70c32715-cd13-4eaf-b923-d6a62952659f
Status: ACTIVE
StartTime: 1405507839
WallDuration: 1037750
NetworkType: IP_LESS_ROUTED
NetworkInbound: 1
NetworkOutbound: 1
CpuCount: 1
Memory: 4096
CpuUsage: 3
Disk: 5
ImageId: 1e2a33c1-fdcb-4b44-8675-94de786770f5
CloudType: synnefo
DiskUsage: 45
DiskLimit: 150
VMUsage: 6
VMLimit: 7
PithosDiskspaceUsage: 3
PithosDiskspaceLimit: 100
RAMUsage: 9
RAMLimit: 26
CPUCoreUsage: 10
CPUCoreLimit: 12
FloatingIpUsage: 6
FloatingIpLimit: 6
PrivateNetworksUsage: 1
PrivateNetworksLimit: 9
PendingAppUsage: 0
PendingAppLimit: 1
%%
```

Figure 3-4: Record of the VM "CentOs Apel Server"

In Figure 3-5 we can see an example of 3 message files produced by the publisher of a user who has VMs in 3 clouds, more specifically 3 VMs in "Cloud 1", 1 VM in "Cloud 2" and 2 VMs in "Cloud 3". These files are independent and named differently, but are all sent to the next aggregation point with the same mechanism.

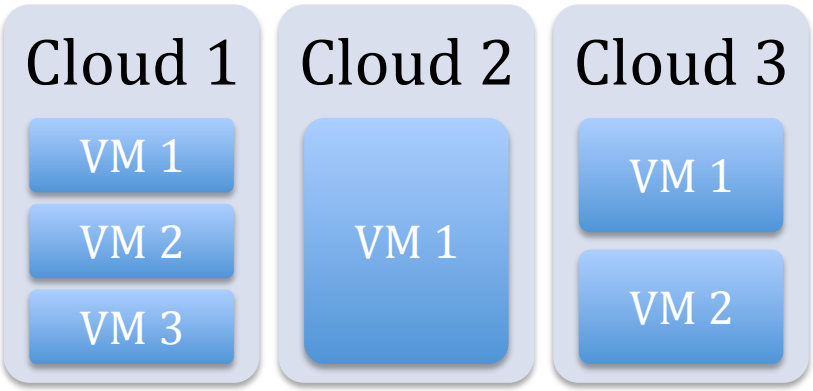

Figure 3-5: Files of the publisher, each one with VM records of a specific cloud

3.2 Storage and organization of data

All the publishers send the files with the data that they accumulated to one central database, which may either exist in a VM of the cloud federation, or it may be on a physical machine operated by the federation. For now, this machine will be referred to as the **Collector**. The data is then organized and imported into a MySQL database with an automatic procedure. The collector will be responsible for information collection, information processing and querying.

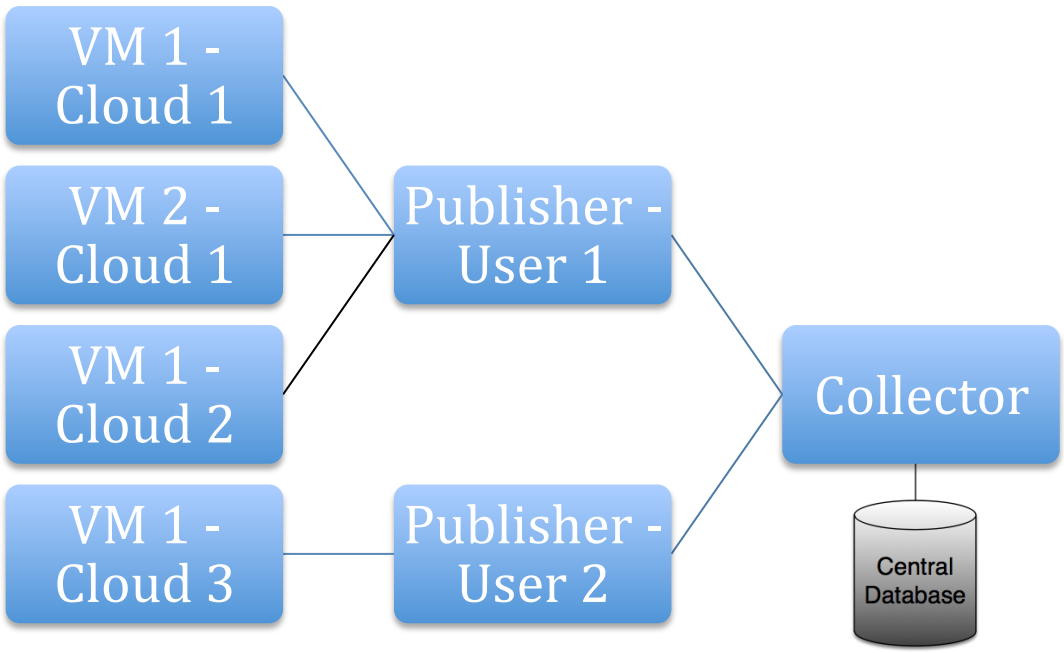

Figure 3-6: The publishers of 2 users who have VMs in 3 clouds send their data to the collector, which stores it in a central database.

3.2.1 Reception and storing of the data

All the files sent from the publishers are destined to a queue of the collector. As the collector uses the same queuing system with the publisher, it is ensured that there will be no temporal inconsistencies between the incoming messages.

In the collector, there are two running daemons responsible for the insertion of data in the database. The first one receives the message files sent from the publishers through the network and forwards them to the defined queue of the server, using protocols and frameworks that we will present in Chapter 4. It constantly checks for incoming connections and checks the authenticity of the sender in order to continue its task.

The second daemon is the database loader, which is responsible for the loading of the data into the database. It parses the raw text from the files and, after checking their consistency, inserts the data into the appropriate tables of the schema. The queue is checked and when new files are found the loading

application starts, determining which fields are to be added or updated in the database. Finally, the loader calls the MySQL procedures required to store that data to the suitable tables.

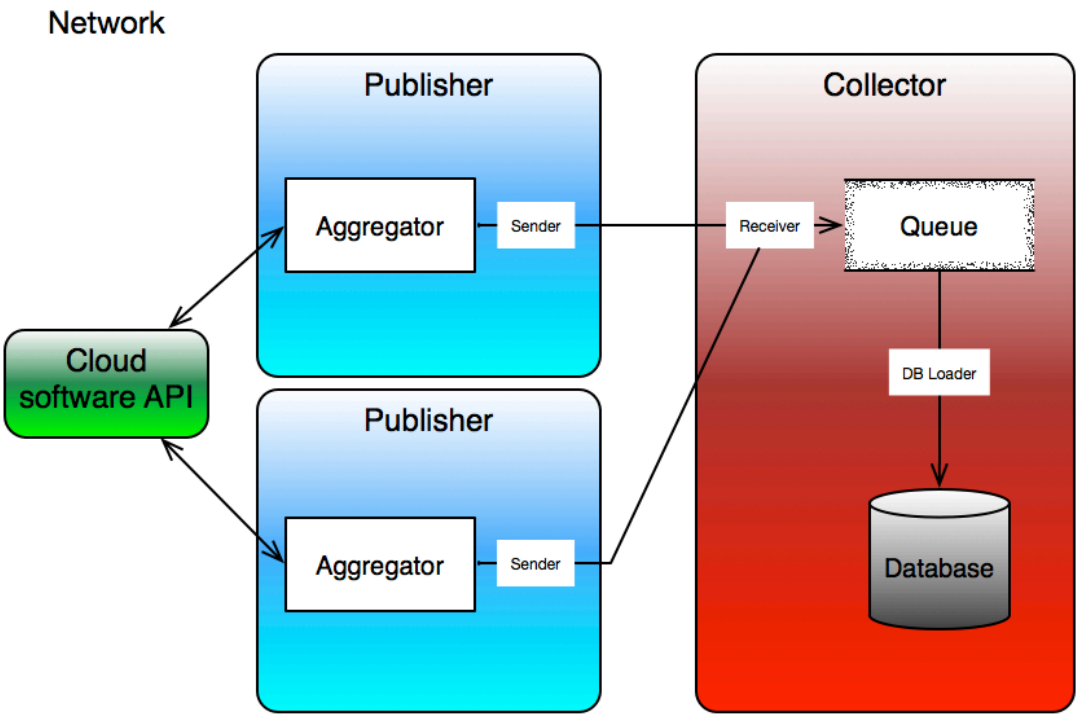

Figure 3-7: Aggregation, sending, receiving and storing the data

In Figure 3-7, we can see an overview of the main procedures so far. First, the publishers aggregate the data using the cloud software API, and then they send it to the collector who stores it in a central database.

3.2.2 The database schema

We will now describe the central database that we saw in Figure 3-7, where the collector stores all the information about the users and the VMs. The tables of this database that currently interest us are Cloud Records, Cloud Records History and Sites, while the only procedure that we use is Replace Cloud Records.

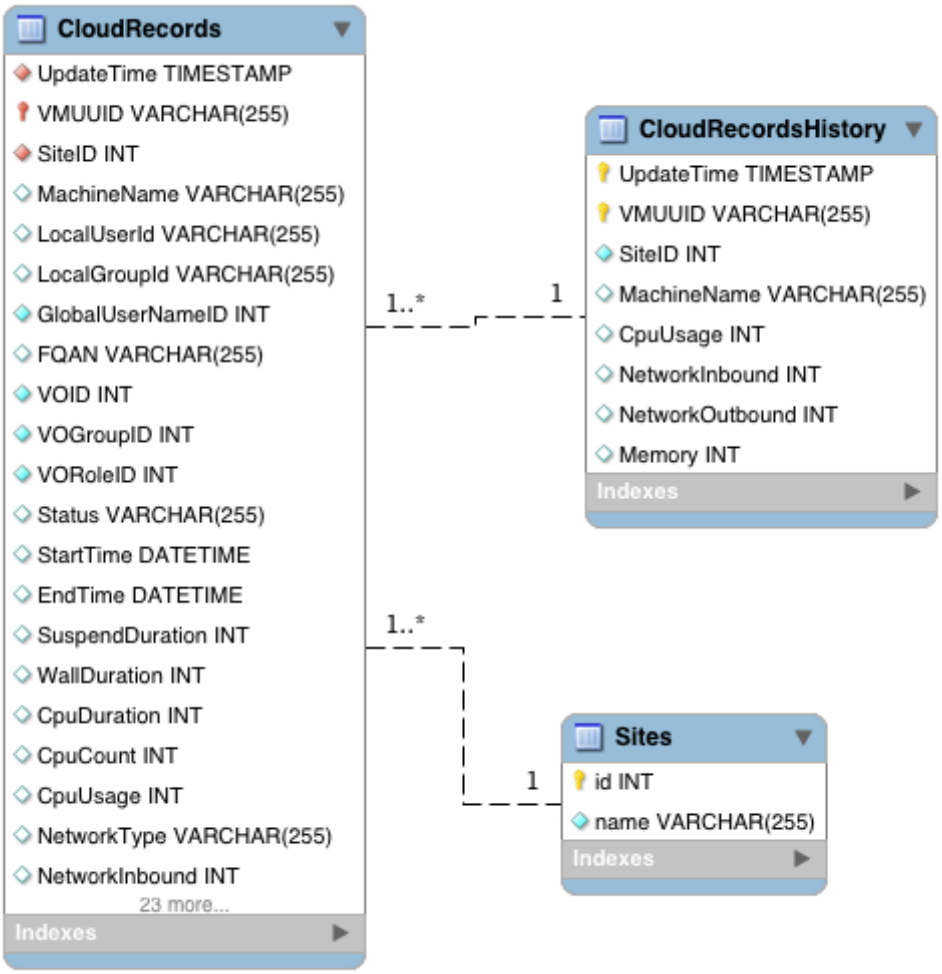

Figure 3-8: The database schema

"CloudRecords" is the table where the most current information is stored for each VM of the cloud federation. It is completely updated every time a new message arrives, as the most recent VM record replaces the old one. All the VMs that are active, building, restarting or stopped-shut down are included in this table. When a VM is deleted, its record is no longer updated but stays in the database with the last valid **Update Time**. It is up to the cloud administrator to decide how often the records of deleted VMs will be erased from the database. The table's primary key is the VM Universal Unique Identifier (**VMUUID**) and the fields **Update Time** and **Site ID** are foreign keys.

"Cloud Records History" is the table where a history is kept for specific metrics of the VMs. These metrics include CPU usage, Memory and Network Download / Upload speed. The primary key is the tuple of **VMUUID** and **Update Time**, as the most common use of this table is to find the values of these metrics of a VM at a certain time or during a time period. Additionally, **Site ID** is stored in order to be able to directly compare the history of different clouds (sites), and **Machine Name** is used with a semantic meaning, in order to be able to declare the intended use of the VM. Thus, different use cases will provide different historical

patterns and, when in need to create a new VM for a specific use, the performance of each cloud for this use can be taken into account.

In the table "**Sites"**, there is a record for every cloud or cloud site, assigning it to an ID. The reason of this decision is that using an integer is much more efficient when performing queries. Cloud sites will be used in many lists , structures and queries, as we will see in Chapter 4, so using strings each time would be meaninglessly aggravating.

Finally, the procedure "**Replace Cloud Records**" is the one called each time by the database loader. When a record for a new VM arrives, a new entry for that VM is added to the tables "**Cloud Records"** and "**Cloud Records History**". If it belongs to a cloud site that has not been reported before, a new entry is also added to the "**Sites**" table. If a record of a VM that already exists in the database arrives, its entry in "**Cloud Records"** is replaced and a new entry is added to "**Cloud Records History**", if the time that the stats were collected (declared by the collector) are different from last time's. This is pictured in the simple activity diagram of Figure 3-9.

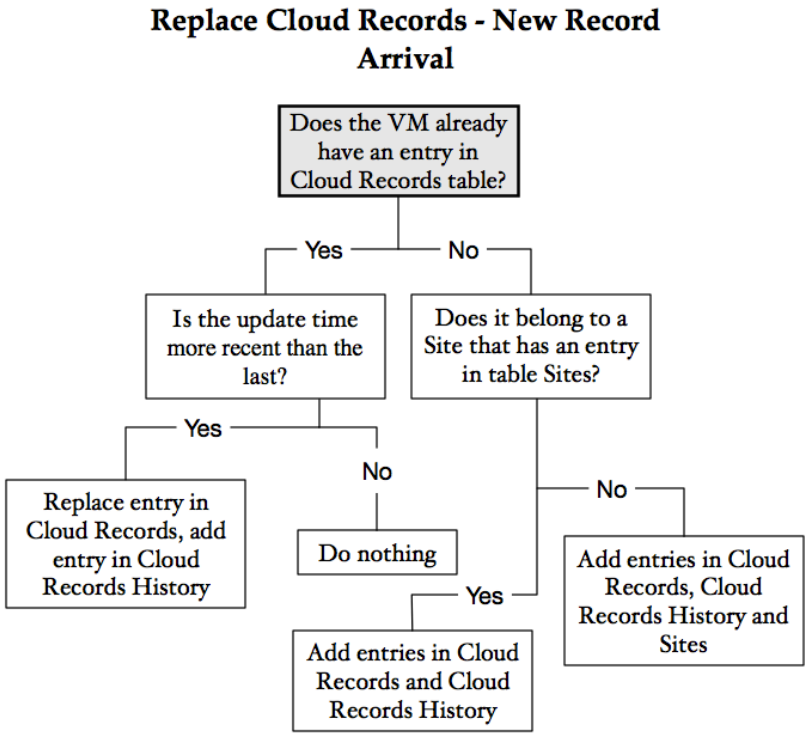

Figure 3-9: Activity diagram for the arrival of a new record at the procedure "Replace Cloud Records"

3.3 Data-mining and smart decision-making

After all the necessary data is collected in a central database, we must find a way to use it in order to make decisions about the deployment of future VMs. This is achieved by using **Sql**, a script that we created and is responsible for all the data mining and decision-making. In this way, both a user and a provider can query any information that they want in order to make their decision, or request an automatic provider proposition by the application, given their required specifications and intended use.

Figure 3-10 sums up the capabilities of our system. The publishers aggregate all the VM statistics from all clouds on a per user basis. Then this data is stored in the central database and the user (in our case) or the provider has the ability to query information and ask for an optimal cloud in order to deploy resources from.

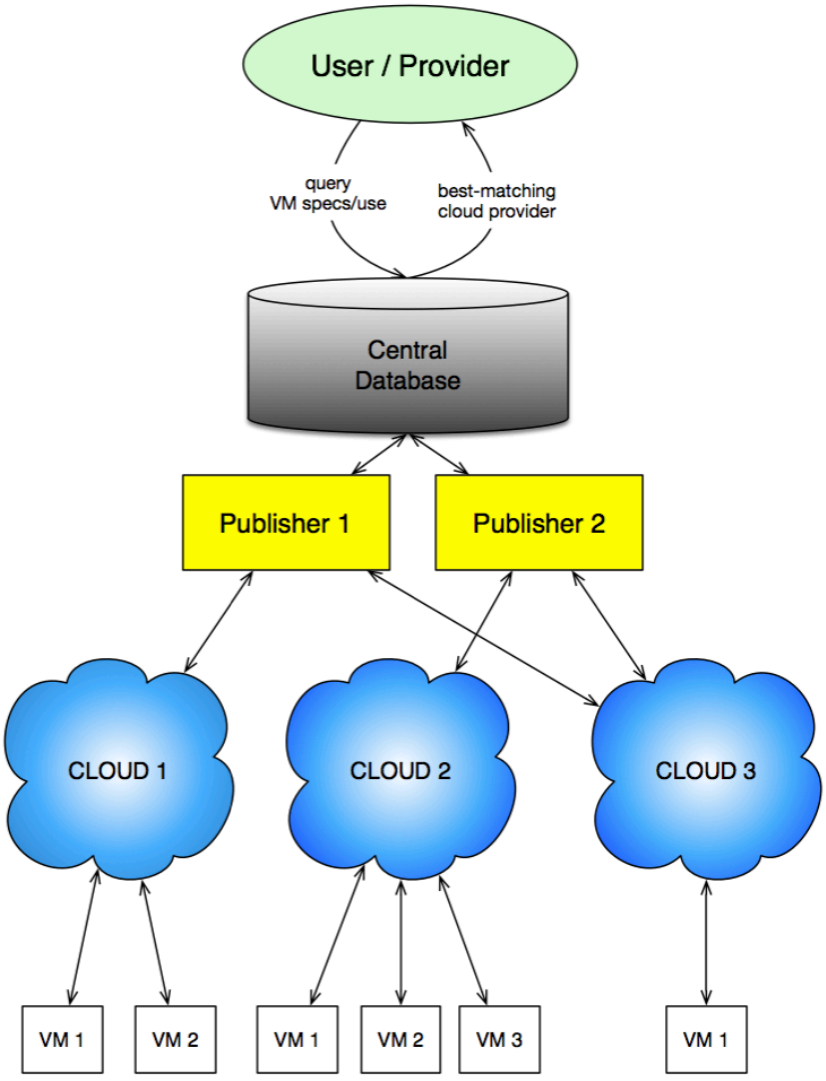

Figure 3-10: Overview of our system

3.3.1 Strategies for selecting the recommended clouds

In order to propose the most suitable cloud, our system uses certain strategies and the user can select which one will be used for the result.

1. Enough quota

The cloud site must be able to provide the resources requested from the user. Typically, a user is granted a limited access to cloud resources from a specific provider, especially in cases of free (educational) or otherwise confined providers. If the user's demands exceed his quota limit given his current usage - meaning the VMs that he has already deployed-, then the cloud is no longer a candidate.

2. Price

Besides the practical issue of quotas, price is always one of the most important factors in commercial environments. There can be different pricing models, most of which are proportionate to the resources provided but with possible details differentiating them. Thus, every time and for every cloud, the price is calculated taking into account the exact pricing policy and the requested resources.

3. Best match for the intended use

The user can choose whether his intended use is cpu-intensive, memory intensive or network intensive. Furthermore, he can create a custom scenario by providing his own preferences in a personalized model. In Chapter 4 we will see how these standard or custom scenarios have been implemented in order to make a final cloud proposition.

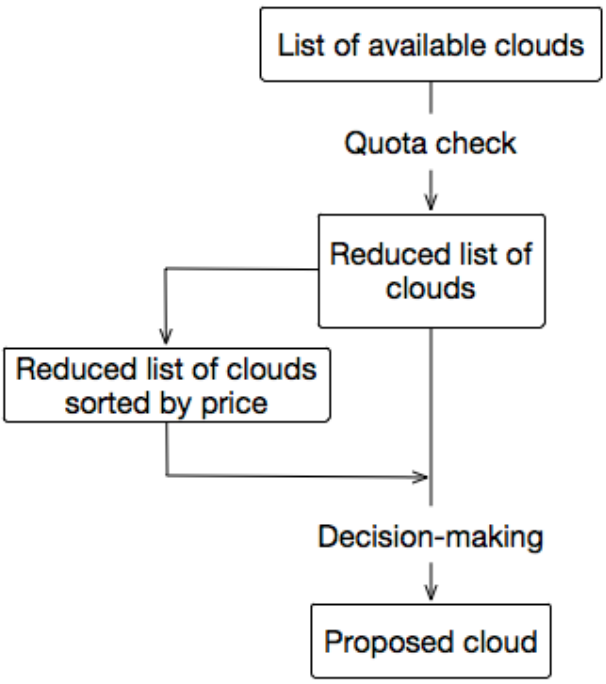

Figure 3-11: The strategies that lead to the selection of the proposed cloud

As it is shown in Figure 3-11, the decision-making process must start with the quota check. Optionally, the produced list of clouds can be sorted by price before the final proposition is made.

3.3.2 Application features

The query interface we developed offers a list of features presented in a command-line interface, trying to be as user-friendly, versatile and efficient as possible. Its fundamental functionalities are listed below, but it must be noted that the modification or addition of more capabilities is a straightforward task, due to the well-organized database in which the information is stored. During the application's initialization, a list of all the clouds belonging to the federation is created.

1. Quota check

   The user types in the resources that he demands and the application checks the availability in each cloud according to its registered resources. If a cloud is not able to provide any resource, it is subtracted from the active cloud list and the user is informed about its deficiency. By the time that all clouds are checked, the list includes only those that can offer the demanded resources without restrictions. Then, the price of each case is calculated and the list is sorted by this factor and presented to the user.

2. Specific cloud query

   This function provides the user with the capability to select a specific cloud from the cloud list and execute queries about it. These queries may relate to specification or performance issues. General information about the selected cloud can be obtained, or more complex requests can be answered, for example how many dual-core machines are currently active inside that cloud.

3. General queries about all clouds

   Using this function, the user can pose queries similar to the above, but concerning all the clouds in the list. In this way, he can compare specs and stats from different clouds, or he can filter the list by enquiring all the clouds fulfilling certain specifications. For example, he can demand a list of all clouds that have more than 10 VMs with CPU usage below 50%.

4. General queries about the clouds' history

   The user can retrieve information about the historical pattern of all the clouds. Queries regarding this feature include the retrieval of information such as the min, max and average of the metrics stored, and can also see the specs of the VMs that have produced those values. Additionally, he can demand the same information for the machines running a specific application type, such as a cpu-intensive one etc., so that the comparisons between different clouds are better defined and more specialized for his needs. As we

have already mentioned, the type of the application running on a VM (if any) is stated in its Machine Name. For example, a VM running network-intensive apps should be named "Net_Bench" in order to make such a capability feasible. This naming should be noted that it is optional. In this case, our system could use the "Net_Bench" term in order to query historical stats (e.g. the network inbound) of only those VMs that had explicitly stated that they were executing network-intensive tasks.

5. Cloud proposal

This is the function that automatically proposes a cloud provider to the user, taking his intended use into account. Currently, this is implemented by using an objective function, whose weights are defined by the user's preferences, in order to find the best match. The latter can either choose a standard use case or he can set the weights manually. The data on which the decision is based can either be the current values from the Cloud Records table, or it may be a combination of those values and the ones from the Cloud Records History table.

6. Advanced query

The purpose of this function is to give full potential to an expert user to make his custom MySQL queries directly to the database. He can just type the query manually and the results are returned with exactly the same way that they would if the MySQL application was used. A simple database schema is presented as help for the queries.

A user or a provider, in order to make queries and/or ask for a proposition of the most suitable cloud to deploy the requested resources from, can choose any of these actions, as it is shown in Figure 3-12.

As it can be observed, all the functions except for the Quota check are independent and can be called in any sequence unaffected. If they are executed before Quota check, then the produced results will include all the clouds of the federation, even those who do not have enough resources to match the user demands. Otherwise, the user may first execute Quota check and then proceed with the other functions. In this way, the produced results will concern only clouds that are able to offer the required resources and will be presented in a list of ascending price.

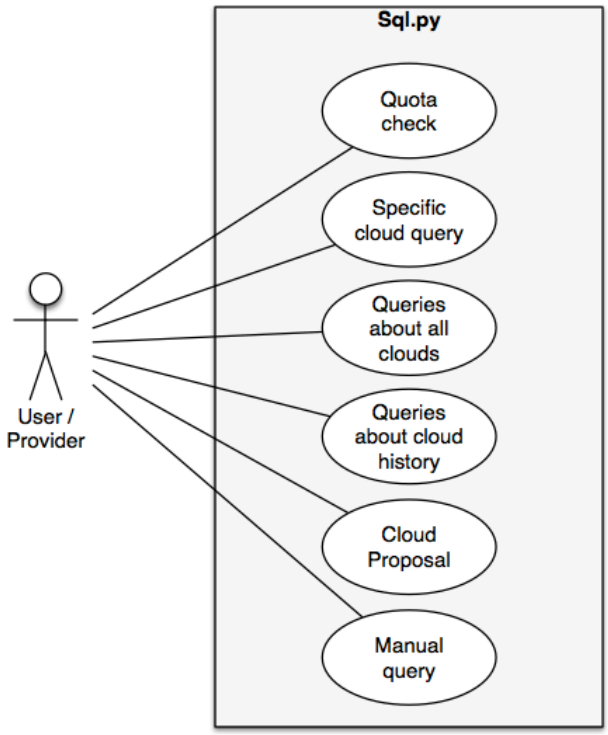

Figure 3-12: All the actions that a user or a provider can do using the Sql component

Chapter 4

Technical Details

In this Chapter, we are going to present the full technical details of our implementation. These details concern the cloud environment that we worked in (~okeanos), the application that gathers information from the clouds, the messaging frameworks to send the files from the publishers to the collector, the database schema and the application that is responsible for querying the collected data and proposing a cloud for deployment. Some of the code used was developed for the purposes of this project, while other parts were available from the Internet and were either modified or used unaltered.

The operating system of the machines that we used as publisher and collector was CentOS 6.5, as some of the already existing code that we used was at the time only available in rpm packages. However, even this code can be executed in other Linux OSes with the proper modifications, it just was easier for the purposes of this thesis to use CentOS. In our tests, we also used Debian and Ubuntu virtual machines for collecting stats.

4.1 ~okeanos

~okeanos is GRNET's cloud service for the Greek Research and Academic Community, providing cloud services such as virtual machines, networks and storage [9]. It is powered by Synnefo, a complete open source cloud stack written in Python that provides Compute, Network, Image, Volume and Storage services. Synnefo manages multiple Ganeti clusters at the backend and uses the OpenStack API at the frontend.

4.1.1 Components of ~okeanos

~okeanos is a collection of the components below:

- Identity Management (codename: astakos)
- Object Storage Service (codename: pithos+)
- Compute Service (codename: cyclades)

- Network Service (part of Cyclades)
- Image Registry (codename: plankton)
- Billing Service (codename: aquarium)
- Volume Storage Service (codename: archipelago)

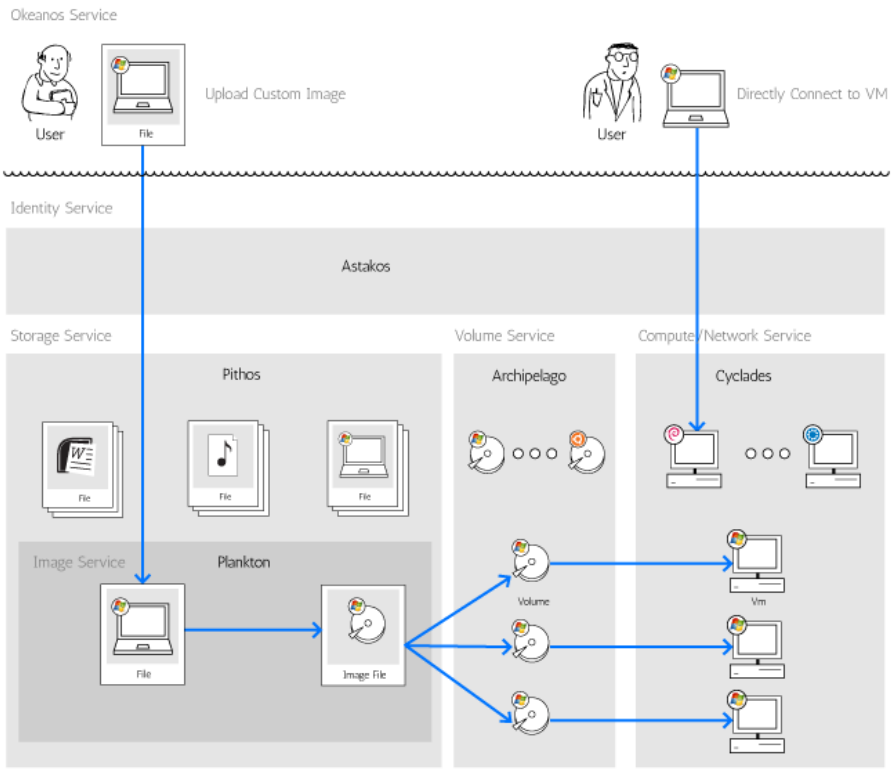

Figure 4-1: The components of ~okeanos service and their roles

The only user-visible ~okeanos services are Cyclades and Pithos. Cyclades is the Compute and Network part of ~okeanos. It provides access to the virtual machines that can be created, booted, shutdown or destroyed on demand, and to networking functionalities including firewalls, Internet access and virtual networks. Cyclades also keeps the statistics of the VMs concerning the compute and network resources that are used.

Pithos is the file storage service of ~okeanos, where everything can be stored, from user documents to custom Images for the creation of new VMs. So, it can be viewed either as a stand-alone service, accessible via a web browser, command-line or a native client, or as an integral part of ~okeanos.

Astakos, the identity management service, provides the single point of authentication and authorization for Cyclades and Pithos. Via Astakos, ~okeanos users can register, login and manage their account tokens.

Plankton is the Image Registry service, implemented on top of Pithos. Every Image of Plankton is a file stored on the Pithos backend and users can synchronize or upload their Images to Pithos, then register them with Plankton.

Aquarium is the accounting and billing service of ~okeanos, which is currently under development. It monitors the resources used by each user and matches them to credits.

Finally, Archipelago is the volume storage service, a custom storage layer that handles volumes as set of distinct blocks in the backend. [10]

4.1.2 Components of Synnefo

Synnefo has a completely layered architecture, at the lowest level of which every VM operates using KVM, an open-source full virtualization solution that supports virtual machines running Linux and Windows images. Those VMs are organized in clusters, the management of which is assigned to Ganeti nodes. Ganeti is a virtual machine cluster management tool developed by Google, which was chosen thanks to its scalability and reliability as a software infrastructure for managing VMs. [11]

Synnefo runs on top of Ganeti, having the components that we mentioned before, such as Cyclades, Astakos, Pithos etc. There is a strong bond between ~okeanos and Synnefo components, as the latter was mainly developed for the needs of the former.

At the front end, Synnefo uses a superset of the Openstack API, achieving this way compatibility and simplicity. One of the tools of Synnefo for the management of a deployment is Kamaki.

This transition between the layers of Synnefo is portrayed in Figure 4-2.

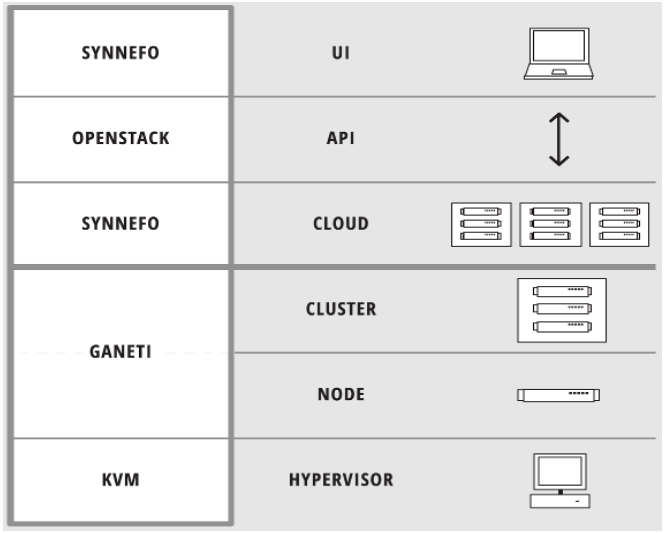

Figure 4-2: Layers of the Synnefo software

In Figure 4-3 we can see a more detailed layout of the Synnefo architecture. A user can access Synnefo either by command-line (Kamaki) or by the web user interface from the official site. In both cases, the communication with the lower level is achieved by using the Openstack API. The cloud level's architecture is the one of ~okeanos, the components of which we have already described. Below, there are the cluster, node and hypervisor levels, accessible only by the administrators, which are operated by Ganeti and KVM. The storing service uses Rados (reliable autonomic distributed object store), a free object-based storage system provided by Ceph. [12]

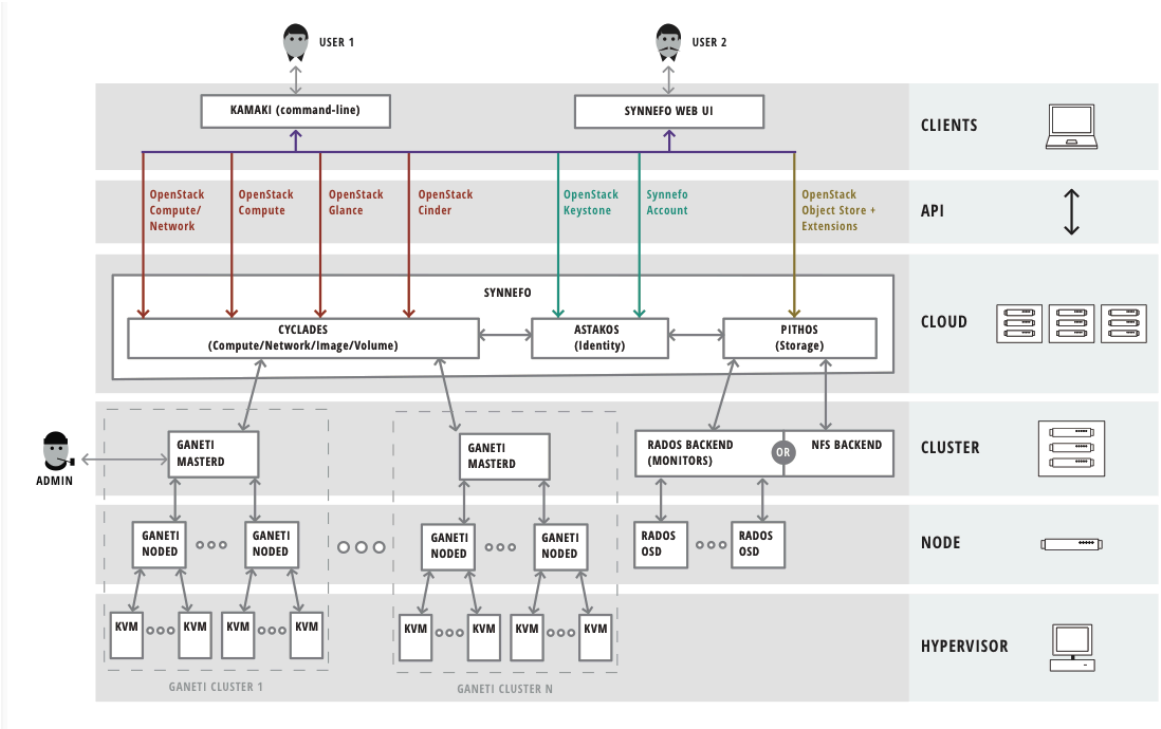

Figure 4-3: Detailed layout of Synnefo architecture

4.1.3 Kamaki

Kamaki is an interactive command-line tool and a client development API for managing clouds, which implements OpenStack together with other custom extensions. Kamaki has many uses, including testing of the Synnefo software, operating the ~okeanos services by the deployment team of GRNET, being the command-line tool for the Pithos client, testing and debugging of software developed by third-party Synnefo deployers and as an API for other Synnefo components or external applications. [13]

Kamaki has a long list of commands, the most important of which are listed below:

- User: info, authenticate, add, delete are some of the capabilities of this command, which is used to access or modify attributes of an existing ~okeanos user (or create a new one).
- Quota: request the user quotas.
- Flavor: flavor corresponds to the list of specifications of a VM, such as the number of CPU cores, the ram and the hard disk size.
- Image: list, get info about, (un) register and modify images accessible by the user.
- Server: the most important command, it gives the capability to the user to list, get info about, create, modify, start – shutdown – reboot, delete or use by console the VM with the provided VMUUID.
- Network: manage the network capabilities of ~okeanos.
- File: manage files and images stored in the Pithos service.

We will use Kamaki as a client development API, as the part of our project that gathers data and stats from the VMs uses it to request information from ~okeanos. This API is responsible for and makes the appropriate HTTP requests to the ~okeanos server. Most of the methods used are paired one to one with shell commands of the command-line application of Kamaki.

At first, the application run by the publisher uses Kamaki to connect to the identity manager (Astakos), who will provide information about the user account, his quotas and his VMs. Then, using the information about these VMs, the publisher will connect to the compute and network services of Cyclades, in order to retrieve all the data required. In the following Section, we will see how the publisher connects to those servers, which API calls it uses and how this information is returned and processed.

4.2 Collecting and sending the information

This is the first stage of the operation of our system, where all the required information about each user and his VMs is collected from the ~okeanos servers or the VMs directly. As we explained in Chapter 3, the component responsible for this task is the publisher. The software that implements the publisher is a python script developed entirely for the needs of our project and is named publisher.py. It is a platform-dependent script, since it needs to be integrated with the provided API of Synnefo in order to interact with the ~okeanos servers and retrieve the required information.

As we have already mentioned, the information that we collect is on a per-user basis, and this script is developed to work with the Synnefo software. Consequently, the publisher.py script is responsible for acquiring information from all the user's VMs in the cloud ~okeanos.

4.2.1 Retrieving information from the servers of ~okeanos

The first thing to do in publisher.py is to import the Kamaki packages that we are going to need. Next, we have to load the user credentials that will be demanded by Astakos (the identity manager of ~okeanos). These credentials are the authentication URL of Synnefo and the token, which is unique for each user, and they are loaded by parsing the configuration file .kamakirc, which is stored in the user directory. The indicative code is shown in Figure 4-4.

```
from kamaki.clients.astakos import AstakosClient
from kamaki.clients.compute import ComputeClient
from kamaki.clients.network import NetworkClient


config = ConfigParser.ConfigParser()
config.read("/Users/Apostolis/.kamakirc")
AUTHENTICATION_URL = config.get('cloud "mycloud"', 'url')
TOKEN = config.get('cloud "mycloud"', 'token')
```

Figure 4-4: Imports and credentials

Then, we need to set up a client instance for every service that we are going to use, in our case the compute, network and astakos clients, which will respectively provide information about the VM capabilities and stats (except for network), the network capabilities and stats and the user attributes and quotas. This is achieved by retrieving the endpoints of each service. The astakos service's endpoint is the authentication URL and the endpoints of the other two services are requested from astakos, as shown in Figure 4-5.

```
user = AstakosClient(AUTHENTICATION_URL, TOKEN)
cyclades_endpoints = user.get_service_endpoints('compute')
network_endpoints = user.get_service_endpoints('network')
CYCLADES_URL = cyclades_endpoints['publicURL']
NETWORK_URL = network_endpoints['publicURL']
compute = ComputeClient(CYCLADES_URL, TOKEN)
network = NetworkClient(NETWORK_URL, TOKEN)
```

Figure 4-5: Setting up the clients

The next thing to do is iterate over all the VMs of this user. Now that the clients have been initialized, we can use their methods that correspond to the Kamaki commands. For example the following method returns a dictionary containing records for every VM and its **VMUUID**.

```
servers = compute.list_servers()
```

Using this dictionary, we create a list containing only the VMUUIDs of all the VMs. This is the list that will drive our iteration over all the user's VMs. We then create a list of dictionaries, **extracts**, which consists of one extract for each VM. Each extract has fields for every bit of information that we want to collect from the VMs and for their quotas. These fields are:

- VMUUID
- SiteName: the name of the cloud or the cloud site
- MachineName: the name of the VM
- LocalUserId: the user's ID in ~okeanos
- LocalGroupId: the ID of the group of ~okeanos to which the user belongs
- Status: the status of the VM, e.g. Active, Deleted etc.
- StartTime: time of the last boot of the VM
- WallDuration: the time that the VM has been active (if its status is active)
- NetworkInbound: network's download speed
- NetworkOutbound: network's upload speed
- CpuCount: number of CPUs of the VM
- Memory: the size of the VM's RAM
- CpuUsage: the VM's current CPU usage
- Disk: the size of the VM's hard disk
- ImageId: the ID of the image of the VM
- CloudType: the software that runs on the cloud infrastructure (in our case Synnefo)
- DiskUsage: the total of the currently used GBs by the user as VM disk space
- DiskLimit: the quota of disk space of the user
- VMUsage: the total of the VMs of the user
- VMLimit: the quota of VMs
- PithosDiskspaceUsage: the total of GBs used for Pithos service
- PithosDiskspaceLimit: the quota of Pithos disk space
- RAMUsage: the total GBs of the RAM of the user's VMs
- RAMLimit: the quota of RAM
- CPUCoreUsage: the total number of CPU cores deployed by the user
- CPUCoreLimit: the quota of CPU cores
- FloatingIpUsage: the total number of IP addresses used
- FloatingIpLimit: the quota of IP addresses
- PrivateNetworksUsage: the total number of private networks used
- PrivateNetworksLimit: the quota of private networks

Now that we have the extracts, we just need to fill them with the information that we will retrieve from the servers of ~okeanos.

Inside the loop that iterates for every VMUUID in the list that we previously created, we call the method shown below in order to get all the details for the VM (server) with the current id.

```
server = compute.get_server_details(id)
```

Now all the information we need for this VM, except for the network, is stored in the dictionary **server**. In order to place the pieces of that information that we want in the dictionary **extract**, we use commands like in the following example, which stores the **MachineName** in our desired dictionary.

```
extract['MachineName'] = server['name']
```

In order to fill all the fields of the **extract** we need to do the same thing for the respective fields of **server**. Accordingly, in order to get the network information that we want we do the following

```
extract['NetworkType'] =
(network.get_network_details(server['attachments']
[0]['network_id']))['type']
```

Next, with the following commands we retrieve the user's quotas and store them in the dictionary **extracts**.

```
r_quotas = user.get_quotas()
```

```
extract['DiskUsage']                                    =
r_quotas['system']['cyclades.disk']['usage']/1073741824
```

The last command stores the total of the currently used GBs by the user as VM disk space in the field 'DiskUsage' of our dictionary **extract**. In the same way we store the other fields for the rest of the quotas.

Finally, at the end of our iteration per VM, we store the dictionary **extract** in the list of dictionaries of all our VMs, **extracts**.

```
extracts[id] = extract
```

4.2.2 Preparing the message file

At this point, we have a list of dictionaries, one for every VM that the user owns in this particular cloud. The message file is plain text, so we need to transform all the information from the fields of these dictionaries in plain text. In order to form this text, we have to follow the standards required by our messaging system.

The messaging system that we use is the Secure Stomp Messenger (**SSM**), which is designed to send messages using the STOMP protocol. It was developed as a way of using python and STOMP to securely and reliably send messages from APEL clients to the APEL server [14]. **APEL** is an accounting tool designed by the European Grid Infrastructure (**EGI**) in order to collect accounting data from sites participating in EGI infrastructures [15]. SSM and APEL are open-source, so we are going to use some of their code in our system. Finally, APEL/SSM Openstack (**OSSSM**) is a system that extracts usage records of monitored tenants and forwards them to APEL/SSM accounting system, interacting with OpenStack [16].

This means that **OSSSM** has the same functionality as our **Publisher**, but it is developed to interact with the OpenStack API, which has many similarities but also a few differences with the Kamaki API of Synnefo. Since OSSSM uses the same messaging system as we do, we will use one of its configuration files, **osssmrc**, which includes all the necessary information about forming our message file according to the requirements of our messaging system.

```
conf = ConfigParser.ConfigParser()
conf.read('/etc/osssmrc')
config = {}
for item in (
    'ssm_input_header',
    'ssm_input_sep',
    'ssm_input_path'
    ):
    config[item] = conf.get( 'Main', item )
```

Figure 4-6: Importing message elements

Figure 4-6 shows how we parse the required elements for the message file. These elements are the input header, which is the header of the message file as required by the messaging system SSM, the input separator, which separates the records of each VM and the input path, which specifies the directory where the file will be saved.

Next, we create the message file, starting with the header. For every extract (VM), we add its fields to the file in the form of one line of plain text for each field. When we finish with the fields of this extract/VM, we add the input separator and we follow the same procedure for the next extract/VM until every one has been recorded to the message file.

In Figure 4-7, we can see a message file with only one record of VM (one extract). The input header is "APEL-cloud-message: v0.2", followed by all the fields and their values and the input separator is "%%". If there were more VMs in this file, then the line right after the input separator would Figure the VMUUID of the next VM, followed by the rest of its fields and so on.

```
APEL-cloud-message: v0.2
VMUUID: 213260
SiteName: Okeanos
MachineName: CentOs Apel Server
LocalUserId: 70c32715-cd13-4eaf-b923-d6a62952659f
LocalGroupId: 70c32715-cd13-4eaf-b923-d6a62952659f
Status: ACTIVE
StartTime: 1405507839
WallDuration: 1037750
NetworkType: IP_LESS_ROUTED
NetworkInbound: 1
NetworkOutbound: 1
CpuCount: 1
Memory: 4096
CpuUsage: 3
Disk: 5
ImageId: 1e2a33c1-fdcb-4b44-8675-94de786770f5
CloudType: synnefo
DiskUsage: 45
DiskLimit: 150
VMUsage: 6
VMLimit: 7
PithosDiskspaceUsage: 3
PithosDiskspaceLimit: 100
RAMUsage: 9
RAMLimit: 26
CPUCoreUsage: 10
CPUCoreLimit: 12
FloatingIpUsage: 6
FloatingIpLimit: 6
PrivateNetworksUsage: 1
PrivateNetworksLimit: 9
PendingAppUsage: 0
PendingAppLimit: 1
%%
```

Figure 4-7: A message file with one record

Finally, we add this file to a queue in the directory that is specified by the input path. This is achieved by using Dirq. The goal of this module is to offer a simple queue system using the underlying filesystem for storage, security and to prevent race conditions via atomic operations, focusing on simplicity, robustness and scalability [17]. As Dirq is used throughout our messaging system, it ensures that the message files will be stored, sent and received with correct order based on the time of their creation.

4.2.3 Sending the message file

So far, the publisher collects the required information from the servers of ~okeanos and saves it in the form of message files in a queue. In order to send it to the Collector, we use the SSM messaging system. In the configuration file of the SSM sender (sender.cfg), we define that outgoing messages will be read and removed from the directory of the queue. We also define the host name, port and queue of the destination as well as the certificate that will be used, as SSM supports signing and encryption of the messages during transit. In our use case, however, we have deactivated the encryption, as we only had test purposes.

When all is set, executing the command **ssmsend** will start the SSM messenger, which will look for any message files in our queue. If none is found, nothing more happens. Otherwise, the SSM sends the file(s) to the appropriate destination, and then terminates. As we will explain later in Chapter 5, in working conditions the publisher.py and the ssmsend should be included in a single cron task that, according to our settings, will be scheduled and executed in order to retrieve, aggregate and send the VM and user information to the Collector, where it will be stored in a central database.

4.3 Receiving and storing the information

The destination of the message files of all the publishers is the collector. After receiving all these files, the collector has to store their data in a central database. The reception and storing of this data is an automated procedure, based on two daemons that are constantly waiting for new data to arrive.

The first daemon is a part of the messaging system that we use, SSM. In order to store the incoming data, our collector uses the APEL accounting tool, taking advantage of its ease of use, stability and compatibility with the SSM. Therefore, the second daemon is part of the APEL system.

4.3.1 Receiving the message files

In order to be able to receive the message files that the publishers send, the collector needs a message broker. The message broker is an intermediary program for message validation, transformation and routing, minimizing the mutual awareness that applications should have of each other in order to be able to exchange messages. We chose the Apache ActiveMQ open source message broker, which is written in Java, as it is a certified and stable solution. So, ActiveMQ should be always running in the background, waiting for new messages to receive.

Having the message broker running, we can execute the **ssmreceive** command, which will start the **ssmreceive** daemon. This is the daemon of the SSM system, responsible for the reception of all the messages sent from the publishers using **ssmsend**. In the configuration file of the SSM receiver (receiver.cfg), we define the queue destination and port to which the SSM will listen (which are the same as those defined in sender.cfg), the certificate and the directory to which the accepted messages will be saved. When the receiver receives one or more messages, ssmreceive first validates the sender. If the sender is accepted, then the messages are stored in the specified path (/var/spool/apel/incoming), from where they have to be loaded so that their information is stored in the central database of the collector.

4.3.2 Storing the incoming information in the central database

The database schema is defined by the file cloud.sql, which belongs to the APEL files and we use it with some modifications. As we described in Chapter 3, the tables of the schema that we use are three, “CloudRecords”, “CloudRecords History” and “Sites”, while the only involved procedure is the “ReplaceCloud Records”.

The first table keeps the latest information about all the VMs. It keeps all the fields that the publishers’ records contain, which we mentioned in Chapter 4.2.1. The only difference is in the filed “SiteName”, as in “CloudRecords” it is replaced by “SiteID”. As we have already explained, the APEL system has a separate table, “Sites”, for the relation between site names and their ID for reasons of maintenance and speed for the SQL queries, and we had no reason to change this. Additionally, as the original database scheme did not include fields for the user quotas, we had to add those in the cloud.sql file.

The APEL database scheme did not include at all a table for keeping historical data of the VMs, so we had to create “CloudRecordsHistory”. As we can see in Figure 4-8 below, the records of this table consist of UpdateTime, VMUUID, SiteID, MachineName, CpuUsage, NetworkInbound/Outbound and Memory. The tuple of UpdateTime and VMMUID constitute the primary key of the table, as every VM has multiple records for different time periods. SiteID and

MachineName are used so that more complicated queries can be executed, and the other fields are the stats of which we want to keep the historical pattern.

```
CREATE TABLE CloudRecordsHistory (
  UpdateTime TIMESTAMP,
  VMUUID VARCHAR(255) NOT NULL,
  SiteID INT NOT NULL,                -- Foreign key
  MachineName VARCHAR(255),
  CpuUsage INT,
  NetworkInbound INT,
  NetworkOutbound INT,
  Memory INT,

  PRIMARY KEY (VMUUID, UpdateTime)
);
```

Figure 4-8: The table "CloudRecordsHistory"

Having all the message files saved in the specified directory, we need a tool to load them into the database. This tool will be a part of the APEL system, the script **apeldbloader** that will be executed and then run as a daemon. This loader checks the directory and finds the new files. If their text is in accordance with the SSM standards, then it stores the data in the database and moves the message files to the directory /var/spool/apel/accept. The loader checks that the number, the field names and the value types are correct, and then it calls the procedure "ReplaceCloudRecords" from the database schema.

The version of this procedure that is included in the APEL package is only responsible of replacing the records of the table "CloudRecords" with the new ones, included in the newly arrived message files. So, in order to add support for our history table, we had to inject the code that is shown in Figure 4-9 into "ReplaceCloudRecords".

```
IF status <> "DELETED" THEN
    REPLACE INTO CloudRecordsHistory(VMUUID, SiteID, MachineName, CpuUsage,
                        NetworkInbound, NetworkOutbound, Memory)
        VALUES (
            VMUUID, SiteLookup(site), MachineName, CpuUsage, NetworkInbound,
                        NetworkOutbound, Memory    );
END IF;
```

Figure 4-9: Added functionality for "CloudRecordsHistory"

This extension first checks that the VM whose record is under inspection is not deleted. This is mandatory as when an ~okeanos VM is deleted, for a short period of time it will still exist in the user's list of VMs in the servers, but its stats should not be added as a new set of data in our history table, as it will be a duplicate of its last values. Next, it uses the **REPLACE** function of MySQL that if there is not already a record with the provided primary key (and there will not

be, as the UpdateTime field keeps changing), a new record will be added. The same function is used by the procedure for the "CloudRecords" table. In that case, however, the primary key is just the VMUUID, so when a new record for a VM that already has been recorded arrives, we only have an update of the fields with the new values, instead of the addition of a new record for the same VM.

Once the database with our modified schema is set up and the ActiveMQ broker and our two daemons, **ssmreceive** and **apeldbloader** are active, no action is required by the user or administrator in order to have the information of the incoming message files stored in the database. Once new messages arrive, our system automatically updates the "CloudRecords" table and augments the "CloudRecordsHistory" table.

4.4 Data mining and decision-making

When our system is up-and-running, the incoming information from all the publishers is automatically stored in the central database of the collector. So now we need a program to query that information and to automatically suggest the best-fitted resources for the user requirements. We achieve this by using the **Sql** script (sql.py) that we developed, which is written in python and is accountable for both tasks.

The script begins by connecting to the database, after it has parsed the configuration info that is needed (username, password, hostname, database name). This can be seen in the following Figure.

```
config = ConfigParser.ConfigParser()
config.read("/etc/apel/db.cfg")
hostname = config.get('db', 'hostname')
username = config.get('db', 'username')
password = config.get('db', 'password')
name = config.get('db', 'name')

db = MySQLdb.connect(hostname,username,password,name)
cursor = db.cursor()
```

Figure 4-10: Connecting to the database

Next, we construct a list named **clouds**, which will be the basic element of this script, used in all the iterations, checks and listings of different clouds or cloud sites. **Clouds** can be filled either statically, by modifying the code of sql.py, or can be input by the user during the execution time of the script (Figure 4-11).

```
item = str(raw_input('Type the name of the first cloud:\n'))
while item != '' :
    clouds.append(item)
    item = str(raw_input('Type the name of the next cloud:\n'))
```

Figure 4-11: Filling the list 'clouds'

The first use of this list is to create a new list of dictionaries named **price**, consisting of one dictionary for each cloud, containing its name and price. The price can be set statically, or a calculating function can be inserted in the code of sql.py. The user is, then, given the option to choose his next step, which can either be quota check, cloud querying, cloud proposal or advanced querying, where the user types his own MySQL query. In the last case, the user is given some information about the database scheme that will help him form correctly the MySQL query and then is asked to type it in. The description of the first three options follows.

4.4.1 Quota check

After choosing the quota check, the user is asked to provide the amount of resources that he requires, that is the number of VMs, the disk capacity, RAM size, number of cpu cores, number of floating IPs and number of private networks. These values will be saved in variables that will be compared to the available resources of every cloud in the list **clouds**.

```
cursor.execute('select VMUsage from CloudRecords where SiteID = ' + str(cloud_id) + ' order by UpdateTime DESC')
x1 = cursor.fetchone()[0]
cursor.execute('select VMLimit from CloudRecords where SiteID = ' + str(cloud_id) + ' order by UpdateTime DESC')
x2 = cursor.fetchone()[0]
if (r_VM > (x2-x1)) :
    print 'Not enough resources in cloud %s : number of VMs (available %d) ' %(item, x2-x1)
    capable = False
```

Figure 4-12: Quota check

The Figure 4-12 displays the part of the iteration over the members of the list **clouds** where the requested number of VMs (r_VM) is compared to the difference between the given cloud's VMUsage and VMLimit quotas. If the requirements are met, then nothing happens and the cloud remains in the list. Otherwise, a message is printed that explains the deficiency and the cloud is labeled as incapable, thus is later eliminated from the cloud list. The SQL query is formed using the SiteID of each cloud (as cloud_id) and returns values from the most recent record of that cloud, as we have mentioned that records of deleted VMs may remain for a limited time in the table "CloudRecords", containing possibly incorrect quota values.

The comparisons for the other resource requirements are done in the same way, so we end up having a list of clouds having enough resources to meet the user

demands. If this list is empty, then the message that there are no available clouds capable of providing the required resources is printed. If the list is not empty, then the list **price** is sorted and the list is printed in ascending order by price. This is achieved by the code shown in Figure 4-13.

```
price1 = sorted(price, key=itemgetter('price'))
print 'The clouds capable of providing the required resources are listed in ascending order by price :'
for item in price1 :
    print item['name']
```

Figure 4-13: Printing the sorted cloud list

At this point, the user is once again given the option to choose from cloud querying, cloud proposal or advanced querying, but the difference is that this time the list of clouds is sorted by price and reduced to only those capable of providing the requested resources.

4.4.2 Cloud Querying

Cloud querying includes three functions, one for querying specific clouds, one for making queries about all the available clouds and one for making queries concerning the clouds' history. In any case, the list of clouds includes either all the clouds, if cloud querying was the first choice of the user, or only those having the requested resources, if quota check was executed first.

In order to make queries for a specific cloud, the user first has to choose it from the list. In order to do so, the cloud list is printed and the user is prompted to type the name of the desired cloud. If the typed name belongs to a cloud, its SiteID is retrieved from the table "Sites" and is used in a predefined set of queries, from which the user can select those that will be executed. In the following Figures, we will see how the cloud_id is retrieved from "Sites" and a query example that returns all the VMs whose status is "Active".

```
print 'Select one of the clouds listed below:'
for item in clouds :
    print '%s'%item
cld = str(raw_input(':\n'))
cursor.execute('select id from Sites where name = "'+cld+'"')
cloud_id = cursor.fetchone()[0]
```

Figure 4-14: Retrieving the cloud's SiteID

```
cursor.execute('select VMUUID,MachineName from CloudRecords where SiteID = '
               + str(cloud_id)+' and Status = "ACTIVE"')
print cursor.fetchall()
```

Figure 4-15: SQL query example

In the second case, when a user wants to make queries about all the available clouds, again he can choose one from a predefined set. Let's take an example where the user asks for the list of clouds that have one or more VMs with a CPU count greater than one. In this occasion, we work the same way as in the quota check. While iterating between the available clouds, we perform a MySQL query that returns a list of all the VMs of the selected cloud that have CpuCount>1. If this list is empty, then it means that this cloud has no VM with more than 1 CPU cores, so it is eliminated from the list of clouds that we are going to print as a result of the user's query. Otherwise, the iteration continues and the cloud remains in the list. The respective code is displayed in Figure 4-16.

```
for item in clouds :
    cursor.execute('select id from Sites where name = "'+item+'"')
    cloud_id = cursor.fetchone()[0]
    cursor.execute('select VMUUID from CloudRecords where SiteID = ' + str(cloud_id)+
                   ' and CpuCount > 1')
    dumb = cursor.fetchall()
    if not dumb :
        clouds.remove(item)
if clouds == [] :
    print 'There are no such clouds.'
else :
    print 'The clouds who have a VM with CPU Count greater than 1 are listed below :'
    for item in clouds :
        print item
```

Figure 4-16: Example of a query about all clouds

The last case is when the user opts to make queries concerning the history of the available clouds. Once more, the user may choose from a set of predefined queries. This time, we will describe two examples.

In the first one, the query returns the max, min and average of the CPU usage, memory and network inbound-outbound of all clouds. In the case of CPU usage, especially, the query also returns the values for the CPU count and memory, as they are factors that influence the CPU usage. Inside the iteration between clouds, we initially retrieve the cloud's SiteID from the table "Sites". Then, the commands displayed in Figure 4-17 are executed.

```
print 'MAX'
cursor.execute('select MAX(CpuUsage),VMUUID from CloudRecordsHistory where SiteID = '+ str(cloud_id))
dummy = cursor.fetchone()
cursor.execute('select CpuCount from CloudRecords where VMUUID = '+ str(dummy[1]))
cpucount = float(cursor.fetchone()[0])
cursor.execute('select Memory from CloudRecords where VMUUID = '+ str(dummy[1]))
mem = float(cursor.fetchone()[0])
print 'CPU Usage : %d       -- with %d processor(s) and %d MBs of RAM' %(int(dummy[0]),cpucount,mem)
cursor.execute('select MAX(Memory),VMUUID from CloudRecordsHistory where SiteID = '+ str(cloud_id))
print 'Memory : %d' %(float(cursor.fetchone()[0]))
cursor.execute('select MAX(NetworkInbound),VMUUID from CloudRecordsHistory where SiteID = '+ str(cloud_id))
print 'Network Inbound : %d' %(float(cursor.fetchone()[0]))
cursor.execute('select MAX(NetworkOutbound),VMUUID from CloudRecordsHistory where SiteID = '+ str(cloud_id))
print 'Network Outbound : %d\n' %(float(cursor.fetchone()[0]))
```

Figure 4-17: Retrieving the max values

The first MySQL query returns a tuple of the max value of CU usage and the VMUUID of the VM that had this max usage. Using this VMUUID we can find the

configuration (memory and cpu count) of the VM and print it along with the maximum value. The queries for the memory and network inbound/outbound are simpler, since we do not care about VM configurations. Likewise, there is no need for mentioning the VM configurations for the minimum and average values of the selected attributes, so the respective queries are as simple as those for the maximum memory and network speeds.

In the second example, the user demands the same values (max, min, average) of the same attributes (CPU usage, memory, network inbound/outbound) per cloud, but only for the VMs who are running a specified type of application. As we have already mentioned, the type of application that a VM is running can be optionally stated in its MachineName field. Consequently, our application asks the user to input the name of the selected application and then all the MySQL queries inside the iterations include a clause for the MachineName field. For example, the one that retrieves the maximum network inbound becomes

```
'select MAX(NetworkInbound) from CloudRecordsHistory
where SiteID = '+ str(cloud_id)+' and MachineName =
"'+name+'"'
```

4.4.3 Cloud proposal

This is the function responsible for the decision-making in order to propose the best-fitted resources that meet the user requirements. The basic idea is that the user states his preferences in the sense of the preferred use of the resources, for example he can opt for CPU performance or memory size. These preferences will define the weights of an objective function, according to the outcome of which the proposed cloud will result.

The first step is to set the weights of the function. The user is given five choices, each one setting different values for the weights of CPU (wcpu), memory (wmem), network inbound (wnetin) and network outbound (wnetout). There is one choice where each of the above values is the most important one, e.g. the CPU weight for a CPU-intensive use case, plus one more where the user can input the weights that he wants, as long as their sum is equal to one. The corresponding code and the values of the weights in each case can be seen in Figure 4-18 below.

```
opt = int(raw_input('Press "1" if you opt for CPU\nor press "2" if you opt for
            Memory Capacity\nor press "3" if you opt for Download Speed\nor
            press "4" if you opt for Upload speed\nor press "5" if you want to set
            the parameters yourself:\n'))
if opt == 1 :
    wcpu = 0.6
    wmem = 0.2
    wnetin = 0.1
    wnetout = 0.1
elif opt == 2 :
    wcpu = 0.2
    wmem = 0.6
    wnetin = 0.1
    wnetout = 0.1
elif opt == 3 :
    wcpu = 0.2
    wmem = 0.2
    wnetin = 0.5
    wnetout = 0.1
elif opt == 4 :
    wcpu = 0.2
    wmem = 0.2
    wnetin = 0.1
    wnetout = 0.5
else :
    wcpu = float(raw_input('Enter the weight for CPU:\n'))
    wmem = float(raw_input('Enter the weight for Memory Capacity:\n'))
    wnetin = float(raw_input('Enter the weight for Download Speed:\n'))
    wnetout = float(raw_input('Enter the weight for Upload Speed:\n'))
```

Figure 4-18: Setting the weights of the objective function

Next, we retrieve the current average of the values of the respective weights for each cloud in the list. For example, in order to fetch the average CPU usage of a cloud, the MySQL query is the following

```
'select AVG(CpuUsage) from CloudRecords where SiteID = '+
str(cloud_id)
```

Therefore, we create a dictionary for each attribute (cpu, mem, netin, netout), which contains a record for the average value of every cloud. Since the values of each attribute are of different scale but should be combined in a single objective function, we divide all the values by the maximum one, in order to have a normalized number for each one, like a percentage. For example, if the max of CPU usage is 80% and a certain cloud has an average of 40%, then the CPU dictionary value of this cloud will be 0.5. The code for the CPU part follows in Figure 4-19.

```
maxcpu = max(cpu.values())
for key, value in cpu.items():
    cpu[key] = value / maxcpu
```

Figure 4-19: Normalization of the CPU average values

The next step is to calculate the result of the objective function for each cloud by adding the products of the weights times their corresponding normalized value, as displayed in Figure 4-20. The results are stored in a dictionary named **objective_function**.

```
for item in clouds :
    objective_function[item] = wcpu * cpu[item] + wmem * mem[item]
                    + wnetin * netin[item] + wnetout * netout[item]
```

Figure 4-20: Calculating the objective function of each cloud

Then, we have to find the maximum of these results and trace it back to the cloud that produced it. This procedure is shown below. The best_cloud is the cloud that will be proposed to the user as the best option for his needs.

```
max_obj = max(objective_function.values())
best_cloud = [key for key, value in objective_function.iteritems() if value == max_obj][0]
print best_cloud
```

Figure 4-21: Arriving at the result

Chapter 5

Performance Evaluation

In this Chapter, we are going to describe the tests that we run on our system. These tests concern the database and its ability to handle thousands of records and a big number of clients, all accessing it at the same time. In order to have realistic data stored in the database, we set up a number of VMs running different benchmarks. Then, for different number of records stored in our database, we ran scripts that perform various queries and measured the average response time. We ran the same tests evaluating the performance of the database when more than one clients were accessing it and found its limits. We are going to present and discuss the results of the tests to find out how these conditions affect our system.

During the tests, we also created another account in the Demo service of Synnefo, practically a second account in the ~okeanos cloud but with different credentials from our proper account. The VMs of this new account are registered in a different cloud site, giving us the opportunity to test our system with a simulated environment of multiple cloud sites.

5.1 Setting up the system for the tests

The topics that are related to our preparation for the tests are the demo Synnefo account and how our system supported it, how we retrieved some metrics that are not yet supported by the Synnefo API, the description of the benchmarks that we used and the scripts perform the testing.

5.1.1 The Synnefo Demo account

The Synnefo Demo is a service providing a demo cloud environment, where the VMs that a user creates are deleted after 3 hours. It is solely meant for testing purposes and demonstrates the basic functionality of the Synnefo cloud stack. The reason that we chose to include VMs from Synnefo Demo was to test the

ability of our system to support more than one cloud sites, specifically the performance of our **Sql** script.

As we have mentioned in previous Chapters, each publisher.py script creates one file per cloud (if the user has VMs in this cloud). In this case, even the user is different and has different Authentication URL and token, but for convenience we used a second publisher2.py script in order to collect the data from the demo cloud in the same machine that executes publisher.py. The only difference of the two publisher scripts is the way they retrieve the user's credentials, and how they also retrieve some data that the Kamaki API cannot currently provide (more information on this later).

In order to have a constant and automatic flow of data from the VMs through the publishers and to the collector, we created a bash script named cron_job.sh, containing the two scripts responsible for collecting the information per user and the ssmsend so that the concentrated information is sent to the collector. Then, the bash script is recorded in the crontab so that it is executed as a daemon once per minute.

```
#!/bin/bash

/usr/local/bin/python2.7 /home/publisher.py
/usr/local/bin/python2.7 /home/publisher2.py
ssmsend
```

Figure 5-1: Cron_job.sh

As the VMs of the demo cloud were deleted every three hours, we used the tool `snf-image-creator` provided by Synnefo, which enables the creation of images from the current state of a VM. In this way, as soon as a VM was deleted, we could recreate a duplicate so that more data could fill our database.

5.1.2 Retrieving information about the CPU usage and network download and upload speed

At this time, the Kamaki API does not support a function or call that returns the current CPU usage of a VM, neither its network inbound and outbound. Even though these will be included in the API list in the near future, our need to measure these statistics led us to find other ways of collecting them.

The method that was preferred was the access via ssh. After connecting to the VM, our publisher script executes remotely two test files that we have already installed in every VM, that measure the current CPU usage and the network's inbound and outbound. In order to be able to make successive ssh connections, the pubic keys of the related VMs had to be exchanged. We are now going to see first how the publisher connects to the VMs and then what are the tests that are run in order to retrieve the statistics and send them back to the publisher.

```
cmd = 'ssh ' + server['metadata']['users'] + '@' + server['SNF:fqdn'] + ' \'./test.sh\''
output = subprocess.check_output(cmd, shell=True)
list = output.split('.',1)
extract['CpuUsage'] = list[0]

cmd = 'ssh ' + server['metadata']['users'] + '@' + server['SNF:fqdn'] + ' \'./test2.sh\''
output = subprocess.check_output(cmd, shell=True)
list = output.split(' ')
extract['NetworkInbound'] = int(list[0])
extract['NetworkOutbound'] = int(list[1])
```

Figure 5-2: Connecting with ssh in order to retrieve the CPU usage and network speeds

As we can see in Figure 5-2, in both cases the publisher connects to the VM in a background task (as a subprocess), using information stored in the dictionaries that were returned from previous API calls of Kamaki in order to construct the address of the target VM. The publisher2.py script that is responsible for gathering the information of the VMs belonging to the Synnefo Demo cloud, require different fields from the dictionaries acquired by the Kamaki API. This is the last difference between the two publisher scripts that we mentioned but did not explain before.

In the case of CPU usage, the target is set to execute the bash script test.sh and return its output in a list. This list is created by the "split" command, which splits a string into parts using the provided character (in our case ".") as a separator and limiting the number of splits to a given number (in our case 1). Consequently, the resulting list contains two members, the integer and the decimal part of the CPU usage. We choose to store only the integer part in our **extract** dictionary, as its level of detail is satisfactory. Respectively, the publisher connects to the VM again and starts the execution of test2.sh, the output of which provides a list with two members, the network inbound and outbound.

The test.sh script, which is used in order to retrieve the current CPU usage of a VM, is shown below.

```
top -b -n2 -p 1 | fgrep "Cpu(s)" | tail -1 | awk -F'id,' -v prefix=
"$prefix" '{ split($1, vs, ","); v=vs[length(vs)]; sub("%", "", v);
printf "%s%.1f%%\n", prefix, 100 - v }'
```

Figure 5-3: test.sh

Finally, the test2.sh script that calculates the current download and upload speeds of the VM is presented in Figure 5-4.

```
awk '{if(l1){print $2-l1,$10-l2} else{l1=$2; l2=$10;}}' \
  <(grep eth1 /proc/net/dev) <(sleep 1; grep eth1 /proc/net/dev)
```

Figure 5-4: test2.sh

5.1.3 The benchmarks

For the purpose of our tests, we used four different benchmarks. Three of those benchmarks are part of the **sysbench** suite and consist of a CPU benchmark, a File IO benchmark and a MySQL benchmark [18]. As a network benchmark, we use the Iperf, which is a tool to measure network performance [19]. For every benchmark there is one VM in the ~okeanos cloud that runs only this particular one. In the case of the network benchmark, Iperf's operation requires a client and a server. We have set up the server in the VM that executes the publisher scripts and the client in the VM used for the benchmark. While the Iperf is capable of calculating the current network speeds, we only use it as a means of creating network traffic, because our simple bash script presented before and shown in Figure 5-4 provides a preferable form of response.

As each VM runs a specific benchmark, we used its MachineName field to indicate the type of benchmark that it is running. As a result, we have four VMs added to our pair of the VM that executes the publishers and the one that executes the collector and contains the central database. These four VMs are named as CPU_Bench, Net_Bench, MySQL_Bench and FileIO_Bench.

- CPU Benchmark

```
while true
do
    sysbench --test=cpu --cpu-max-prime=20000 run
done
```

Figure 5-5: Bash script for the CPU benchmark

Above, in Figure 5-5 we can see the bash script that continuously executes the CPU test of the sysbench suite. This script is run in the background (using the **screen** command) and practically has the processor work under full load, resulting in a CPU usage of approximately 100% for a single-core or 50% for a double-core VM, as our test is 1-threaded.

- MySQL Benchmark

```
while true
do
    sysbench --test=oltp --oltp-table-size=1000000 --mysql-db=test --mysql-user=root
     --mysql-password=1234  --max-time=60 --oltp-read-only=on --max-requests=0 --num-threads=8 run
done
```

Figure 5-6: Bash script for the MySQL benchmark

In Figure 5-6, we can see the bash script that runs in the background and is executing the MySQL benchmark. This benchmark requires the creation of a database and a test table and then performs various queries providing statistics such as the achieved transactions per second.

- FileIO Benchmark

```
while true
do
    sysbench --test=fileio --file-total-size=10G --file-test-mode=rndrw --init-rng=on
    --max-time=300 --max-requests=0 run
done
```

Figure 5-7: Bash script for the FileIO benchmark

The script that runs in the background and is responsible for the execution of the FileIO benchmark is shown in Figure 5-7 above. As an initial step, this benchmark requires the creation of a file much bigger than the available RAM of the VM, so that it will not be possible for a large part of it to be cached there. Then, during the execution time, the benchmark performs several reads and writes and measures the time of the transactions (read and write times).

- Network Benchmark

```
while true
do
    iperf -c 83.212.87.22
done
```

Figure 5-8: Bash script for the Iperf

In the above Figure, we can see the script that executes the iperf command in client mode, connecting to the server with the provided IP address. This is the IP of the VM that operates the publishers, which has executed the command

```
iperf -s
```

in order to host an iperf server. In this way, the client and server perform transactions meant to measure the maximum and average download and upload speeds between the two. As we have already mentioned, we are just taking advantage of the created traffic in order to get our own measurements.

5.1.4 The scripts that perform the tests

The core of the tests consists of five python scripts, named test1-5.py. These scripts contain a loop of 20 iterations of a MySQL query to the central database. Each script contains a different query and we are going to present all five of them later on. At the beginning, those scripts connect to the database after they have parsed the required information from the db.cfg file, as our sql.py script did. Then, they execute the following for-loop, as shown in Figure 5-7 for the case of the first type of query.

```
for i in range(0, 20) :

    cursor.execute('select * from CloudRecords')
    x1 = cursor.fetchall()
```

Figure 5-9: The loop including the MySQL queries

These python test scripts are then used inside a bash script, named superscript.sh, which is responsible for executing these tests simultaneously in order to simulate queries to the central database by multiple clients at the same time.

```
start=$(date +%s.%N)

for i in {1..50}
do
        nohup python test1.py & >output.out
done

end=$(date +%s.%N)
runtime=$(python -c "print ${end} - ${start}")

echo "Runtime was $runtime"
```

Figure 5-10: Supertest.sh

As we can see in Figure 5-10, the superscript.sh also measures the time of its execution. Thus, if we divide this time by the number of the iterations (50 in the example above) and by 20, which are the iterations of the loop inside the test python scripts, we will calculate the average time per query.

Let's see now the MySQL queries that each test script performs:

1. `'Select * from CloudRecords'`

This query accesses the table "CloudRecords" and has a big selectivity as it returns all the fields of all its records. However, this table has a relatively small number of records, as it holds a single record for every VM that has been accounted. In our tests, for example, this table had less than 20 records, 6 for the VMs of ~okeanos (the 4 benchmarks, the publisher and the collector) and some more from the Synnefo Demo cloud, as each time a VM was automatically deleted and redeployed from the same image, it had a different VMUUID. Consequently, this query did not depend on the number of records of the database in general, as throughout the tests and the database bulking, the number of its records remained more or less the same and practically negligible when compared to the records of the table "CloudRecordsHistory", as we will see later on.

2. `'Select VMUsage from CloudRecords where SiteID = 1 order by UpdateTime DESC'`

This query again accesses the "CloudRecords" table, but it is less selective as it only returns the field "VMUsage" from the VM records that belong to the SItename (cloud or cloud site) with ID 1. In our case, this is the ~okeanos cloud, and the row of this field's values will be sorted by the UpdateTime of each record, from the most recent to the older.

```
3.   'Select * from CloudRecordsHistory'
```

This query retrieves info from the table "CloudRecordsHistory". Each record of this table only has 8 fields, however the number of records is always increasing as the system is running, so this query is strictly related to the number of records of the table, thus the number of records of the database. So, in the cases of the queries that access this table, we measured the average query time both as a function of the simultaneous client requests and as a function of the database size (number of records).

```
4.   'Select VMUUID from CloudRecordsHistory where SiteID
  = 1 and CpuUsage > 30'
```

Again, the table that is accessed is "CloudRecordsHistory", but this query returns only the field VMUUID of the VMs belonging to ~okeanos and having a CPU usage greater than 30%.

```
5.   'Select          AVG(NetworkOutbound)          from
  CloudRecordsHistory where SiteID = 1'
```

The last query calculates and returns only one number, the average of the field NetworkOutbound (meaning the upload speed) from all the VMs belonging to ~okeanos. Even though it is a query that returns only one number, there still is a cost for its calculation.

5.2 The test results

As we have already mentioned, the measured quantity is the average execution time of a single query to the central database. For the first two types of query, we explained that the size of the database did not affect them, so we have only measured the query time as a function of the number of clients that simultaneously access the database. In contrast, for the rest of the queries, the measured time is a function of both the number of records in the database and the number of clients.

In order to get the results for a different number of records in the database, we left the system working and, consequently, the "CloudRecordsHistory" table started bulking up. We paused the system and executed the supertest.sh several times when our database contained 5.000, 10.000, 20.000, 30.000 and 50.000 records. In each milestone, we ran the supertest.sh for 1, 10, 20, 50 or 100 iterations of the loop shown in Figure 5-10 for each of the three queries. Each iteration of that loop calls the test3.py, test4.py or test5.py, which perform 20 times the respective MySQL query.

5.2.1 Execution time for queries at the table "CloudRecords"

The following table displays the average query time for each of the first two queries as a function of the number of clients that are simultaneously accessing the database.

Table 1: Average query execution time for queries 1 and 2

Number of clients	Query 1 (time in msec)	Query 2 (time in msec)
1	0.46	0.45
10	0.51	0.52
20	0.74	0.73
50	0.77	0.83
100	1.25	1.25

Based on these results, we created a graph for each query and one that combines both, as it is displayed in Figures 5-11, 5-12 and 5-13.

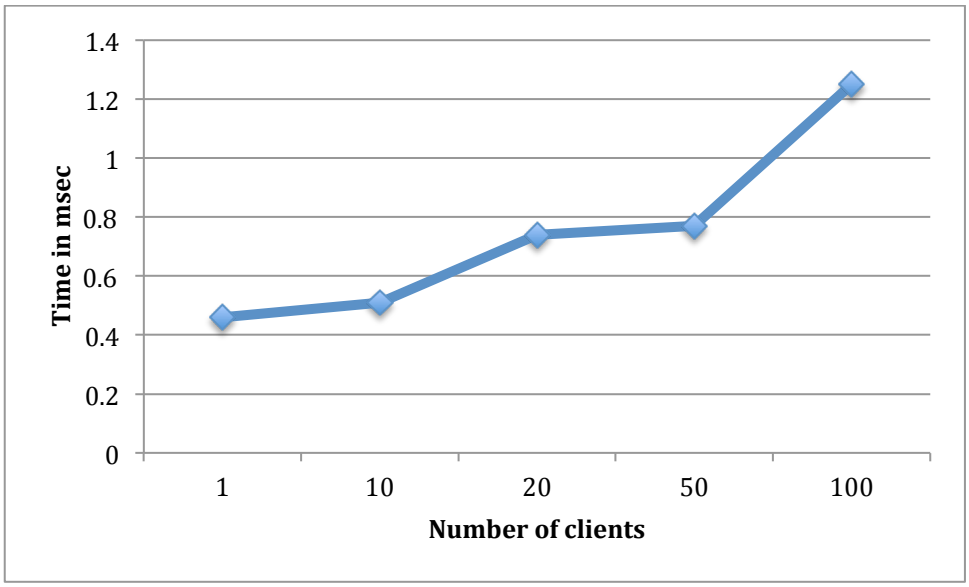

Figure 5-11: Results for Query 1

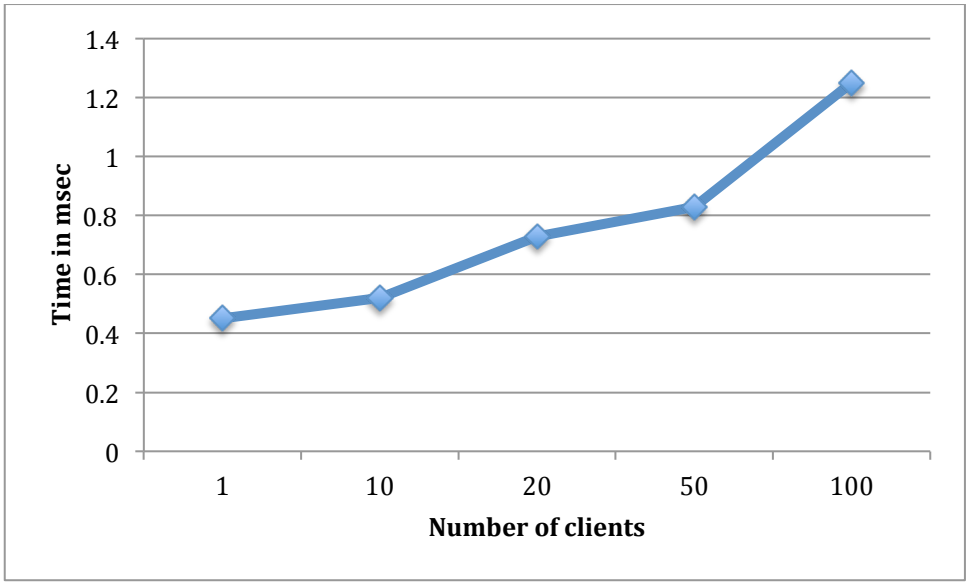

Figure 5-12: Results for Query 2

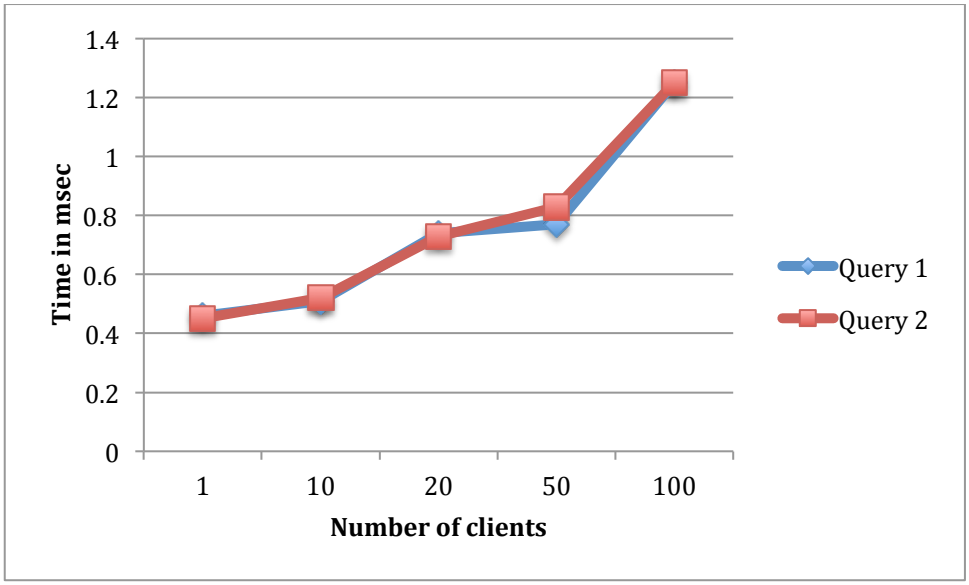

Figure 5-13: Combination of both Query 1 and Query 2

As we can notice from the last Figure, the two queries produced almost identical results, which was kind of expected since both queries access a small number of records from the database, so most of their execution time relates to connecting to the database and generally running the test.py script.

On the other hand, we notice that the query execution time increases alongside the number of clients. This was expected, as the simultaneous accesses create traffic in the database. We can say that the delay added is acceptable, as the duration of a query when 100 clients access the database is only three times greater than the case of a single client.

5.2.2 Execution time for query 3,4 and 5

The 3rd query is `'Select * from CloudRecordsHistory'`.

In the Table 2 below, we include all its execution times as a function of the clients that simultaneously access the database and the number of its records.

Then, we will present a graph that contains one plot for every case of clients' number as a function of the number of records in the database.

Table 2: Average query execution time for query 3

Clients\Records	5k	10k	20k	30k	50k
1	0.46	0.46	0.46	0.46	0.46
10	0.48	0.52	0.53	0.59	0.56
20	0.68	0.72	0.78	0.75	0.64
50	0.89	0.9	1.02	1.19	1.03
100	1.36	1.35	1.36	1.34	1.36

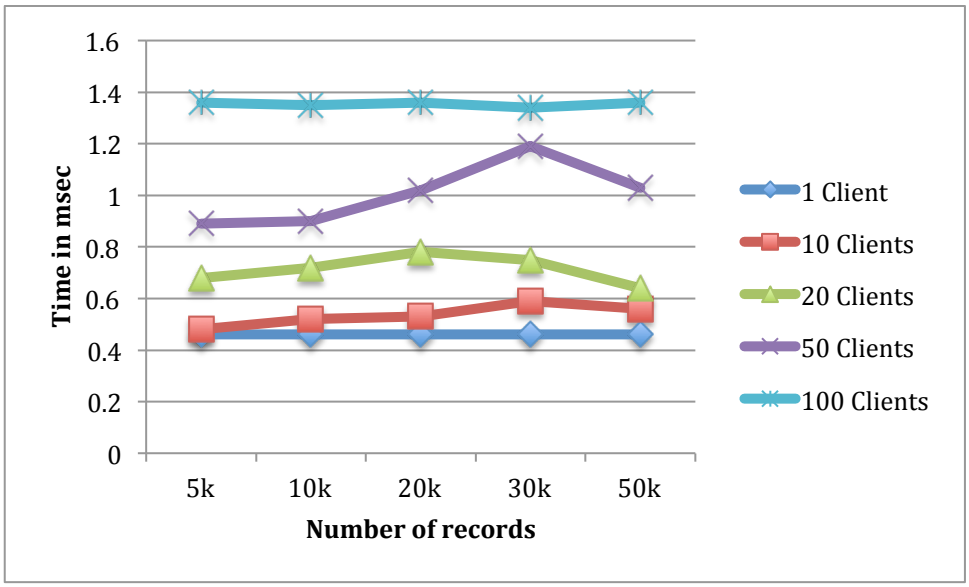

Figure 5-14: Graph of the average execution time for query 3

The 4th query is `'Select VMUUID from CloudRecordsHistory where SiteID = 1 and CpuUsage > 30'`.

In Table 3, we include all its execution times as a function of the clients that simultaneously access the database and the number of its records.

Then, we will present a graph that contains one plot for every case of clients' number as a function of the number of records in the database.

Table 3: Average query execution time for query 4

Clients\Records	5k	10k	20k	30k	50k
1	0.48	0.47	0.47	0.46	0.46
10	0.5	0.54	0.55	0.6	0.53
20	0.76	0.73	0.8	0.77	0.72
50	0.84	0.8	0.86	0.8	0.86
100	1.41	1.3	1.45	1.29	1.39

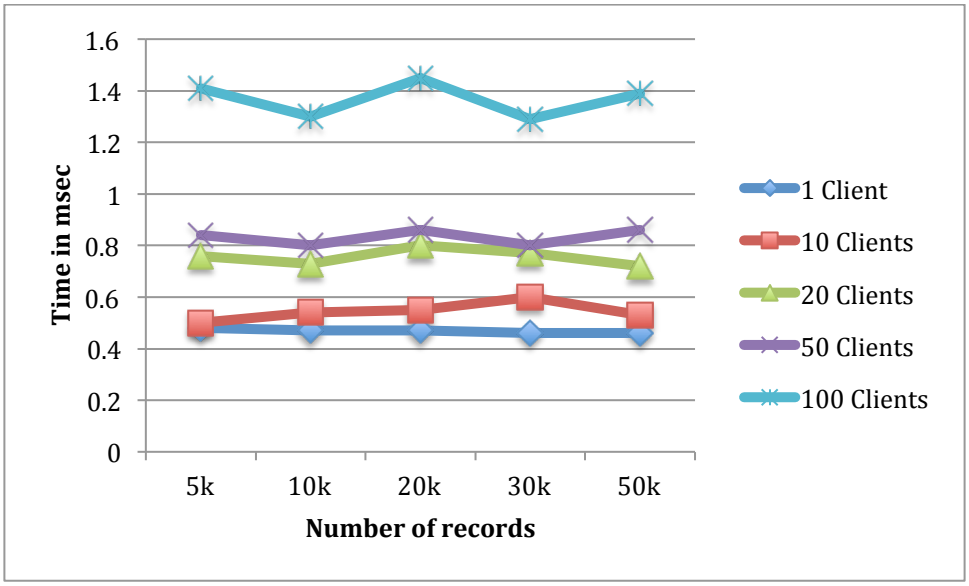

Figure 5-15: Graph of the average execution time for query 4

The 5th query is `'Select AVG(NetworkOutbound) from CloudRecordsHistory where SiteID = 1'`.

In Table 4, we include all its execution times as a function of the clients that simultaneously access the database and the number of its records.

Then, we will present a graph that contains one plot for every case of clients' number as a function of the number of records in the database.

Table 4: Average query execution time for query 5

Clients\Records	5k	10k	20k	30k	50k
1	0.46	0.47	0.48	0.46	0.47
10	0.57	0.52	0.51	0.53	0.5
20	0.7	0.78	0.85	0.72	0.77
50	0.88	0.9	0.88	0.78	0.83
100	1.37	1.4	1.35	1.26	1.25

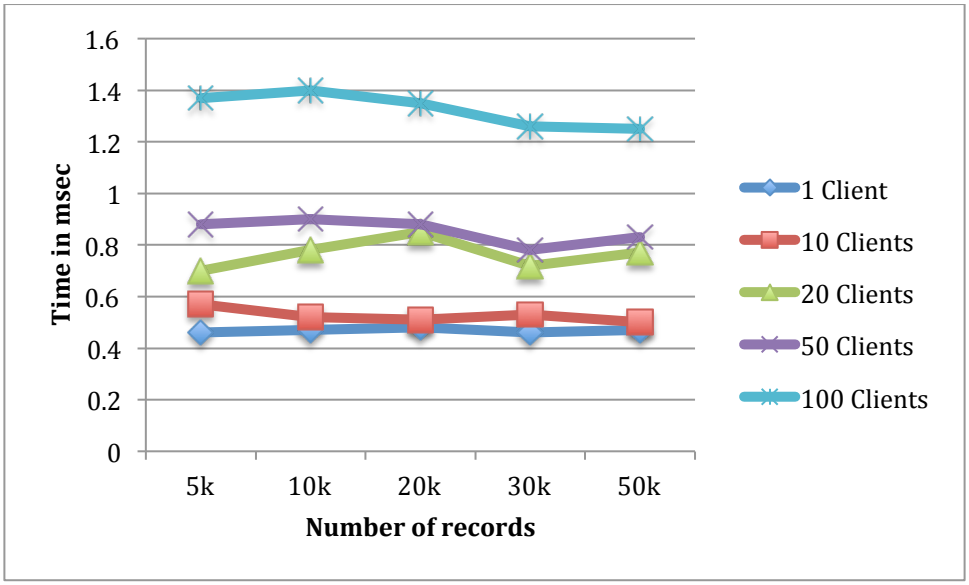

Figure 5-16: Graph of the average execution time for query 5

From Figures 5-14, 5-15 and 5-16, we can observe two things. First, the number of records does not affect systematically our results. There are many cases where more results are combined with greater query execution time, but in other cases the execution time is unaffected or even reduced as the databases grows bigger. In tests that we ran with more than 100 clients, e.g. 200 or 500, the database could not serve all the simultaneous requests, and produced errors with the description "too many connections". This obviously happened only for the queries at the table "CloudRecordsHistory", which is the one with the numerous records that could not be accessed at the same time. Consequently, we can conclude that as long as the database can support our queries, then the number of records does not influence their performance.

The second observation is related to the number of clients. It is evident that as their number increases, so does the execution time of all the queries, which is something expected as the extra accesses create a delay for all the database transactions. This increase also seems to influence all the queries in the same way as the execution times were similarly extended for every query type.

Chapter 6

Conclusions

In the previous Chapters, we have explained the motivation, the architecture and the implementation details of the system that we developed in the context of this thesis. We are now going to summarize what we achieved and also provide some thoughts concerning future enhancements and extensions of our system.

6.1 Synopsis

In this thesis, we first introduced the concept of cloud computing, as well as the benefits of cloud federations and their need for a mechanism that will be responsible for the deployment of the demanded resources. Then, we presented our implementation, which gathers information for all the VMs of a federated environment on a per-user basis, stores it in a central database and can make propositions for the cloud from which the resources that a user requests can be deployed. As of now, our system is functional for the ~okeanos and Demo Synnefo clouds.

The tests that we ran proved that our implementation is stable, efficient and as scalable as it can be, given the machine that hosted our central database. In a more realistic scenario, this machine would certainly have better specifications than our 1-core CentOs VM, so the database limitations concerning the number of simultaneous requests would be far less. Consequently, we can say that we achieved our goals, as we created a scalable and lightweight tool that with few enhancements could be immediately used in real conditions.

6.2 Future extensions

As we have already mentioned, there are many extensions that could be added to our system, providing better compatibility, performance, precision of proposition and more functionality. First of all, by adding support for the APIs of different cloud software, more clouds would be able to be managed in a federated environment. We have seen that our publisher is generally based on the OSSSM project that is meant to collect data from Openstack-operated clouds;

so adding support for Openstack is a quite straightforward task. In the same way, more clouds running on different software could be supported just by extending the publisher in order to use the respective API.

Concerning our current publisher that supports the Synnefo cloud software, there are two changes that could be made in the future. The first is related to the pricing. Our current implementation uses either static pricing or a calculation based on the demanded resources. However, Synnefo now has its own billing service, Aquarium, and that can be taken into account by our publisher in order to have a more consistent model of the ~okeanos cloud, providing accurate prices and, therefore, making more suitable propositions. The second change concerns the retrieval of the metrics of CPU usage and network inbound/outbound, which are not yet provided by the Kamaki API of Synnefo, forcing us to retrieve them by using ssh. As these functionalities will be soon added to Kamaki, our publisher executable will need to be updated with the added API calls.

Relating to the central database, one modification that would alleviate the database would be to keep another table named “Users”, where the user ID and his quotas would be stored. This would eliminate the need for storing the quotas of the VM’s user in the record of every VM, which now creates unnecessary duplication of data. For example, if a user currently has a large number of VMs in the same cloud, then the database stores the quota data as many times as the number of these VMs. In order to make this change, the Apel system would have to be modified, more specifically the database schema and the database loading script.

Finally, there are two modifications that could be made to the Sql.py, the script that queries the database and is responsible for the cloud proposition. The first of these would be the addition of more queries that could be available for the user. Giving the user more choices of queries, he could be better informed about certain aspects of the VMs statistics or specifications that are most important to him. The second and most sophisticated extension would be the modification of the objective function that is used in order to propose the best fitted resources for the user demands. There are many different algorithms that could be used or maybe even a combination of more than one. These more complex algorithms would have to be tested so as to prove that they return the desired results, but it is an area with great research interest.

Chapter 7

Bibliography

[1] “The NIST definition of Cloud Computing”, http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[2] “A brief history of cloud computing”, http://thoughtsoncloud.com/2014/03/a-brief-history-of-cloud-computing/

[3] “Cloud Federation”, http://searchtelecom.techtarget.com/definition/cloud-federation

[4] “Federation is the Future of the Cloud”, http://www.datacenterknowledge.com/archives/2012/09/17/federation-is-the-future-of-the-cloud/

[5] “Cloud Broker”, http://searchcloudprovider.techtarget.com/definition/cloud-broker

[6] Ang Li, Xiaowei Yang, Srikanth Kandula, Ming Zhang, “CloudCmp: Comparing Public Cloud Providers”, IMC 10, 2010.

[7] Rodrigo N. Calheiros, Adel Nadjaran Toosi, Christian Vecchiola, Rajkumar Buyya, “A coordinator for scaling elastic applications across multiple clouds”, Future Generation Computer Systems, 2012.

[8] Alba Amato, Beniamino Di Martino, Salvatore Venticinque, “Cloud Brokering as a Service”, Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 2013

[9] About ~okeanos, https://~okeanos.grnet.gr/about/

[10] Evangelos Koukis, Panos Louridas, “~~okeanos IaaS”, EGI Community Forum 2012 / EMI Second Technical Conference, 2012

[11] About Synnefo, https://www.synnefo.org/about/

[12] Welcome to the Synnefo documentation, https://www.synnefo.org/docs/synnefo/latest/index.html

[13] Kamaki project documentation, https://www.synnefo.org/docs/kamaki/latest/overview.html

[14] APEL/SSM, https://wiki.egi.eu/wiki/APEL/SSM

[15] APEL, https://wiki.egi.eu/wiki/APEL

[16] APEL/SSM Openstack documentation, https://github.com/EGI-FCTF/osssm/wiki

[17] dirq, https://code.google.com/p/dirq/

[18] How to benchmark your System with sysbench, http://www.howtoforge.com/how-to-benchmark-your-system-cpu-file-io-mysql-with-sysbench

[19] iperf, https://iperf.fr