ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Dottorato di ricerca in
INFORMATICA

Ciclo XXVIII

Settore concorsuale di afferenza 09/H1
Settore scientifico disciplinare ING-INF/05

AUTONOMIC MANAGEMENT OF CLOUD VIRTUAL INFRASTRUCTURES

Presentata da:
DANIELA LORETI

Relatore:
PROF.SSA ANNA CIAMPOLINI

Coordinatore dottorato:
PROF. PAOLO CIACCIA

Esame finale anno 2016

WEBSITE:
https://www.unibo.it/sitoweb/daniela.loreti/

E-MAIL:
daniela.loreti@unibo.it

ABSTRACT

The new model of interaction suggested by Cloud Computing has experienced a significant diffusion over the last years thanks to its capability of providing customers with the illusion of an infinite amount of reliable resources. Nevertheless, the challenge of efficiently manage a large collection of virtual computing nodes has just been partially moved from the customer's private datacenter to the larger provider's infrastructure that we generally address as "the cloud". A lot of effort – in both academic and industrial field – is therefore concentrated on policies for the efficient and autonomous management of virtual infrastructures.

The research on this topic is further encouraged by the diffusion of cheap and portable sensors and the availability of almost ubiquitous Internet connectivity that are constantly creating large flows of information about the environment we live in. The need for fast and reliable mechanisms to process these considerable volumes of data has inevitably pushed the evolution from the initial scenario of a single (private or public) cloud towards cloud interoperability, giving birth to several forms of collaboration between clouds. The efficient resource management is further complicated in these heterogeneous environments, making autonomous administration more and more desirable.

In this thesis, we initially focus on the challenges of autonomic management in a single-cloud scenario, considering the benefits and shortcomings of centralized and distributed solutions and proposing an original decentralized model. Later in this dissertation, we face the challenge of autonomic management in large interconnected cloud environments, where the movement of virtual resources across the infrastructure nodes is further complicated by the intrinsic heterogeneity of the scenario and difficulties introduced by the higher latency medium between datacenters. According to that, we focus on the cost model for the execution of distributed data-intensive application on multiple clouds and we propose different management policies leveraging cloud interoperability.

PUBLICATIONS

Parts of the work in this thesis have previously appeared in the following publications:

[1] D. Loreti, A. Ciampolini, F. Chesani, and P. Mello, "Process mining monitoring for map reduce applications in the cloud," in *CLOSER, 6th International Conference on Cloud Computing and Services Science*, 2016.

[2] D. Loreti and A. Ciampolini, "Mapreduce over the hybrid cloud: a novel infrastructure management policy," in *Proceedings of the 8th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, 2015.

[3] F. J. Clemente-Castellò, B. Nicolae, K. Katrinis, M. M. Rafique, R. Mayo, J. C. Fernàndez, and D. Loreti, "Enabling big data analytics in the hybrid cloud using iterative mapreduce," in *Proceedings of the 8th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, 2015.

[4] D. Loreti and A. Ciampolini, "A hybrid cloud infrastructure for big data applications," in *Proceedings of the 17th International Conferences on High Performance Computing and Communications*, IEEE, 2015.

[5] D. Loreti and A. Ciampolini, "Shyam: a system for autonomic management of virtual clusters in hybrid clouds," in *1st Workshop on Federated Cloud Networking (FedCloudNet)*, Springer, 2015.

[6] D. Loreti and A. Ciampolini, "A distributed self-balancing policy for virtual machine management in cloud datacenters," in *High Performance Computing Simulation (HPCS), 2014 International Conference on*, pp. 391–398, July 2014.

[7] D. Loreti and A. Ciampolini, "A decentralized approach for virtual infrastructure management in cloud datacenters," *International Journal On Advances in Intelligent Systems*, vol. 7, pp. 507–518, December 2014.

[8] D. Loreti and A. Ciampolini, "Policy for distributed self-organizing infrastructure management in cloud datacenters," in *ICAS 2014, The Tenth International Conference on Autonomic and Autonomous Systems*, pp. 37–43, April 2014.

AWARDS

The work "Policy for Distributed Self-Organizing Infrastructure Management in Cloud Datacenters" was awarded with a Best Paper Award price after the presentation at ICAS 2014, the Tenth International Conference on Autonomic and Autonomous Systems, held in Chamonix, France on April 20 - 24, 2014.

CONTENTS

ACRONYMS

API	Application Programming Interface 14
ASP	Application Service Provider 3
BF	Best Fit xiv
BFD	Best Fit Decreasing 15
BPM	Business Process Management 145
CLI	Command Line Interface 75
CPU	Central Processing Unit 3
CSD	Complete Scale Down 100
DAM	Distributed Autonomic Migration 20
DAM-Sim	DAM Simulator 34
EC	External Cloud 2
FSD	Fast Scale Down 100
GPU	Graphics Processing Unit
GUI	Global User Interface 67
HDD	Hard Disk Drive 86
HDFS	Hadoop Distributed File System 75
HyIaaS	Hybrid Infrastructure as a Service 79
HyMR	Hybrid MapReduce 97
IaaS	Infrastructure as a Service 4
IC	Internal Cloud 2
ICT	Information and Communications Technology 3
I/O	Input/Output 59
IT	Information Technology 1
LTL	Linear Temporal Logic 130
MC	Maximum Correlation 16
MB	Mobile Balance 30
MBFD	Modified Best Fit Decreasing 15
MIPS	Millions of Instructions Per Second 23
MMT	Minimum Migration Time 16

MoM	Minimization of Migrations 16
MWF	Mobile Worst Fit 30
OCCI	Open Cloud Computing Interface 66
OGF	Open Grid Forum 66
PaaS	Platform as a Service 3
QoS	Quality of Service 8
RAM	Random Access Memory 16
SaaS	Software as a Service 3
SLA	Service Level Agreement 5
SHYAM	System for HYbrid clusters with Autonomic Management 78
URL	Uniform Resource Locator
VM	Virtual Machine 1
VCPU	Virtual Central Processing Unit 86
WF	Worst Fit xiv

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

As the advent of Cloud Computing has suggested a new model of interaction between the Information Technology (IT) provider and the customers, the computing infrastructure has been turned into a service that the customer can exploit according to her needs after a contract negotiation with the provider. This paradigm experienced a significant diffusion during the last few years thanks to its capability of relieving companies of the burden of managing their IT infrastructures. At the same time, the demand for scalable yet efficient and energy-saving cloud architectures has made the Green Computing area stronger, driven by the pressing need for both greater computational power and restraint of economical and environmental expenditures.

The challenge of efficiently managing a collection of physical servers avoiding bottlenecks and power waste, is not completely solved by the Cloud Computing paradigm, but it is only partially moved from the customers's IT infrastructure to the provider's data centers. Since cloud resources are often managed and offered to customers through a collection of Virtual Machines (VMs), a lot of efforts concerning the Green Computing trend are now concentrating on finding the best VM allocation to obtain efficiency without compromising performances.

Furthermore, Cloud Computing is facing the challenge of an ever growing complexity due to the increasing number of users and their augmenting resource requests. This complexity can only be managed by providing the cloud infrastructure with an autonomic behavior, so that, it can autonomously allocate and move VMs across the datacenter's nodes without the human intervention.

Recently, the cloud computing model has seen the evolution from the initial scenario of a public cloud offering its resources to customers through virtualization and Internet, toward the concept of hybrid cloud, where the classic scenario is enriched with a private (company owned) cloud e.g., for the management of sensible data. The hybrid cloud paradigm has

gained further attraction as the attention to computationally intensive applications increased. As a relevant example, we consider the so-called *big data* scenario.

The trending evolution towards the "Internet of things" and the general increase in broadband are constantly creating large volumes of data that need to be processed for higher intelligence. In this scenario, assuming a limited set of computing resource available on-premise – i.e., in the private INTERNAL CLOUD (IC) –, it is crucial to allow the dynamic provision of additional computing nodes by relaying on the resource availability of an EXTERNAL CLOUD (EC), e.g. a public cloud.

Similarly to the classic cloud paradigm, the hybrid scenario would benefit from an autonomous management system able to dynamically scale-up towards the public cloud when further resources are requested (or scale-down to reduce the infrastructure costs).

Nevertheless, in a hybrid setup, further issues emerge e.g., the benefits of accelerating the computation can be mitigated (or even frustrated) by the challenges of data locality and data movement crossing the on-premise boundaries – which is usually over a higher latency medium, when compared to a co-located servers scenario.

This thesis studies the policies and the mechanisms for autonomic VM management in public and hybrid cloud architectures. On one hand, in the context of a single public cloud, we investigate both distributed and centralized solutions by reviewing previous state-of-the-art works, and by contributing with our own proposal to decentralize the management of the virtual infrastructure.

On the other hand, we highlight the main challenges in the field of multiple clouds by discussing the details and the intricacies of executing intensive data processing over hybrid environments. In this regard, we propose our contribution to enable infrastructure scaling and overcome the drawbacks introduced by the multiple cloud scenario.

In the rest of this chapter, we describe cloud architectures (Chapter 1.1) as well as the issues related to VM management (1.2) and we introduce the multiple cloud scenario in detail (1.3).

1.1 THE CLOUD

As Information and Communications Technology (ICT) is getting more and more important for business, a growing number of modern companies is facing the need for dedicated hardware and computing infrastructure. Nevertheless, the management of a datacenter can be too expensive for a small enterprise. Companies can autonomously buy the hardware and software their business requires but they need the support of an IT provider to firstly install and configure these complex architectures. Moreover, the computational requirements of an enterprise can dynamically change as its business grows or decrease making the continuous intervention by IT providers crucial to adapt the datacenter structure. The enterprise can also decide not to relay on external IT companies but hire high qualified personnel to manage the computer infrastructure. However, the costs of all these solutions remain very high.

The effort of many IT companies concerning cloud computing is to supply infrastructures that can be completely managed in a rather simple way, not only by the provider, but also by nontechnical employees of the customer enterprise. From the cloud customer point of view, since no more specific operations on hardware and operating systems are necessary, no more high qualified staff is needed to deal with computing infrastructure issues.

As Cloud Computing rewrite the rules of business model involving IT providers and customers, it is sometimes addressed as the "fifth generation of computing", because its revolution comes after the Mainframe, the personal computer, the client-server model and the web [3]. However, Cloud Computing can be seen as a complex service: a set of technologies allowing the customer to remotely use hardware – storage, Central Processing Unit (CPU), network, etc. – and software resources owned by the IT provider. As pointed out by [3], there are three fundamental kinds of services related to Cloud Computing:

- Software as a Service (SaaS) - Allow the remote use of some softwares, usually web services, recalling the Application Service Provider (ASP) philosophy.
- Platform as a Service (PaaS) - Similar to SaaS, but involving more than one software. Usually refers to a collection of services, programs and libraries utilized by

a remote user as a platform for the deployment and execution of his applications.

- INFRASTRUCTURE AS A SERVICE (IAAS) - Addresses a remote use of hardware resources to allow distributed parallel computations. The IT infrastructure is a collection of physical machines connected by a network and able to cooperate and create clusters for elaborations with high computational requirements. The physical resources are allocated to the customer on request only when she actually needs to use them.

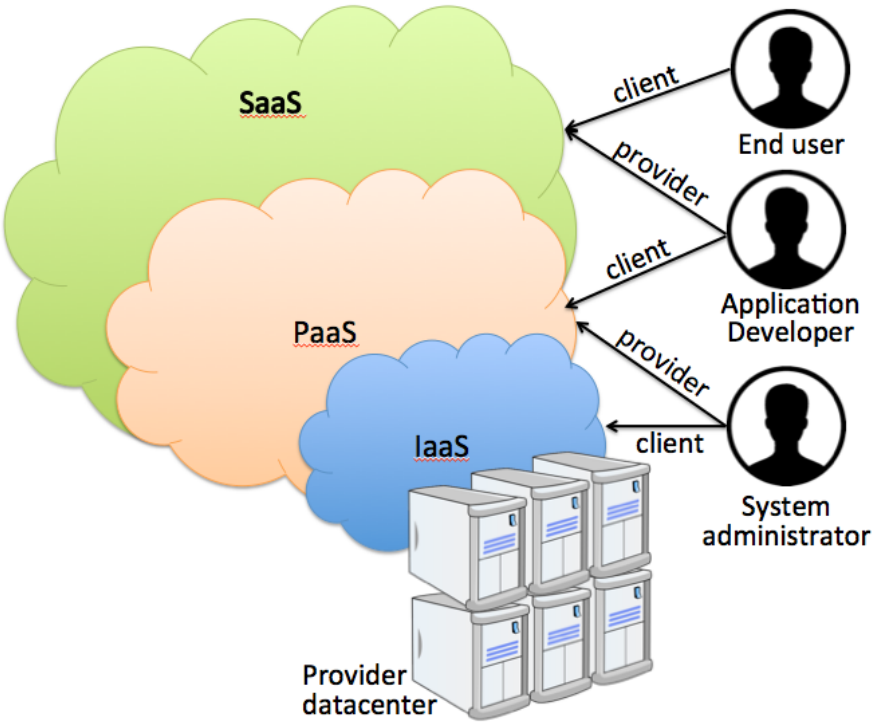


Figure 1.1: The cloud paradigm involves different kinds of users, providing them with software, development platforms or infrastructural resources according to their needs.

As illustrated in figure 1.1, the cloud paradigm can include all these service models because it refers to both the applications delivered as services over the Internet and the hardware and platforms that provide those services. The services themselves have long been referred as SAAS. The hardware and software of the datacenter hosting these services is what now is commonly referred with the term "cloud" [4].

1.2 CLOUD MANAGEMENT OPEN ISSUES

A cloud datacenter is typically composed of a large number (hundreds or even thousands) of physical servers, hosting a collection of VMs accessed remotely by customers. In this scenario, a software component (e.g., OpenStack [5], Amazon EC2 [6], Aptana Cloud [7], Aneka Cloud [8], etc.) is responsible for the management of the cloud infrastructure, including the migration of VMs and the allocation of physical resources to customers.

Through the use of virtualization, multiple VMs can run on a single physical machine. A widespread practice in this scenario is to allocate the VMs on a machine such that the total amount of virtual resources requested never exceeds the physical resources available. Nevertheless, since the resource requirements of the applications running on the VM are highly dynamic, such a static allocation of resources can still lead to a significant resource underutilization. To address this, the resources can be *oversubscribed* – e.g., by promising to a set of VMs more CPU than actually possessed by the hosting physical machine. This can drastically increase the utilization of individual hosts, however, it can also lead to resource contention, and thus to VM performance degradation. A dynamic approach to VM allocation is therefore required. This can be achieved through the use of live migration, i.e., moving a running VM from one host to another facing a minimal downtime.

The open issues of cloud management are strictly related to the target to be achieved. Therefore, we can identify three research directions according to the specific objective:

- *power saving*: since an idle server is demonstrated to consume around 70% of its peak power [9], packing the VMs into the lowest possible number of servers and switching off the idle ones, can lead to a higher rate of power efficiency, but can also cause performance degradation in customers's experience and Service Level Agreement (SLA) violations. The operation of turning back on a previously switched off host can be very time-consuming. In modern data centers, in order to obtain a more reactive system, the underloaded hosts are never completely switched off, but only put into sleep mode. This technique slightly increases the power

consumption, but also speeds up the wake up process when further computational power is needed;
- *load balancing*: allocating VMs in a way that the total cloud load is balanced among nodes results in a higher service reliability and less SLA violations, but forces the cloud provider to maintain all the physical machines switched on and, consequently, causes unbearable power consumption and excessive costs;
- *dynamic behavior*: we must take into account that such a system is continuously evolving. The demand for application services, computational load and storage may quickly increase or decrease during the execution, thus further complicating management operations.

Due to these targets (often in contrast with each other) the VM management in a Cloud Computing datacenter is intrinsically very complex and can be hardly solved by a human team of system administrators, especially when the size of the datacenter is big. For this reason, it is desirable to provide the infrastructure with the ability to operate and react to dynamic changes without human intervention.

While the majority of the efforts in this regard relays on centralized infrastructures (where a single cloud controller is responsible for identifying and reacting to critical conditions), we propose a solution to increase system scalability and reliability by leveraging a distributed approach.

1.3 FROM THE SINGLE CLOUD TO CLOUD INTEROPERABILITY

Over the years, several technologies – e.g., virtualization, grid computing, etc. – have matured and significantly contributed to enable the cloud paradigm. However, cloud computing still suffers from lack of standardization: most cloud providers propose their own solutions along with proprietary interfaces to access resources and services. This heterogeneity is a crucial obstacle to the realization of ubiquitous cloud [1]. An unavoidable barrier at this stage is *vendor lock-in* [10, 11]: customers using cloud services need to follow cloud-specific interfaces when creating their own applications. This complicates a future relocation and makes it expensive.

A relevant survey on cloud interoperability scenarios and challenges is presented in the work by Toosi et al. [1].

According to [1], cloud interoperability can be obtained in two ways:

- with *standard interfaces*, to which providers must be compliant;
- by using a *service broker*, which translates messages between different cloud interfaces, makes customers able to switch between different clouds and allow cloud providers to interoperate.

However, since one comprehensive set of standards is difficult to develop and hard to be adopted by all providers, a combination of both these approaches is also possible.

There are several benefits of an interconnected cloud environment for both cloud providers and customers, and there are essential motivations for cloud interoperability such as scalability, availability, low-access latency and energy efficiency.

Without *provider-centric* solutions – such as the adoption and implementation of standard interfaces, protocols, formats, and architectural components that facilitate collaboration – cloud interoperability is hard to achieve. *Hybrid Cloud, Cloud Federation*, and *Inter-cloud* are the most remarkable scenarios of provider-centric approaches [1].

The hybrid cloud is the result of the collaboration between a private (company-owned) and a public (provider-owned) cloud that enables *cloud bursting* (i.e., the dynamic provisioning of provider-owned additional resources to an on-premise architecture). This practice allows a customer application to be executed in an on-premise private data center and burst off-premise towards a public cloud when peaks in resource demand occurr.

When providers share their cloud resources, we talk about cloud federation. This paradigm is very similar to the hybrid cloud model because the providers aim to overcome the limited nature of their local infrastructure by outsourcing requests to other members of the federation. Differently from hybrid cloud, the actors of the cloud federation no longer include the final users of the infrastructure, but only two or more providers. Moreover, cloud federation allows providers operating at low utilization to lease part of their resources to other federation members in order to avoid wasting their unused compute resources.

Finally, in the Inter-cloud paradigm, all clouds are globally interconnected, forming a worldwide cloud federation. This paradigm supports the VM migration and dynamic scaling of applications across multiple clouds.

In case cloud interoperability is not supported by cloud providers, the customers can benefit from *client-centric* interoperability facilitated by user-side libraries or third-party brokers. *Multicloud* application deployment using adapter layer provides the flexibility to run applications on several clouds and reduces the difficulty in migrating applications across clouds. *Aggregated service by broker*, a third-party solution in this regard, offers an integrated service to users by coordinating access and utilization of multiple cloud resources [1].

Cloud interoperability is a challenging issue and requires substantial efforts to overcome the existing obstacles. These include both functional and nonfunctional aspects. This dissertation mainly focus on the hybrid cloud paradigm as a first step towards the complete cloud interoperability realizable through the Inter-cloud. We expect the middle step in this road of innovation to be the federated cloud. Nevertheless, Inter-cloud and federated cloud are out the scope of this dissertation.

Combining both on-premise (company owned) and off-premise (owned by a third party provider) cloud infrastructures, the hybrid scenario can capture a broader use-case than the public cloud: e.g., using off-premise resources for being able to guarantee a minimum Quality of Service (QoS), satisfying a predefined deadline for a data-processing application, or partitioning computation between on- and off-premise zones in compliance with security requirements (for instance, if part of the data is not allowed to cross the boundary of the on-premise infrastructure).

Usually, the on-premise cloud is owned and managed by the customer company, while the off-premise cloud is a collection of physical nodes owned by the cloud provider and reserved (upon payment) to the customer, who can either decide to manage the nodes by herself or pay for a provider managed service.

Another possible hybrid scenario consist of the off-premise cloud realized through a public cloud, where the customer does not have any control on the physical allocation of the VMs spawned off-premise.

Offering the possibility to extend the company-owned cloud by bursting towards an off-premise datacenter, the hybrid cloud paradigm has gained particular attraction as the so-called *big data* field has become more and more important. Indeed nowadays, data is the new natural resource. The tremendous increase in broadband and generally Internet penetration is constantly creating large volumes of data that need to be processed for higher intelligence. The trending evolution towards an 'Internet-of-everything' is further aggravating the problem e.g. through the exponential increase in the use of mobile devices and the deployment of sensors across application/industrial domains (from surveillance cameras for national security to biometric sensor in healthcare). All these create a massive need for faster, affordable and reliable time to insight; the latter depends in part on the availability of large-scale analytics infrastructures and platforms.

The issues of VM management in a large datacenter for public cloud are mirrored in the hybrid context and further complicated by the constraints introduced by the limited inter-cloud bandwidth and the inherent lack of functionality in application level software (e.g., high level mechanisms to efficiently face the infrastructure partitioning).

All these challenges, must be taken into account when dealing with a hybrid scenario and especially when the practice of spawning VMs towards off-premise cloud is used to enable big data analysis.

1.4 THESIS CONTRIBUTIONS AND OUTLINE

In this dissertation, we investigate design and implementation issues related to the autonomic management of VMs in single data centers and hybrid clouds. In the latter case, we particularly focus on techniques to enable the execution of data-intensive applications deployed over multiple clouds.

Starting from deep technical analyses of existing and state-of-the-art contributions from industry and academia, we define a classification of the works in the field of autonomous infrastructures and we propose two different approaches to the problem – one for the single cloud datacenter and one for the hybrid context.

We validate our ideas by building prototypes of systems, techniques, and algorithms and by evaluating them through extensive experimental campaigns on simulated and physical distributed deployments of realistic application scenarios.

This thesis proposes our contribution to the emergent autonomous computing field through several works briefly introduced in the following.

- *Chapter 2*. We propose a classification of the existing works on autonomic infrastructure management in the context of a single cloud and we describe the architecture of our contribution as well as the proposed VM management policies.
- *Chapter 3*. We focus on multiple clouds by clarifying the classification of the existing interoperability scenarios and we describe the proposed framework architecture and policies to enable infrastructure management with particular attention to the execution of data-intensive applications.
- *Chapter 4*. We describe a novel declarative approach to crucial elements of autonomic strategies: the monitoring and recovery features. Leveraging the previously described framework architecture, we depict the main advantages of the approach when dealing with complex distributed computations, such as data-intensive applications.

The thesis is concluded by Chapter 5, where we summarize the most important findings of our work and highlight interesting and still open research directions.

2 AUTONOMIC VM MANAGEMENT IN A SINGLE CLOUD

While the cloud computing paradigm has been extensively used to enable the provisioning of elastic services (where the demand for resources can rapidly change during the month, or even the day), the problem of efficiently manage a collection of physical servers hosting a virtual infrastructure (in order to minimize the power waste without compromising the performance) is still an open challenge for cloud providers. Most of them prefer not to risk violating the SLAs with the client and, therefore, suffer the extra-cost due to all the servers turned on instead of putting some of them in sleep mode and save power.

Nevertheless, both the academia and industry has dedicated a lot of attention to the so-called Green Computing area and, in particular, to the VM management problem as a possible way to increase the efficiency of large scale data centers.

In this Chapter, we first propose a classification of the works in the Green Computing area (Chapter 2.1). Then, in Chapter 2.2, we describe our approach in detail and finally (Chapters 2.3 and 2.4) focus on two policies for VM management relevant to the proposed model. The last Chapter (2.5) is dedicated to our conclusions and expectation for the future of this field.

2.1 POSITIONING OUR CONTRIBUTION

Most of the efforts in the field of VM management relay on centralized solutions [12–14]. Following this approach, a particular server in the cloud infrastructure is in charge of collecting information on the whole set of physical hosts, taking decisions about VMs allocation or migration, and operating to apply these changes on the infrastructure [15, 16]. The advantages of centralized solutions are well known: a single node with complete knowledge on the infrastructure can take better decisions and apply them through a restricted number of migrations and communications. However, scalability and

reliability problems of centralized solutions are known as well. Furthermore, as the number of physical servers and VMs grows, solving the allocation problem and finding the optimal solution can be time expensive, so some other approximation algorithm is typically used to reach a sub-optimal solution in a fair computation time [17–19].

The adoption of a centralized VM management is even unfeasible in those contexts (like Community Cloud [20, 21] and Social Cloud Computing [22]), in which both the demand for computational power and the amount of offered resources can rapidly – and even dramatically – change.

An alternative approach in VMs management is bringing allocation and migration decisions to a decentralized level, allowing the cloud's physical nodes to exchange information about their current VM allocation and self-organize to reach a common reallocation plan.

Decentralized solutions [23–26] can positively face scalability and reliability problems, providing the infrastructure with an autonomic collective behavior. As in a swarm of insects or a bird flock, each agent (running on a physical server or a VM) is able to execute only simple reactions to events, but the sum of these actions results in a complex emergent behavior. However, the distributed management of VMs brings other challenges, like coordination of the agent colony, loss in the policy efficacy due to a partial local knowledge and difficult evaluation of the policy evolution.

To better investigate the field of VM management in a cloud infrastructure, we start focusing on the related works by classifying them according to specific subproblems addressed (as suggested by Beloglazov et al. in [17]):

- *VM Placement* solutions, focus on finding the best allocation for a given collection of VMs;
- *VM Selection*, given a group of VMs allocated on a collection of servers, the goal is to find which is the best VM to relocate;
- *Holistic* solutions, focus on both selection and placement of VMs at the same time.

As illustrated in Figure 2.1, orthogonally to this classification, we can identify other categories related to the technology adopted to solve problems:

- works exploiting some concepts from the *autonomic control theory*, in order to maintain the system in balance;

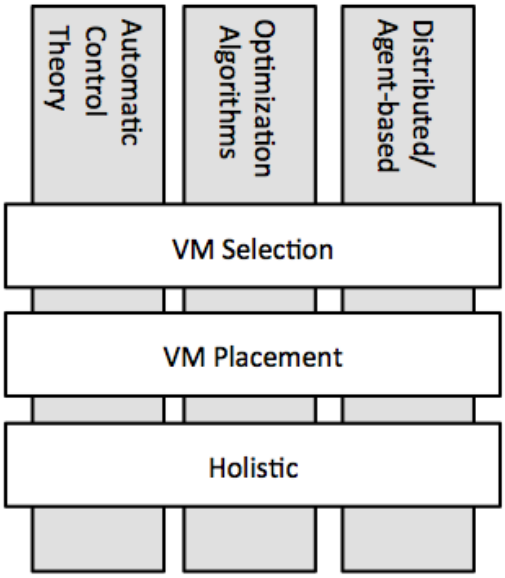


Figure 2.1: Classification of the main works about VM management in single datacenters.

- works based on *optimization algorithms*: given the current configuration and resource utilization, they can determine the best VM relocation;
- *distributed/agent-based* solutions, able to manage the cloud datacenter without centralized control.

In a cloud scenario, there are a lot of other related topics that must be taken into account while applying a VM reallocation plan. For example, VMs allocated on the same physical host can be completely independent and unaware of each other, or can collaborate to reach a common objective. For this reason, the infrastructure management may also take into account other challenges related to traffic-aware and memory-sharing placement of VM [27, 28].

Furthermore, to save power is important to predict the energy consumption of a single VM. While power metering of a physical host is quite simple, a lot of work remains to do to understand exactly how much energy is consumed by a running VM [29, 30].

In the following section, we deepen the state-of-the-art classification by providing some examples of solution to the VM management problem.

2.1.1 Automated Control-inspired Approaches

Some approaches to VM management are based on automated control theory, treating the infrastructure as if it was a system of sensors and actuators. The goal is to maintain the cloud in a condition of equilibrium, in which the power consumption is minimized, while limiting the SLA violations. Therefore, these approaches need a method to estimate the power consumption and a specific feedback control policy to keep the infrastructure utilization levels inside two threshold values.

Many works on feedback-controlled adaptive resource provisioning assume a central controller that combines application control and arbitration policy (e.g., [31–33]). Urgaonkar et al. [33] use queueing theory to model a multi-tier application and to determine the amount of physical resources needed by the application. Soundararajan et al. [32] present control policies for dynamic provisioning by replication of a database server, while Lim et al. [13] address the challenge of building an effective controller as a customer add-on outside of the cloud utility service itself. Such external controllers must function within the constraints of the utility service Application Programming Interfaces (APIs). It is important to consider techniques for effective feedback control using cloud APIs, as well as how to design those APIs to enable more effective control. The work by Lim et al. [13] especially explores *proportional thresholding*, a policy enhancement for feedback controllers that enables stable control across a wide range of guest cluster sizes using the coarse-grained control offered by popular virtual compute cloud services as Amazon Elastic Compute Cloud [6], Aptana Cloud [7] and Joyent [34].

While Soundararajan [32] approach is based on static thresholds with a target range, Padala and Lim [13, 35] dynamically adjust the CPU entitlement of a VM to meet QoS goals by empirically modeling the relationship of CPU entitlement and utilization to tune the parameters of an integral control.

Static thresholding is indeed simple to use, but it may not be robust to scale. Consider a simple motivating scenario in which a small startup company runs a web application service, e.g., a Tomcat [36] server cluster that serves dynamic content to clients. Rather than purchasing its own infrastructure to run its service, the company leases a slice of resources from a cloud hosting provider to reduce capital and operating costs.

The application is horizontally scalable: it can grow to serve higher request loads by adding more servers. Since the Tomcat cluster is request-balanced, going from 1 to 2 machines can increase capacity by 100% but going from 100 to 101 machines increases capacity by not more than 1%. The relative effect of adding a fixed-sized resource is not constant, so using static threshold values may not be appropriate.

Furthermore, considering the derivative component of a feedback control to dynamically vary the thresholds [37], can give an idea of the speed of change in the amount of resource requests.

The feedback-controlled adaptive resource provisioning can also take advantage of an integral component ([13, 35]) to add memory in the feedback loop and to prevent oscillations in resource provisioning.

2.1.2 Optimization algorithms

In the cloud scenario, a fundamental goal is to determine the best allocation for a collection of VMs on a datacenter composed by many hundreds or even thousands physical servers, in a way that the number of active hosts is minimized to save power.

The problem of finding the optimal allocation of VMs to hosts can be seen as a Bin Packing Problem with variable bin sizes and prices and, therefore, is known to be NP-hard [38].

A lot of works is now concentrating on the possibility to provide the virtual infrastructure with the ability of quickly calculate a sub-optimal solution to the bin packing problem applied on the cloud scenario. These solutions are intrinsically centralized because the state of each host of the datacenter is supposed to be known by a central cloud manager responsible for the computation of the best relocation plan.

Beloglazov et al. [39] faced the VM placement problem developing a very simple and effective heuristic. They apply a modification of the Best Fit Decreasing (BFD) algorithm that is shown to use no more than (11/9 OPT+1) bins (where OPT is the number of bins given by the optimal solution) [40]. In the Modified Best Fit Decreasing (MBFD) algorithm [39] , the authors sort all VMs in decreasing order of their current CPU utilizations, and allocate each VM to a host that provides the least increase of power consumption due to this allocation. This allows leveraging the heterogeneity of resources by

choosing the most power-efficient nodes first. The complexity of the allocation part of the algorithm is $O(nm)$, where n is the number of VMs that have to be allocated and m is the number of hosts.

When a host is detected to be in a critical overloaded condition because of the *oversubscription* of resources, some SLA violation can occur and the VMs running on that server generally experience performance degradation. To face this critical situation, some VMs must be migrated to another host. The selection of the best VM to migrate can be done again with an approach involving optimization research.

Beloglazov et al. [39] addressed this problem with the Minimization of Migrations (MoM) algorithm that sorts the list of VMs in the decreasing order of the CPU utilization. Then, it repeatedly looks through the list of VMs and finds a VM that is the best to migrate from the host. The best VM is the one that satisfies two conditions. First, the VM should have the utilization higher than the difference between the host's overall utilization and the upper utilization threshold. Second, if the VM is migrated from the host, the difference between the upper threshold and the new utilization is the minimum across the values provided by all the VMs. If there is no such a VM, the algorithm selects the one with the highest utilization, removes it from the list of VMs, and proceeds to a new iteration. The algorithm stops when the new utilization of the host is below the upper utilization threshold. The complexity of the algorithm is proportional to the product of the number of over-utilized hosts and the number of VMs allocated to these hosts.

In [17], authors suggest two other heuristic for VM selection problem: the Minimum Migration Time (MMT) policy and the Maximum Correlation (MC) policy.

The MMT policy migrates a VM that requires the minimum time to complete a migration relatively to the other VMs allocated to the host. The migration time is estimated as the amount of Random Access Memory (RAM) utilized by the VM divided by the spare network bandwidth available for the host.

The MC policy is based on the idea proposed by Verma et al. [41]. The idea is that the higher is the correlation between the resource usage by applications running on an oversubscribed server, the higher is the probability of server overloading. According to this idea, authors select those VMs to be migrated that have the highest correlation of the CPU utilization with

other VMs. To estimate the correlation between CPU utilizations by VMs, they apply the multiple correlation coefficient [42]. It is used in multiple regression analysis to assess the quality of the prediction of the dependent variable. The multiple correlation coefficient corresponds to the squared correlation between the predicted and the actual values of the dependent variable. It can also be interpreted as the proportion of the variance of the dependent variable explained by the independent variables.

Some works in the field of management of a virtual cloud infrastructure focus on both VM selection and placement simultaneously. An example is the work by Jung et al. [12] that presents Mistral, a controller framework able to optimize power consumption, performance benefits, and the transient costs incurred by various adaptations and the controller itself to maximize overall utility. Mistral can handle multiple distributed applications and large-scale infrastructures through a multi-level adaptation hierarchy and scalable optimization algorithm.

The idea is exploit a *Performance-Power optimizer* to find the optimal capacities of VMs that can be packed in as few server machines as possible while balancing performance and power usage. It employs a heuristic bin-packing algorithm with a classic gradient-based search algorithm to place VMs to hosts, but extends the algorithm to deal with variable number of active physical machines and their power consumption.

Furthermore Mistral takes into account a workload prediction to deal with frequent changes in resource requests, and the cost determined by the stated adaptation, considering even the computational overhead generated by the algorithm itself. The framework online builds an oriented graph where the vertex are cloud configuration and the arches are the costs of the adjustments necessary to go from a configuration to another. A configuration can be either "intermediate" or "candidate". A candidate satisfies the allocation constraint that the sum of all VMs's CPU and memory capacities on each host must be less than 100%, while an intermediate does not satisfy the constraint. For instance, Mistral may assign more CPU capacity to a VM than available on a host. When a candidate c_i is determined to be the final optimal configuration, the shortest path starting from the initial configuration c to configuration c_i denotes the optimal sequence of adaptation actions needed to achieve optimal utility for a given control window.

As all the holistic solutions, Mistral framework is intrinsically very complex and can be applied only to a centralized environment because it presumes that a single physical server receives the state of the whole system and operate to optimize it.

Nevertheless, we must notice the all that existing solutions exploiting optimization theory focus on a centralized architecture, requiring global knowledge of the datacenter state. Given the large scale and highly dynamic nature of the scenario, a centralized solution is unlikely to scale to meet realistic demands. In the following section we discuss some examples of distributed approaches to the VM management problem.

2.1.3 Distributed/agent-based approaches for VM management

Concentrating in a single physical machine all the responsibilities for finding the best way to relocate all the VMs of a cloud datacenter can lead to poor solutions in terms of scalability and reliability. A possible answer is to provide the infrastructure with a set of replicated servers which can substitute the central controller in case of fault or can work together with it in case the number of nodes is too high for a single datacenter manager. Both this solutions requires to face non trivial coordination challenges and do not solve the problem in an absolute way, but only provide solutions for a maximum number of nodes in the cloud.

Decentralized management solutions try to address both scalability and reliability problems by spreading the control of VM allocation and migration decisions to all the physical nodes. The idea is to give each node a set of simple operation to execute, thus to provide the whole system with an emergent intelligent behavior.

The work of Babaoglu et al. [25] is one of the most complete in this field. It is based on a simple gossip protocol [43] that does not require any central coordinator or global shared data structure and is completely VM and application agnostic. It does not require any instrumentation of either the VMs or the hosted applications. The distributed algorithm is executed periodically to identify a new arrangement of existing VM instances so that the number of empty servers is maximized. Once the new allocation has been identified, it is possible to

migrate VMs to their final destination using the live migration feature provided by most VM monitors.

An overlay network is built and maintained using a peer sampling service which provides each node with peers to exchange information with. The peer sampling service is implemented as follows: each node periodically sends its local view (i.e., list of VMs currently hosted) to K neighbors (i.e., other peers hosting VMs), and builds a new local view by merging the old one with those received by neighbors. Thus Babaoglu implicitly assumes each node of the datacenter can easily communicate with each other.

Each physical server has a map H_i of the current VM allocated, a Passive and an Active Thread running on it.

The Active Thread is executed each δ time units; iterates over each neighbor j, to which it sends the current number of hosted VMs; it then receives an updated map H_i of allocation (which is possibly different from the current one), and updates it accordingly. Each server must keep track of the initial location of each VM it receives, so that at the end it can pull the assigned VMs from their original location.

The Passive Thread listens for messages coming from the other peers. Upon receiving the allocation map H_j from peer j, the server decides whether some VMs should be pushed to j, or pulled from it. VMs are always transferred from the least loaded peer to the most loaded one; the number of VMs to transfer is limited by the residual capacity of the receiving node.

This work does not address the important issues of deciding which specific VM to migrate; probably it makes sense to transfer those with smaller memory footprint, so that the transferred image is smaller.

Since neighborhoods of physical servers are always changing during the protocol execution, the information about needed migrations are spread across the datacenter environment, without a central management node required.

However, this approach does not take into account that in a physical cloud datacenter the network is typically organized in a tree configuration, with high level nodes prone to be bottleneck in an intra-cloud communication. Thus, making every node able to exchange information with each other may congestionate the network.

The same issue can be pointed out in the work by Tighe et al. [44] where an overloaded host determines the minimum

amount of CPU required to be available on another host to migrate out one of its VMs and relieve the stress situation. To this purpose, the node broadcasts a resource request message to all other hosts.

In [44], each host monitors its resource utilization on a periodic interval, every 5 minutes, and performs a check for stress or under-utilization by comparing average CPU utilization over the last 5 monitoring intervals with threshold values. Taking into account the past load conditions can give to the monitoring system an idea of how much fast the resource utilization is changing, thus recalling the advantages of a derivative component in an automated controller.

An important element of Tighe study [44] is the idea of reducing thrashing between highly utilized hosts, by implementing a *relocation freeze*, preventing a host from offering resources for a specified amount of time after the same host evicts a VM. Indeed, this mechanism can give memory to the system similarly to the integral component in a feedback automated controller and prevent oscillations between provisioning and de-provisioning of physical hosts.

We took inspiration from distributed management solutions to build an alternative decentralized protocol of interaction between physical hosts [45]. The goal is to find the best interaction that maximize the information exchanged without creating communication overhead and VM performance degradation.

We propose a novel decentralized way [46] to apply a VM migration policy to the cloud: we imagine the datacenter is partitioned into a collection of overlapping neighborhoods, in each of which a local reallocation strategy is applied. Taking advantage from the overlapping, the VM redistribution plan propagates on the global cloud.

We analyze the effects of this approach by comparing them with the centralized application of the same policy. In particular we focus on the definition of the Distributed Autonomic Migration (DAM) protocol, used by cloud's physical hosts to communicate and get a common decision as regards the reallocation of VMs, according to a predefined global goal (e.g. power-saving, load balancing, etc.) [47].

Table 2.1 summarizes the main features of the referred works in the field of VM management. The last two lines refer to our contributions presented in the next section.

Table 2.1: Main features of the works in the field of VM management

Contribution	Addressed problem	Goal	Employed technique	Threshold mechanism
Urgaonkar et al. [33]	Cluster commensuration (number of VMs to provide for application)	QoS	Queuing theory	-
Soundararajan et al. [32]	Cluster commensuration	QoS	Control theory	proportional/static
Lim et al. [13]	Holistic	QoS	Control theory	derivative and integral
Padala et al. [35]	VM commensuration	QoS	Control theory	derivative and integral
Loreti [37]	Holistic	QoS	Control theory	derivative
Abawajy et al. [39]	Holistic	power saving & load balancing	Heuristic	static
MMT-Beloglazov et al. [17]	VM Selection	min migr. time	Heuristic	-
MC-Beloglazov et al. [17]	VM Selection	load balancing	Heuristic	-
Jung et al. [12]	Holistic	power saving & load balancing	Heuristic	-
Marzolla et al. [25]	Holistic	power saving	Agents	static
Tighe et al. [44]	Holistic	power saving & load balancing	Agents	pseudo-derivative
Loreti et al. [48]	Holistic	load balancing	Agents	tunable
Loreti et al. [49]	Holistic	power saving & load balancing	Agents	tunable

2.2 THE MODEL

We propose a distributed solution for Cloud Computing infrastructure management, with a special focus on VM migration. A detailed description of the work presented in this chapter was published in [49]. According to the model, each physical node of the system is equipped with a software module composed of three main layers (see Figure 2.2):

- the infrastructure layer, specifying a software representation of the cloud's entities (e.g., hosts, VMs, etc);
- the coordination layer, implementing the DAM protocol, which defines how physical hosts can exchange their status and coordinate their work;
- the policy layer, containing the rules that every node must follow to decide where to possibly move VMs.

The separation between coordination and policy layer allow us to use the same interaction model with different policies. In this way, different goals can be achieved by only changing the adopted policy, while the communication model remains the same. We describe each layer in more detail in the following sections.

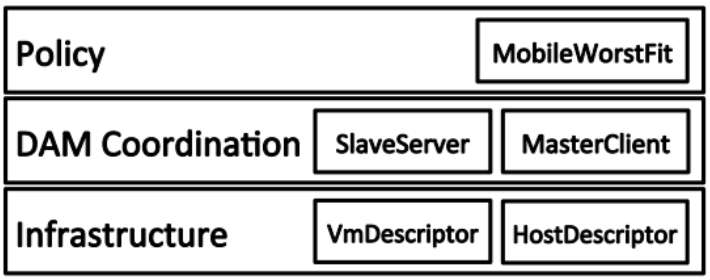


Figure 2.2: The three tiers architecture. The separation between layers ensures the possibility to test different policies and protocols with the same infrastructure implementation.

2.2.1 Infrastructure Layer

The infrastructure layer defines which information must be collected about each host's status. To this purpose two basic structures are maintained: the *HostDescriptor* and the *VmDescriptor*.

The *HostDescriptor* can be seen as a bin with a given capacity able to host a number of VMs, each one with a specific request for computational resources. We only take into account the amount of computational power in terms of MILLIONS OF INSTRUCTIONS PER SECOND (MIPS) offered by each host and requested by a VM. An empty *HostDescriptor* represents an idle server that can therefore be put in a *sleep* mode or switched-off to save power.

The *HostDescriptor* contains not only a collection of *VmDescriptor*s really allocated on it (the *current map*), but also a temporary collection (the *future map*) initialized as a copy of the real one and exchanged between hosts according to the defined protocol. During interactions only the temporary copy is updated and, when the system reaches a common reallocation decision, the *future map* is used to apply the migrations.

In a distributed environment, where each node can be aware only of the state of a local neighborhood of nodes, the number of worthless migrations can be very high. Thus, this double-map mechanism is used to limit the number of migrations (as we describe in Chapter 2.2.2), by performing them only when all the hosts reach a common distributed decision.

Each VM is also equipped with a migration history keeping track of all the hosts where it was previously allocated. For the sake of simplicity, we assume that a VM cannot change its CPU request during the simulation period.

The CPU model

The amount U_h of CPU MIPS used by the host h is calculated as follows:

$$U_h = \sum_{vm \in currentVmMap_h} m_{vm} \frac{T_{vm}}{100} \tag{2.1}$$

where $currentVmMap_h$ is the set of VMs currently allocated on host h; T_{vm} is the total CPU MIPS that a virtual machine vm can request; and m_{vm} is the percentage of this total that is currently used.

Indeed, we consider a simplified model in which the total MIPS executed by the node can be seen as the sum of MIPS used by each hosted virtual machine. In fact, the model

does not take into account the power consumed by the physical machines to realize virtualization and to manage their resources.

2.2.2 Coordination Layer

The coordination layer implements the DAM protocol, which defines the sequence of messages that hosts exchange in order to get a common migration decision and realize the defined policy.

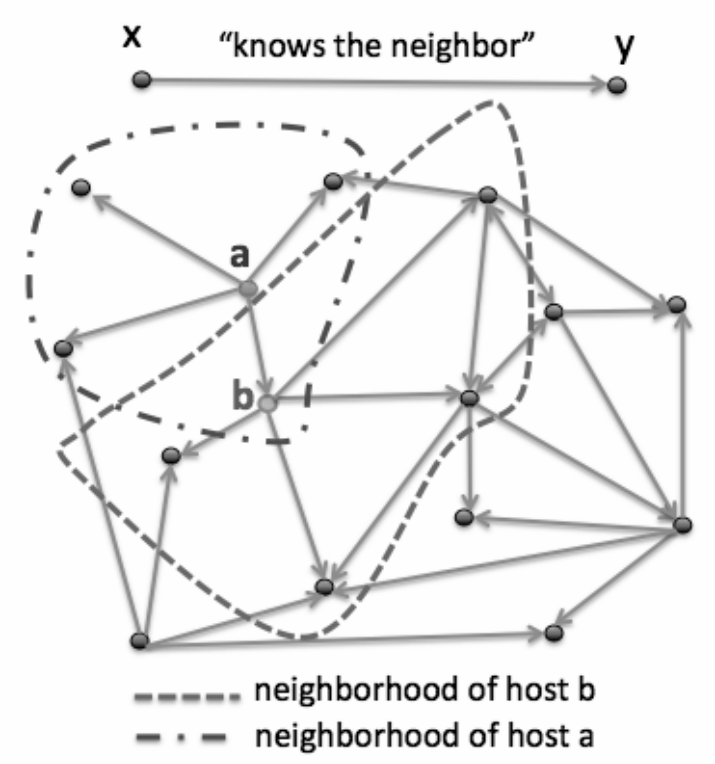


Figure 2.3: Example of *"knows the neighbor"* relation applied on a collection of physical nodes. The relation is not symmetric, thus if node *"a knows the neighbor b"*, this means that b is included in the neighborhood of a but, in general, a is not in the neighborhood of b.

The protocol is based on the assumption that the cloud is divided into a predefined fixed collection of overlapping subsets of hosts: we call each subset a *neighborhood*. From an operational point of view, we define a *"knows the neighbor"* relation between the hosts of the datacenter, which allow us to partition the cloud into overlapping neighborhoods of physical

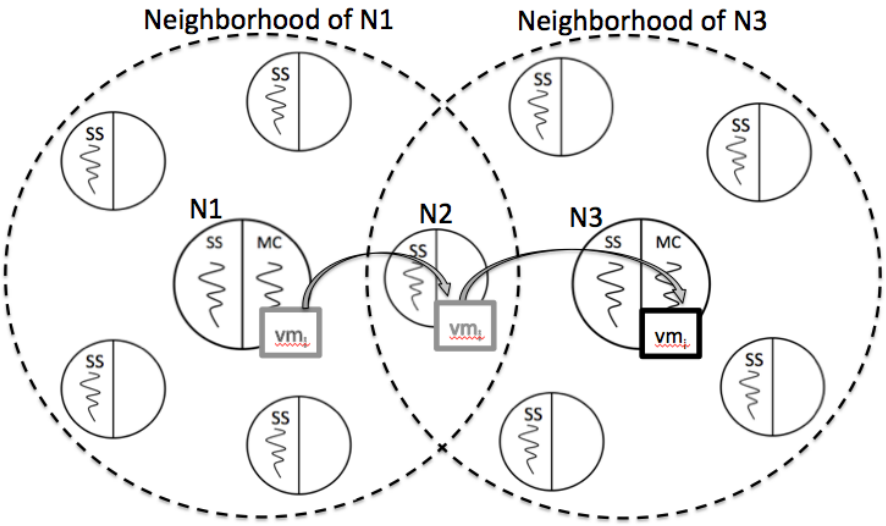


Figure 2.4: Schema of two overlapping neighborhoods. The VM descriptor vm_i is exchanged across physical hosts, crossing the neighborhood boundaries, until the nodes agree with a common reallocation plan i.e., a "stable" allocation hypothesis for vm_i is detected.

machines. As we can see in Figure 2.3 the relation is not symmetric.

We assume that each physical host executes a daemon process called *SlaveServer*, which owns a copy of the node's status stored into an *HostDescriptor* and can send it to other nodes asking for that.

Each node can monitor its computational load and the amount of resources used by the hosted VMs; according to the chosen policy, it can detect either it is in a critical condition or not. A node can, for example, detect to be overloaded, risking to incur in SLA's violations, or underloaded, causing possibile power waste. If one of these critical conditions happens, the node starts another process, the *MasterClient*, to actually make a protocol interaction begin. We call *rising condition* the one that turns on a node's *MasterClient*.

Since there is a certain rate of overlapping between neighborhoods, the effects of migrations within a neighborhood can cause new rising conditions in adjacent ones.

To better explain the DAM protocol, Figure 2.4 shows an example of two overlapping neighborhoods. Each node has a *SlaveServer* (SS in Figure 2.4) always running to answer questions from other node's *MasterClient* (MC in Figure 2.4), and optionally can also have a *MasterClient* process started to handle a local critical situation. A virtual machine vm

allocated to an underloaded node N1 can be moved out of it on N2 and, as a consequence of the execution of the protocol in the adjacent neighborhood of N3, it can be moved again from N2 to N3. It is worth to notice that node N2, as each node of the datacenter, has its own fixed neighborhood, but it starts to interact with it (by means of a *MasterClient*) only if a *rising condition* is observed.

Note that N1's *MasterClient* must have N2 in its neighborhood to interact with it, but the *SlaveServer* of N2 can answer to requests by any *MasterClient* and, if a critical situation is detected (so that N2 *MasterClient* is started) its neighborhood does not necessarily include N1.

As regards this environment, we must remark that the migration policy should be properly implemented in order to prevent never-ending cycles in the migration process.

Algorithms 1 and 2 reports the interaction code executed by the *MasterClient* and the *SlaveServer*, respectively. The *MasterClient* procedure takes as input the list of *SlaveServer* neighbors *ssNeighList* and an integer parameter *maxRound*. The *SlaveServer* procedure takes the *HostDescriptor h* of the node on which it is running. If the *SlaveServer* detects a critical conditions on the host, makes a *MasterClient* process start (lines 1-2 in Algorithm 2).

We must ensure that the neighbors's states the *MasterClient* obtains, are consistent from the beginning to the end of the interaction. For this reason, a two-phase protocol is adopted:

DAM *Phase 1*

As we can see in lines 5-9 of Algorithm1, the *MasterClient* sends a message to all the *SlaveServers* neighbors *ss* to collect their *HostDescriptors h*. This message also works as a *lock* message: when the *SlaveServer* receives it, locks his state, so that no interactions with other *MasterClients* can take place (lines 5-11 in Algorithm 2). If a *MasterClient* sends a request to a locked *SlaveServer*, simply waits for the *SlaveServer* to be *unlocked* and to send its state.

DAM *Phase 2*

The *MasterClient* compares all the received neighbor's *HostDescriptors* with a previous copy he stored (line 10 in Algorithm 1). If the *future map* is changed, performs phase 2A,

ALGORITHM 1: MasterClient DAM protocol code

```
   input : maxRound, ssNeighList
1  neighHDs ← emptylist();
2  neighHDsPast ← emptylist();
3  round ← 0;
4  while true do
5      foreach ss ∈ ssNeighList do                    /* Phase 1 */
6          send(ss,"lock");
7          (h,ss) ← receive();
8          neighHDs.add(h);
9      end
10     if neighHDs = neighHDsPast then                /* Phase2 */
11         round++;
12     else
13         round ← 0;
14         neighHDsPast ← neighHDs;
15     end
16     if round < maxRound then                       /* Phase 2A */
17         optimize(neighHDs);
18         foreach ss ∈ ssNeighList do
19             send(ss,neighHDs.get(ss));
20         end
21     else                                           /* Phase 2B */
22         foreach ss ∈ ssNeighList do
23             send(ss,"update_current_status");
24         end
25         break;
26     end
27 end
```

otherwise increments a counter and, when it exceeds a certain maximum, performs phase 2B:

- *Phase 2A:* the *MasterClient* computes a VM reallocation plan for the whole neighborhood, according to the defined policy, and sends back to each *SlaveServer* neighbor the modified *HostDescriptor* (lines 17-20 in Algorithm 1). The "optimize(neighHDs)" operation in line 17 of Algorithm 1 actually applies the specific chosen policy on the neighborhood's *HostDescriptors* (neighHDs). Indeed, this method is the software connection between the coordination layer and the policy layer.

Algorithm 2: SlaveServer DAM protocol code

```
   input :h
1  if checkRisingCondition() then
2      startMasterClient();
3  end
4  while true do                                   /* Phase 1 */
5      (msg, mc) ← receive();
6      if msg ≠ "lock" then                  /* protocol error */
7          break;
8      else
9          lock();
10         send(mc, h);
11     end
12     (item, mc) ← receive();                     /* Phase 2 */
13     if item ≠ "update_current_status" then
                                                  /* Phase 2A */
14         h ← item;
15     else                                       /* Phase 2B */
16         updateCurrentStatus(h);
17     end
18     unlock();
19     if checkRisingCondition() then
20         startMasterClient();
21     end
22 end
```

As we can see in line 14 of Algorithm 2, the state is accepted passively by the slaves, without negotiation. The migration decisions only change the *future map* of VM allocation. No host switch-on/off or VM migration is performed in this phase. After all new states are sent, the *SlaveServers* are *unlocked* (line 18 in Algorithm 2) and the *MasterClient* begins another round of the protocol interaction by restarting phase 1.

- *Phase 2B:* when the number of round with unchanged neighbor's allocation exceeds a defined maximum (line 16 of Algorithm 1), the *MasterClient* sends an *update-current-status* request (line 23 of Algorithm 1) to all *SlaveServers* and terminates. This last message notifies the *SlaveServers* that information inside the *HostDescriptor* should be applied to the real system state (line 16 of

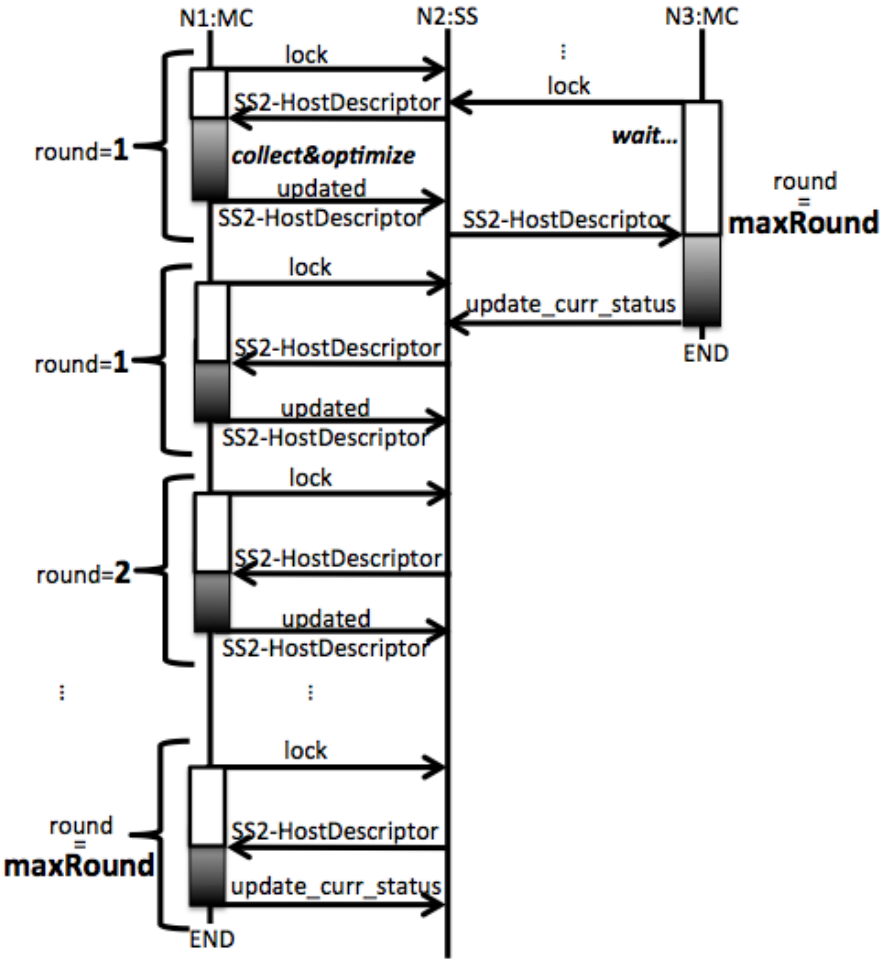


Figure 2.5: Example of protocol interaction rounds. Node N2 is shared by nodes N1 and N3. Therefore, their *Master-Clients* must coordinate to ensure the consistency of status information.

Algorithm 2). The *SlaveServer* again executes it passively and unlocks his state.

Alternatives 2A and 2B come from the need for reducing the number of migration physically performed. Looking at example in Figure 2.4, if hosts only exchange and update the current collection of VMs, every *MasterClient* can only order a real migration at each round, so that vm_i on N1 would be migrated on N2 at first, and later on N3. Using the temporary *future map* (initially copied from the real one) and performing all the reallocations on this abstract copy, real migration are executed only when the N3's *MasterClient* exceeds a maximum number of rounds and vm_i can directly go from N1 to N3.

The same example is represented in Figure 2.5. N2 is shared by the *MasterClient*s of nodes N1 and N3. Two concurrent sessions of the protocol must synchronize in order to maintain the status information consistent. Therefore, node N3 waits until N2 status is updated and released by N1. If no concurrent interactions are taking place in adjacent neighborhoods, the *MasterClient* receives an unchanged *HostDescriptor* and increments the value of the *round* counter.

As a result of DAM protocol, the consensus on migration of VMs is not for the entire infrastructure, but is distributed across the neighborhoods. This element must be taken into account while implementing the policy layer.

2.2.3 Policy Layer

Depending on the algorithm implemented in the Policy layer, different management goals can be achieved.

We used DAM infrastructure to study the performance of two different policies: Mobile Balance (MB) and Mobile Worst Fit (MWF). The following sections show the policies in detail as well as their performance evaluation.

2.3 MOBILE BALANCE POLICY

MB is a novel policy aiming to maintain the computational load of physical nodes balanced. The content of this chapter has been published in [48].

MB exploits a fixed threshold FTH_UP and a dynamic threshold MTH_UP to detect rising conditions. The fixed threshold identifies absolute risky situations: if the host is more loaded than FTH_UP, SLA violations may occur. The dynamic threshold MTH_UP represents the upper value that cannot be exceeded in order to maintain the neighborhood balanced.

According to the DAM coordination protocol, at each iteration the *MasterClient* collects the temporary VM allocation map of the neighbors and executes a MB optimization as detailed in Algorithm 3: the *MasterClient* calculates the average of resource utilization in his neighborhood (`calculateNeighAverage()` in line 1 of Algorithm 3) and uses it to compute the dynamic threshold (MTH_UP) by adding a tolerance interval t (line 2 of

Algorithm 3: MB policy

input: h, t, FTH_UP

```
1 ave ← calculateNeighAverage() ;
2 MTH_UP ← ave + t ;
3 u ← h.getLoad() ;
4 if u > FTH_UP or u > MTH_UP then
5     vmList ← selectVms() ;
6 end
7 if vmList.size ≠ 0 then
8     worstFitMigrateAll(vmList) ;
9 end
```

Algorithm 3). Then the *MasterClient* checks its *HostDescriptor* h and collects the current computational load u by invoking a specific getLoad() method on the *HostDescriptor* (line 3 of Algorithm 3).

The computational load u of the host is compared to fixed and dynamic thresholds: if it is detected to be higher than FTH_UP or MTH_UP, then the selectVms() operation is invoked to pick (from the host h temporary state) only the less loaded VMs whose migration will result in the host load to go back under both MTH_UP and FTH_UP. selectVm() applies the MoM algorithm from Beloglazov et al. [39] and is detailed in Algorithm 4. Differently from [39], we select the threshold thr as the minimum between FTH_UP and MTH_UP.

The list of chosen VMs vmList is finally migrated to neighbors by means of a modified worst-fit policy (worstFitMigrateAll(vmList) in line 8 of Algorithm 3).

As shown in Algorithm 5, the worstFitMigrateAll procedure takes as input the list of vm to move (vmList), the host h where they are currently allocated, the list offNeighList of switched-off hosts in h's neighborhood (if any) and wNeighList of all the working neighbors of h. The procedure considers the VMs by decreasing CPU request and, according to the principles of worst-fit algorithm, tries to migrate it to the neighbor n with the highest value of free capacity (lines 5 to 13 of Algorithm 5). This ensure that neighbors with low CPU utilization are preferred.

Finally, if no hosts in wNeighList can receive vm, but h is more loaded than FTH_UP, then h is in a risky

situation because performance degradation can occur. Thus, a switched-off neighbor is woken up (line 15 of Algorithm 5).

worstFitMigrateAll(*vmList*) operates in a "all-or-none" way, such that the migrations are committed on the future maps (line 22 of Algorithm 5) only if it is possible to reallocate all the VMs in the list (i.e., without making other hosts to exceed FTH_UP), otherwise no action is performed (line 17 of Algorithm 5).

As shown in Figure 2.6, suppose that a protocol execution by h_b's *MasterClient* decides to migrate to h_b a VM vm_i currently allocated on h_c. When the *SlaveServer* of h_b is unlocked, the

Algorithm 4: selectVms() procedure

input : h, MTH_UP, FTH_UP
output: vmsToMove

```
1  u ← h.getLoad();
2  vmList ← h.getFutureVmMap() ;
3  vmList.sortDecreasingLoad();
4  minU ← ∞;
5  bestVm ← null;
6  thr ← min{FTH_UP, MTH_UP};
7  vmsToMove = emptyList();
8  while u > thr do
9      foreach vm ∈ vmList do
10         var ← vm.getLoad() − u + thr;
11         if var ⩾ 0 then
12             if var < minU then
13                 minU ← var;
14                 bestVm ← vm;
15             end
16         else
17             if minU = ∞ then
18                 bestVm ← vm;
19             end
20             BREAK;
21         end
22     end
23     u = u − bestVm.getLoad();
24     vmsToMove.add(bestVm);
25     vmList.remove(bestVm);
26 end
```

Algorithm 5: worstFitMigrateAll() procedure

```
input : vmList, h, offNeighList, wNeighList
1  vmList.sortDecreasingLoad() ;
2  foreach vm ∈ vmList do
3      vmU ← vm.getLoad();
4      maxAvail ← 0; bestHost ← null;
5      foreach n ∈ wNeighList do
6          if n ∉ vm.getMigrationHistory() then
7              avail ← FTH_UP − n.getLoad() + vmU ;
8              if avail > maxAvail then
9                  maxAvail ← avail ;
10                 bestHost ← n ;
11             end
12         end
13     end
14     if bestHostnull and !empty(offNeighList) and
       u > FTH_UP then
15         bestHost ← offNeighList.get(0) ;
16     else
                                  /* all-or-none behavior */
17         migrationMap ← null ;
18         break;
19     end
20     migrationMap.add(vm, bestHost) ;
21 end
22 commitOnFutureMap(migrationMap) ;
```

policy execution on h_a's *MasterClient* can decide to put vm_i into h_a. Now if h_c has a *MasterClient* running, and decides to migrate vm_i back to h_c, then h_c can take the same decision as before and a loop in vm_i migration starts. If this happens, the distributed system will never converge to a common decision. In order to face this problem, the MB policy exploits the migration history inside each *VmDescriptor* to avoid loops in reallocation: a VM can be migrated only on a host that it never visited before. Once the distributed autonomic infrastructure reach a common decision, the migration history of each VM is deleted.

To prevent the case in which a VM cannot be migrated away because it visited all the nodes around her current allocation, we implemented the migration history as an ever-updating list

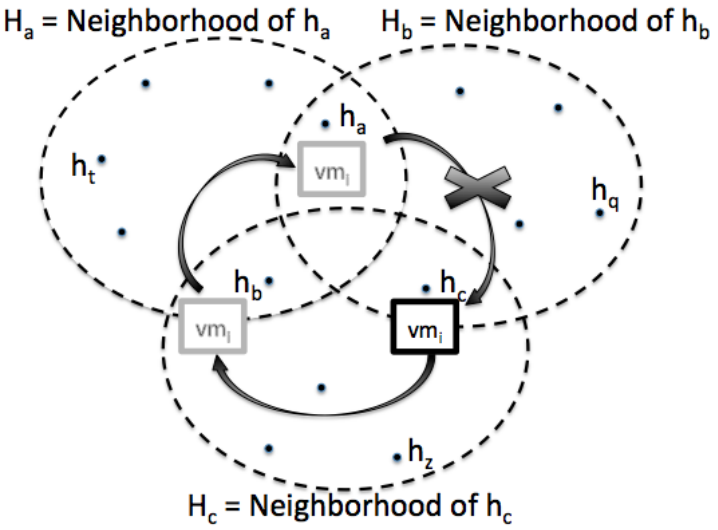


Figure 2.6: Example of three overlapping neighborhoods.

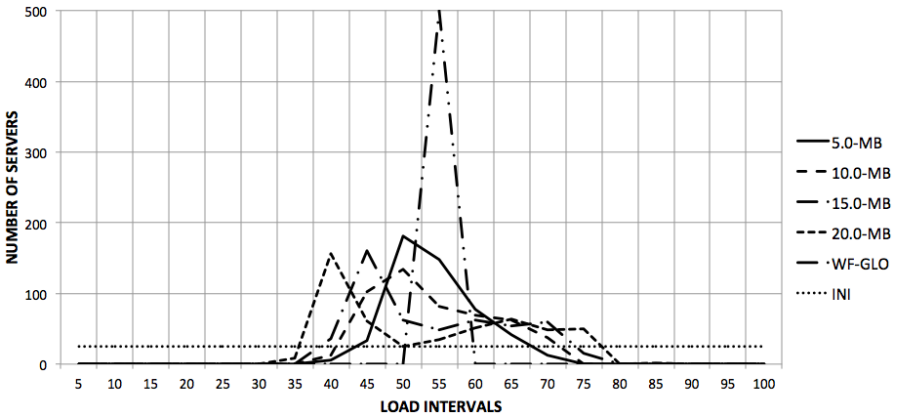
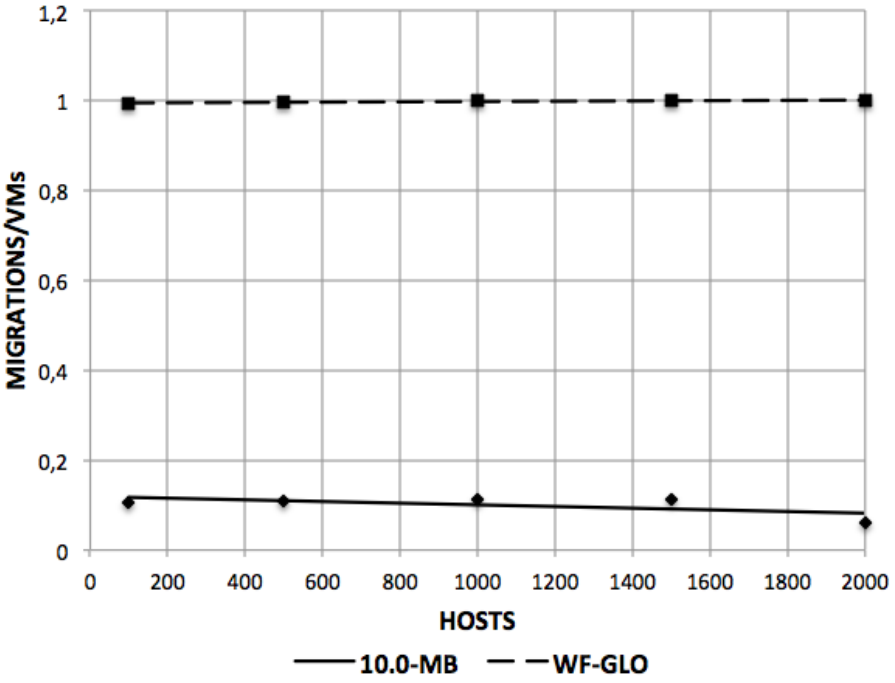


Figure 2.7: Distribution of servers on load intervals.

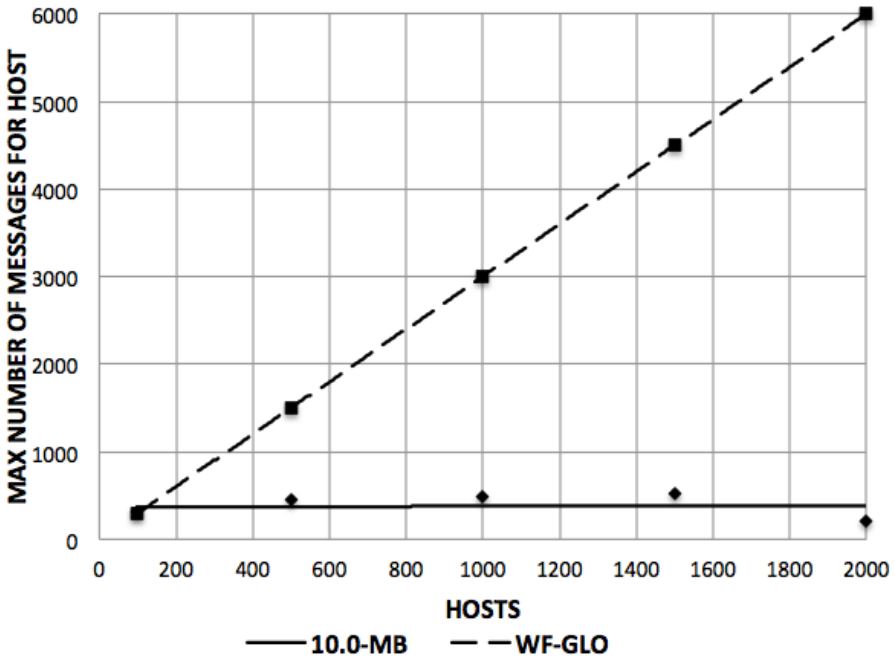
of Most Recently Used (MRU) hosts. This allow us to prevent the permanent blacklisting of available destination hosts and reduce the risk of thrashing in VM migration. Indeed, in order to completely avoid migration oscillations, we should provide a H-long MRU migration history inside each VM (where H is the total number of hosts in the cloud). Since this solution can considerably increase the number of messages exchanged, we accept the risk of migration cycles by reducing the size of the migration history inside each VM.

2.3.1 Experimental results

To understand the effectiveness of the proposed model we used DAM Simulator (DAM-Sim)[47]: a Java simulator able to



(a)



(b)

Figure 2.8: MB and WF-GLO performance comparison. 2.8a: Number of migration performed for increasing number of simulated hosts. We compare the performance of MB with tolerance interval 10.0 with centralized WF-GLO. 2.8b: Maximum number of messages exchanged (sent and received) by a single host of the datacenter. MB significantly outperforms WF-GLO for high values of simulated hosts.

apply a specific policy on a collection of neighborhoods throughDAM protocol and compare the performance with a centralized policy implementation.

We initially tested our approach on a simulated datacenter of 500 physical nodes hosting around 15000 VMs (i.e., an average value of 30 VMs on each host). We adopted a fixed neighborhood dimension of 10 physical server and set FTH_UP to 95%, while repeating the experiment with decreasing values of the tolerance interval t. The length of the migration history inside each VM is set to 10. This allow us to avoid up to ten-hops cycles in VM migration.

We always started from the worst situation for load balancing purposes, i.e., all the servers have different computational loads, resulting in a strongly unbalanced scenario. To understand the efficacy of our approach, we partitioned the set of hosts into 20 intervals of computational load (i.e., each interval is 5% wide).

As we can see in Figure 2.7, at the beginning (INI series) the system is configured such that there are 25 servers in each load interval. We apply the distributed MB policy with tolerance interval $t \in \{5.0, 10.0, 15.0, 20.0\}$ and compare the results with a centralized implementation of a worst-fit reallocation policy (WF-GLO in Figure 2.7).

According to worst-fit principals [50], WF-GLO considers the VMs by decreasing load request and always selects the host with the higher value of free capacity as migration destination. Relying on a global knowledge of the state of each server, the centralized WF-GLO policy can make each node loaded at 55% while the MB policy cannot perform as good. But, as we can see in Figure 2.7, we can significantly improve MB performance by reducing the tolerance interval t. Table 2.2 shows decreasing values of standard deviation when reducing t.

At the moment, the simulator is not able to give trustworthy results about execution time for distributed environments because the CPU executing the simulator code can only sequentialize intrinsically concurrent processes of the protocol. For this reason, no test about execution time is reported.

In order to test the scalability of the distributed approach, we analyzed MB behavior while increasing the number of simulated servers and VMs up to 2000 and 60000 respectively. Figure 2.8a shows the number of migrations stated by MB and compare it with that of WF-GLO policy. Since the number of VMs increases with the number of hosts, we actually compare

Table 2.2: Values of standard deviation σ of traces in Figure 2.7 for decreasing values of tolerance interval t.

t	σ
20	12,9716625
15	10,19978781
10	8,089993572
5	5,972641366

the ratio between migrations and number of VMs in the scenario. WF-GLO policies does not take into account the current allocation while performing the optimization, therefore it results in a very high number of migration, near to the total of VMs. Conversely MB distributed policy only operates on overloaded nodes (with CPU utilization higher than FTH_UP) or on those hosts that are unbalanced in respect to the average of their neighborhood (CPU utilization higher than MTH_UP). For this reason, as shown in Figure 2.8a, the number of resulting migrations is significantly lower for MB.

In Figure 2.8b we consider the maximum number of messages exchanged by a single host. Since WF-GLO is centralized, the coordinator node must collect the state of the all other nodes before starting the optimization and finally return the new configuration to each node. Therefore, as shown in Figure 2.8b, the number of messages exchanged by the coordinator is always proportional to the number of the nodes it manages. Conversely, the behavior of MB policy is rather constant. This comes from the fact that, according to MB, each node of the datacenter always comunicate with a predefined number of neighbors (10 in this simulation). Therefore, considering the maximum number of messages exchanged by a node, for high values of simulated hosts, we can conclude that MB distributed approach performs better than the centralized WF-GLO algorithm.

2.4 MOBILE WORST FIT POLICY

Given the good load balancing performance of MB, we enriched the policy with the ability of switching off the underloaded

hosts to save power. The result is MWF policy, which exploits two fixed thresholds (FTH_UP and FTH_DOWN) and two dynamic (mobile) thresholds (MTH_UP and MTH_DOWN) used to detect rising conditions. This policy is also depicted in [51]

Similarly to MB, the fixed thresholds identify risky situations: if the host is less loaded than FTH_DOWN an energy waste is detected, while, if the host is more loaded than FTH_UP, SLA violations may occur. The dynamic thresholds (MTH_UP and MTH_DOWN) represents the upper and lower values that cannot be exceeded in order to maintain the neighborhood balanced.

ALGORITHM 6: MWF policy

input : h, t, FTH_DOWN, FTH_UP

```
1  ave = calculateNeighAverage() ;
2  MTH_DOWN = ave − t ;
3  MTH_UP = ave + t ;
4  u = h.getLoad() ;
5  if u < FTH_DOWN or u < MTH_DOWN then
6      vmList = h.getFutureVmMap() ;
7  else if u > FTH_UP or u > MTH_UP then
8      vmList = selectVms() ;
9  end
10 if vmList.size ≠ 0 then
11     migrateAll(vmList) ;
12 end
```

As detailed in Algorithm 6: the *MasterClient* calculates the average of resource utilization in his neighborhood (*calculateNeighAverage()* in line 1 of Algorithm 6) and uses it to compute the two dynamic thresholds (MTH_DOWN and MTH_UP) by adding and subtracting a tolerance interval t (lines 2, 3 of Algorithm 6). Then the *MasterClient* checks its *HostDescriptor h* and collects the current computational load u by invoking a specific getLoad() method on the *HostDescriptor* (line 4 of Algorithm 6).

The computational load u of the host is compared to fixed and dynamic thresholds: if it is less than the lower thresholds, the *MasterClient* attempts to put the host in *sleep* mode by migrating all the VMs allocated; otherwise, if the host load exceeds the upper thresholds, only a small number of VMs

Algorithm 7: migrateAll() procedure

```
input : vmList, h, offNeighList, underNeighList,
        otherNeighList
1  vmList.sortDecreasingLoad() ;
2  foreach vm ∈ vmList do
3      vmU = vm.getLoad(); maxAvail = 0 bestHost = null;
4      foreach n ∈ otherNeighList do
5          if n ∉ vm.getMigrationHistory() then
6              avail = FTH_UP − n.getLoad() + vmU;
7              if avail > maxAvail then
8                  maxAvail = avail;
9                  bestHost = n;
10             end
11         end
12     end
13     if bestHost == null then
14         minU = ∞ ;
15         foreach n ∈ underNeighList do
16             if n ∉ vm.getMigrationHistory() then
17                 avail = FTH_UP − n.getLoad() + vmU;
18                 if avail >= 0 and avail < minU then
19                     minU = avail;
20                     bestHost = n;
21                 end
22             end
23         end
24     end
25     if bestHost == null and !empty(offNeighList) and
       u > FTH_UP then
26         bestHost = offNeighList.get(0) ;
27     else
                                  /* all-or-none behavior */
28         migrationMap = null ;
29         break;
30     end
31     migrationMap.add(vm, bestHost) ;
32 end
33 commitOnFutureMap(migrationMap) ;
```

are selected for migration. As we can see in lines 5, 6 of Algorithm6, if the computational load u is less than the fixed (FTH_DOWN) or the dynamic (MTH_DOWN) lower thresholds, all the VMs of the host are collected for migration into an array $vmList$. $h.getFutureVmMap()$ in line 6 is the method to collect the temporary allocation. Indeed in this phase, the policy only works on a copy of the real VM allocation map, because according toDAM protocol, all the migrations will be performed only when the whole datacenter reach a common decision.

As suggested by MB, if the load u is detected to be higher than the fixed (FTH_UP) or dynamic (MTH_UP) upper thresholds, then the $selectVm()$ operation is invoked (Algorithm 4).

The list of chosen VMs $vmList$ is finally migrated to neighbors by means of a modified worst-fit policy ($migrateAll(vmList)$ in line 11 of Algorithm 6). As shown in Algorithm 7, the $migrateAll$ procedure takes as input the list of vm to move ($vmList$), the host h where they are currently allocated, the list $offNeighList$ of switched-off hosts in h's neighborhood, the $underNeighList$ of h's neighbors with load level lower than FTH_DOWN, and $otherNeighList$ of all the other neighbors of h. The procedure considers the VMs by decreasing CPU request and, according to the principles of worst-fit algorithm, tries to migrate it to the neighbor n with the highest value of free capacity (lines 2-12 of Algorithm 7). If no neighbor in $otherNeighList$ can receive the vm, the $underNeighList$ is considered with a best-fit approach (lines 13-24 of Algorithm 7), thus allocating vm on the most loaded host of the list. This ensure that neighbors with CPU utilization near to FTH_DOWN are preferred, while less loaded ones remain unchanged and will be hopefully switched-off by other protocol's interactions. Finally, if neither hosts in $underNeighList$ can receive vm (e.g, because the list is empty), but h is more loaded than FTH_UP, then h is in a risky situation because SLA's violations can occur. Thus, as in MB, a switched-off neighbor is woken up (line 26 of Algorithm 7).

Similarly to $worstFitMigrateAll(vmList)$ of MB policy, $migrateAll(vmList)$ operates in a "all-or-none" way (line 33 of Algorithm 7).

2.4.1 Experimental results

To understand the effectiveness of MWF, we tested it on DAM-SIM [52]. In this case, we evaluated the approach on a set of 100 physical nodes hosting around 3000 VMs (i.e., an average value of 30 VMs on each host), repeating every experiment with an increasing average load on each physical server.

According to the tuning tests of MoM algorithm [17], the FTH_DOWN and FTH_UP thresholds have been fixed at 25% and 95%, respectively, while we initially set the tolerance interval t for load balancing at 8%.

We start from the worst situation for power-saving purposes, i.e., all the servers are switched on and have the same computational load within the fixed thresholds. To make the DAM protocol start we need some lack of balance in the datacenter, so we forced 20 hosts to be more loaded and 20 hosts to be less loaded than the datacenter average value. These hosts are randomly chosen in every experiment.

In Figure 2.9, we compare the MWF performance with 5 and 10 nodes in each neighborhood (MWF5 and MWF10 series), with the application of a centralized best fit policy (BF-GLO in Figure 2.9a) [47]. We also show the performance of BF: a best fit policy applied in a distributed way by means of DAM protocol. BF exploits the two-phase lock protocol, therefore, each time a server detects to be underloaded or overloaded, it start reconsidering the current VM allocation for the whole neighborhood. Details about BF implementation can be found in [47].

Figures 2.9a and 2.9b show the number of servers switched on at the end of the MWF and BF executions. As we expected, the DAM protocol cannot perform better than a global algorithm. Indeed, the global best fit policy (BF-GLO) can always switch off a higher rate of servers resulting in the lower trend. Furthermore, as regards the power saving objective, we can see that BF perform better than MWF for all the selected neighborhood dimensions. This comes from the different objectives of the two policies: MWF tries to switch-off the initially underloaded servers to save power, while keeping the load of the working servers balanced; BF brings into question all the neighborhood allocation at each *MasterClient* interaction, considering only power-saving objectives.

Figures 2.9c and 2.9d show the number of migrations executed. Since the number of VMs can vary a bit from a

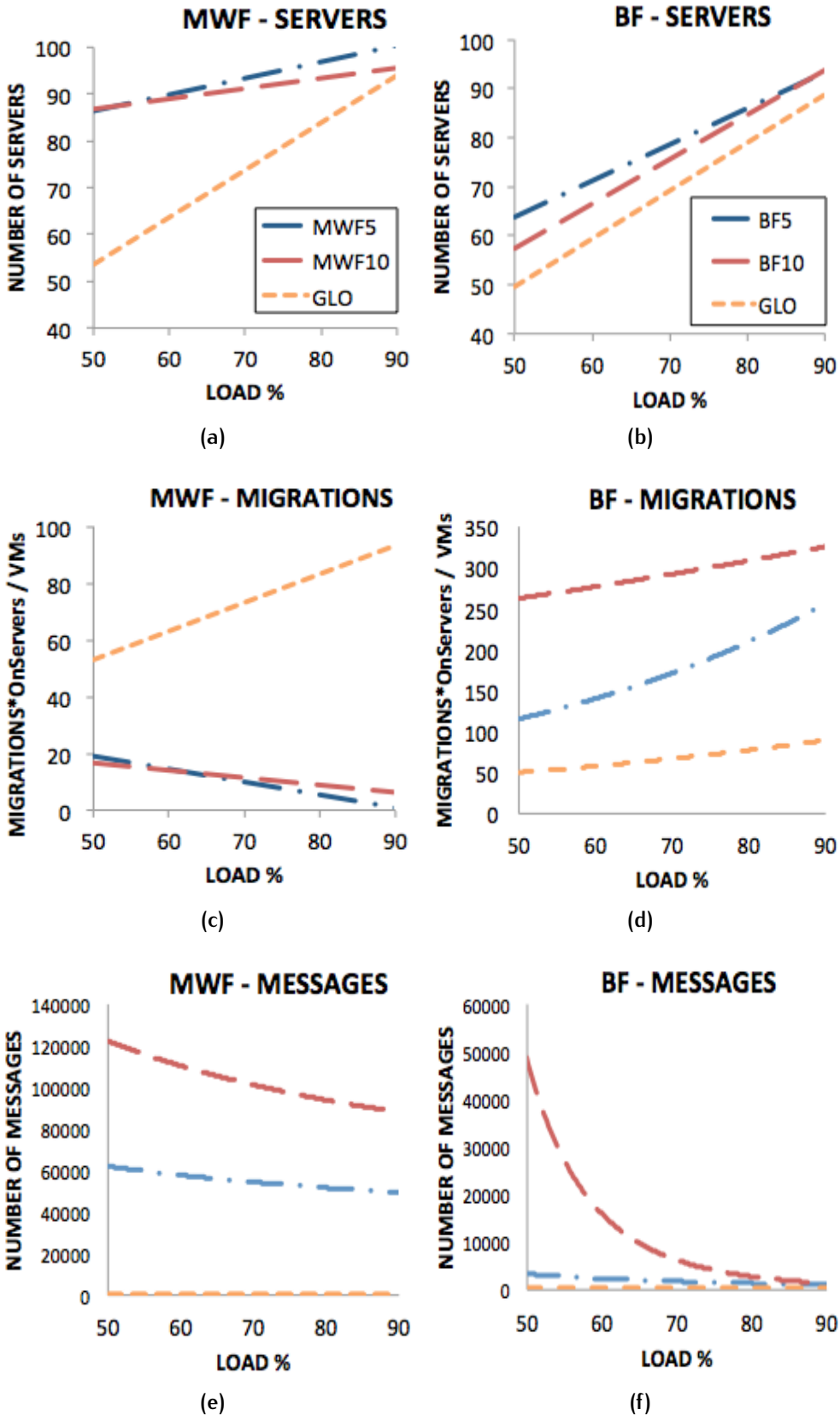


Figure 2.9: MWF end BF performance comparison for various values of average load on each physical server (LOAD% on the x-axis).

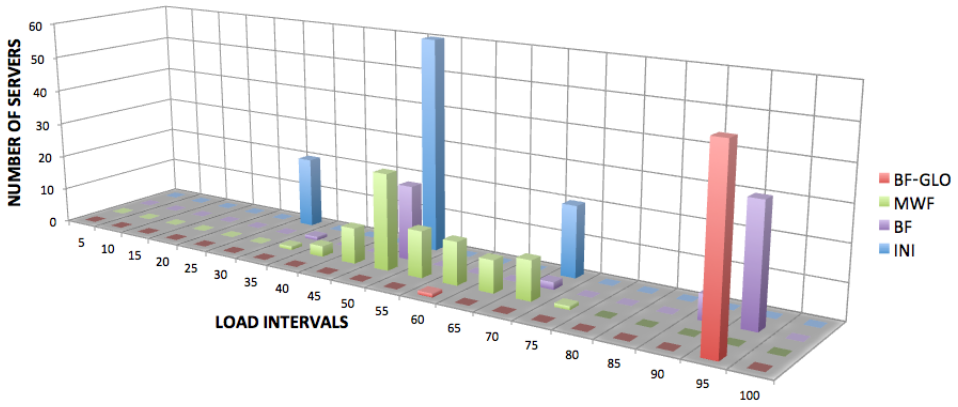


Figure 2.10: Distribution of servers on load intervals. In the initial scenario (INI) all the servers have 50% load except for 20 underloaded and 20 overloaded nodes.

scenario to another and the number of switched off servers influences the result, in the graph we show the following rate:

$$nMig\frac{onServers}{nVM} \tag{2.2}$$

where $nMig$ is the number of migrations performed, $onServers$ is the number of working servers at the end of the simulation and nVM is the number of VMs in the initial scenario.

Since no information about the current allocation of a VM is taken into account during the policy computation in a global environment, the number of migrations can be very high. Indeed is high the resulting trend of migration for the global policy, while DAM always outperforms it. In particular, MWF performs better than BF for every selected neighborhood dimension. Nevertheless, for high values of computational load the performance of MWF in terms of number of switched off server are comparable to those of the global best fit policy, while the migration rate is significantly lower.

Figures 2.9e and 2.9f show the number of messages exchanged between hosts during the computation. As we expected, it significantly increases as the number of servers in each neighborhood grows. Although the number of messages for low values of neighborhood dimension is comparable to the one of the global solution, when it grows, the number of messages exchanged significantly increases.

Figure 2.10 clarifies MWF performance for one of the previous experiments: in the initial scenario (INI in Figure

2.10) all the servers have 50% load except for 20 underloaded and 20 overloaded nodes. Figure 2.10 show the distribution of number of servers along load intervals before and after the intervention of MWF.

The application of a global best fit (BF-GLO) switches-off a large number of servers to save power, but packs too much VMs on the remaining hosts. This results in the red distribution in Figure 2.10, where almost all the switched-on servers are at 95% of utilization, creating an high risk of SLAsviolations. The BF algorithm applied by means of DAM protocol suffers of the same problem: a large number of servers is switched-off, but a part is forced to have 95% load. MWF is more effective from the load balancing perspective: it can switch-off less servers than BF, but is able to decrease the load of the overloaded nodes leaving all the working servers balanced.

As we expected, Figure 2.10 reveals that the median of the MWF distribution is augmented respect to the initial configuration. This is due to the fact that a certain number of servers is switched-off, thus the global load of the remaining servers results increased.

In order to provide a clearer idea of the efficacy of our approach, we separately tested MWF performances in terms of energy efficiency and load balancing. To this purpose, we built three different scenarios.

Scenario 1: load balancing test

Considering MWF from the load balancing perspective only, we created a collection of 50 initially unbalanced scenarios satisfying the constraint:

$$U_{TOT} > FTH\_UP(N-1) \tag{2.3}$$

where U_{TOT} is the total CPU utilization of the datacenter and N is the number of simulated servers. In this way, we can ensure that no server switch-off is possible, and we can test the MWF load balancing performance only.

By defining $U_{AVG\_N} = U_{TOT}/N$ the average load over N servers, the relation 2.3 can be rewritten as follows:

$$U_{AVG\_N} > FTH\_UP\frac{N-1}{N} \tag{2.4}$$

and the initial scenarios can be built such that each server h has a CPU utilization U_h uniformly distributed in the interval:

$$U_h \in [U_{AVG\_N} - q, U_{AVG\_N} + q] \tag{2.5}$$

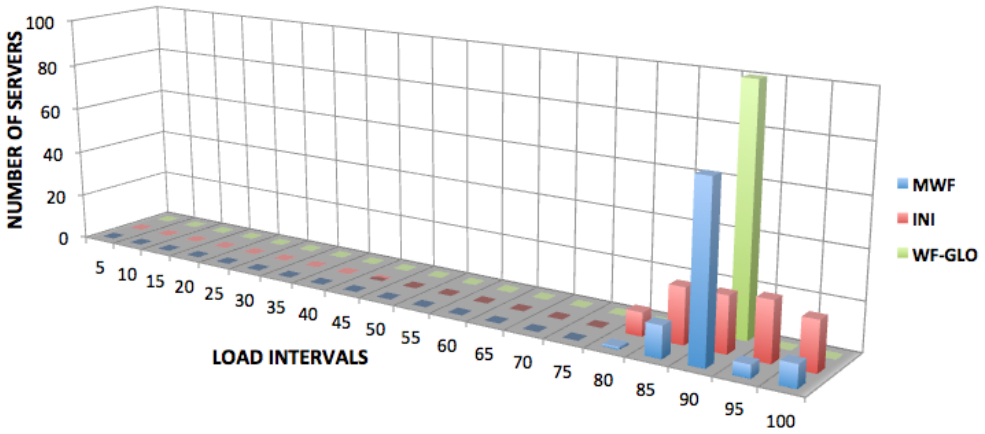


Figure 2.11: Distribution of servers on load intervals. In the initial scenario (INI) all the servers are on average loaded around the value of FTH_UP.

where q expresses the degree of imbalance in the initial scenario. We tested the MWF performance with FTH_UP = 90%, q = 10% and averaged the results over 50 simulations.

In each scenario the topology of the neighborhoods is generated randomly.

Figure 2.11 shows the distribution of servers over load intervals. In the initial scenario (INI) all the servers are on average loaded around the value of FTH_UP. We show the distribution after a global WF optimization (WF-GLO in Figure 2.11) and the application of MWF by means of DAM protocol with 10 as neighborhood size.

MWF shows good performance from the load balancing perspective even if, as we expected, relying on a global knowledge of status of each server, the centralized application of a worst fit policy clearly outperforms the distributed approach.

Scenario 2: power saving test

In order to mainly test the energy saving performance of MWF, we create a collection of scenarios satisfying this constraint:

$$U_{TOT} = FTH\_UP \cdot M, s.t. M < N \tag{2.6}$$

where M is the number of servers that remains switched-on at the end of the optimization. Relation (2.6) can be rewritten as follows:

$$U_{AVG\_N} = FTH\_UP \cdot \frac{M}{N} \tag{2.7}$$

Therefore, given a certain U_{AVG_N} we can calculate the minimum number M_{opt} of servers that can execute the datacenter's workload:

$$M_{opt} = \frac{U_{AVG\_N} \cdot M}{FTH\_UP} \tag{2.8}$$

We create a collection of scenarios with increasing values of U_{AVG_N}, having the load U_h of each server again uniformly distributed in the interval (2.5) and $q = 20\%$, and we use M_{opt} to evaluate the performance of MWF.

Figures 2.12a, 2.12b and 2.12c show the number m of working servers at the end of different MWF distributed executions. These values are compared to the minimum possible number M_{opt} of running servers in each scenario.

Each point in the graphs of Figure 2.12 represents an initial scenario with different value for U_{AVG_N}.

We repeated the experiment with three different values of the rate q/t. Figure 2.12a shows the energy saving performance with $q = 15\%$ of imbalance in the initial scenario and $t = 5\%$ as MWF tolerance interval. The number of switched-off servers is far from the optimum value (expressed by the blu line) for every generated scenario, while decreasing the ratio q/t to $20/5$ and $20/3$ (as reported by Figures 2.12b and 2.12c) the performance of MWF significantly increases.

For low values of U_{AVG_N} the algorithm seems to perform significantly better for every value of the rate q/t, This effect is due to the FTH_DOWN, which is fixed at 25% in every scenario and can therefore contribute to make some *MasterClients* start if the hosts are detected to be underloaded ($U_h <$ FTH_DOWN).

Scenario 3: scalability test

In order to test the scalability of the distributed approach, we analyzed MWF behavior while increasing the number of simulated servers and VMs up to 2000 and 60000, respectively. Figure 2.13a shows the number of migrations stated by MWF and compare it with that of WF-GLO policy.

Since the number of VMs increases with the number of hosts, we actually compare the ratio between migrations and number of VMs in the scenario.

WF-GLO policies does not take into account the current allocation while performing the optimization, therefore, it results in a very high number of migrations, near to the total

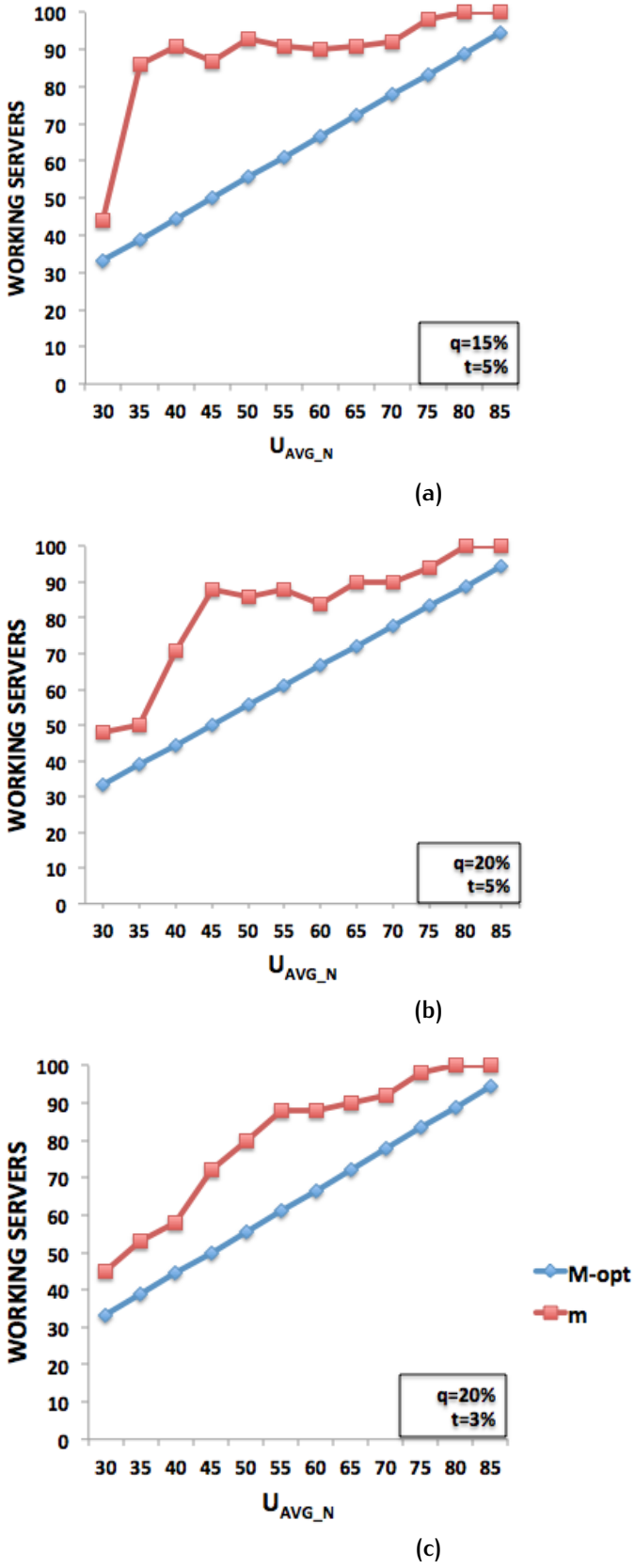
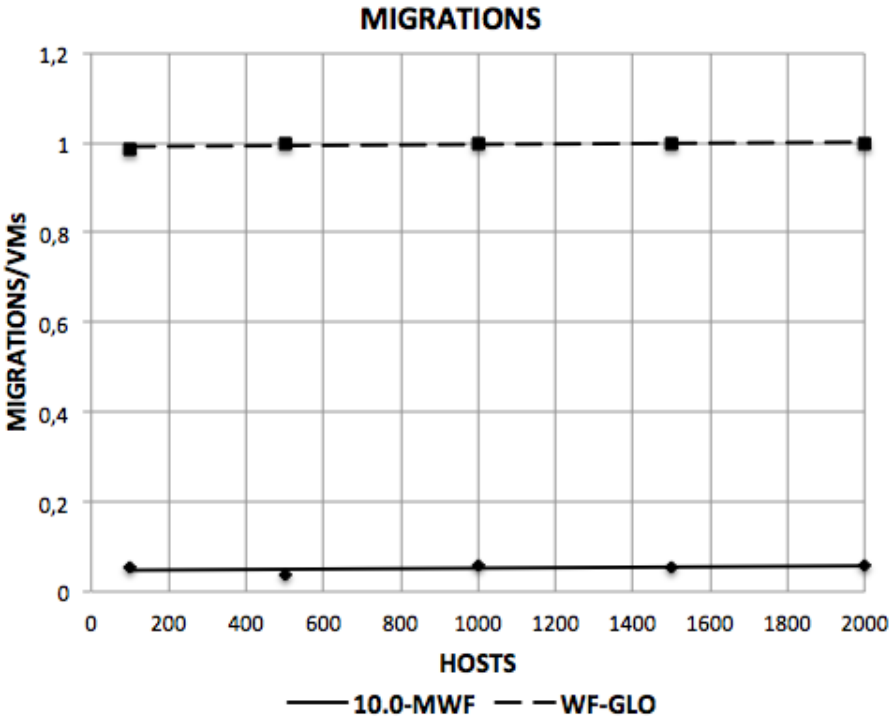
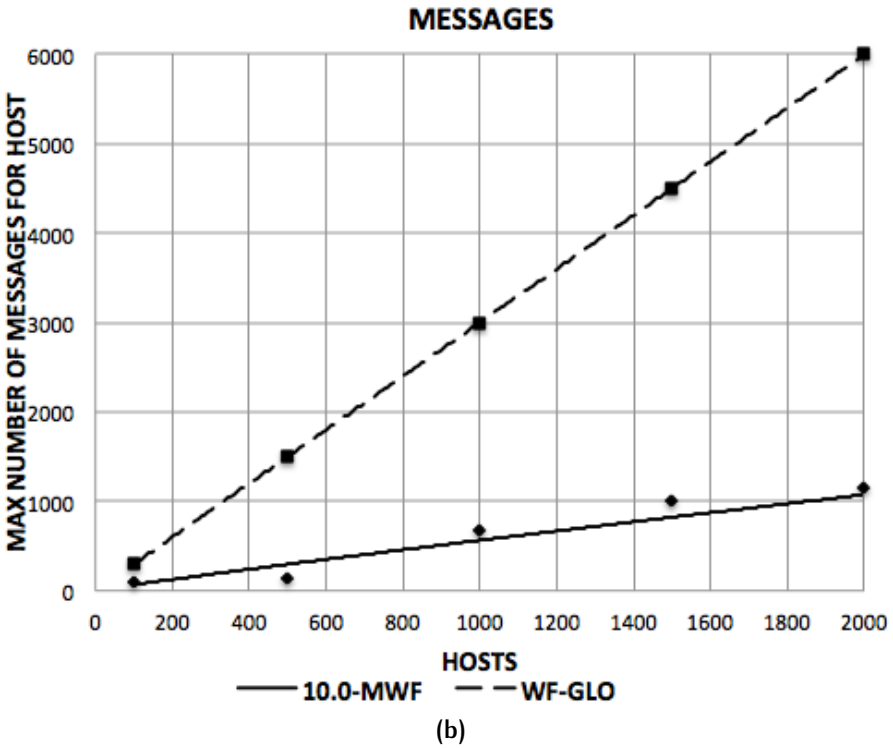


Figure 2.12: MWF power saving performance test. The number m of working servers at the end of different MWF executions is compared to the minimum possible number M_{opt} of running servers in each scenario. The experiments are repeated with different values of the ratio q/t.



(a)



(b)

Figure 2.13: MWF and WF-GLO performance comparison. 2.13a: Number of migrations performed for increasing number of simulated hosts. We compare the performance of MWF with tolerance interval 10.0 with centralized WF-GLO. 2.13b: Maximum number of messages exchanged (sent and received) by a single host of the datacenter. MWF significantly outperforms WF-GLO for high values of simulated hosts.

of VMs. Conversely, MWF distributed policy only operates on underloaded or overloaded nodes (with CPU utilization lower than FTH_DOWN or higher than FTH_UP, respectively) or on those hosts that are unbalanced in respect to the average of their neighborhood (CPU utilization out of the interval [MTH_DOWN, MTH_UP]). For this reason, as shown in Figure 2.13a, the number of resulting migrations is significantly lower for MWF.

In Figure 2.13b, we consider the maximum number of messages exchanged by a single host. Since WF-GLO is centralized, the coordinator node must collect the state of all the other nodes before starting the optimization and finally return the new configuration to each node. Therefore, as shown in Figure 2.13b, the number of messages exchanged by the coordinator is always proportional to the number of the nodes it manages. The behavior of MWF policy is again proportional to the number of nodes but the trends is significantly lower. This comes from the fact that, according to MWF, each node of the datacenter always communicate with a predefined number of neighbors (5 in this simulation).

Therefore, considering the maximum number of messages exchanged by a node, for high values of simulated hosts, we can conclude that MWF distributed approach performs better than the centralized WF-GLO algorithm.

2.5 CONCLUSIONS AND FUTURE WORK

We contributed to the autonomic VM management field by presenting a decentralized solution for cloud virtual infrastructure management (DAM), in which the hosts of the datacenter are able to self-organize and reach a global VM reallocation plan according to a given policy. Relying on DAM protocol, we investigated two VM migration approaches (MB and MWF) suitable for a distributed management in a cloud datacenter.

We tested MB and MWF behaviors by means of DAM-Sim simulator. The policies show good performance for various data centers dimensions in terms of both number of migrations requested and maximum number of messages exchanged by a single host. Therefore, we can assert that the decentralized nature of our approach can intrinsically contribute to increase the scalability of the cloud management infrastructure.

MB is also able to achieve an appreciable load balancing among working servers, even if, as we expected, the distributed MB policy cannot outperform a centralized global worst-fit policy.

MWF inherits from MB the good load balancing performance, while still some work remain to do to decrease the total number of messages exchanged.

As we expected, the distributed MWF policy cannot outperform a centralized global best-fit policy (especially in terms of number of switched-off hosts and exchanged messages), but further investigations of performance on increasing size datacenters has shown that the decentralized nature of our approach can intrinsically contribute to augment the scalability of the cloud management infrastructure.

The current version of DAM-Sim is not able to cover all the aspects of real cloud data centers: it can only simulate CPU resources and is not able to derive the optimal neighborhood structure from a real cloud network. Since our approach benefits from a richly connected data center, a possible future work could be the automatic definition of neighborhoods over a VL2 network [53], while DAM-Sim needs to be extended in order to take into account not only computational resources, but also memory and bandwidth requirements. This will allow us to test different and more elaborated reallocation policies. We also need to introduce variations of VM load requests at simulation time to better mirror real datacenter environments.

Further investigation will be necessary to address issues caused by message losses: the algorithm needs a recovery strategy to avoid the physical servers never-ending blocked while they wait for "unlock" messages.

Finally, since the simulator can only represent a snap-shot of the datacenter workload, it must be extended in order to apply the distributed policy to a dynamically changing scenario and make a benchmark comparison of our approach focusing on time performance. At latter stage, the frequency of variations in VM load requests will be increased to better mirror real datacenter environments.

3 AUTONOMIC MANAGEMENT IN MULTIPLE CLOUDS

Offering "the illusion of infinite computing resources available on demand" [4], cloud computing is the ideal enabler for high computing power demanding applications. For example, when a company deals with data-processing and it is hard to predict the volumes involved, the elasticity of the public cloud can be very useful to dynamically provide the required computational resources.

While the first hype cycle of the cloud model primarily explored the public cloud scenario, many organizations are now focusing on a scenario composed of multiple clouds as the one shown in Figure 3.1. Indeed, albeit one of the most important features of cloud computing is the illusion of unlimited resources, the capacity in cloud providers' data centers is finite and eventually can be fully utilized [54, 55].

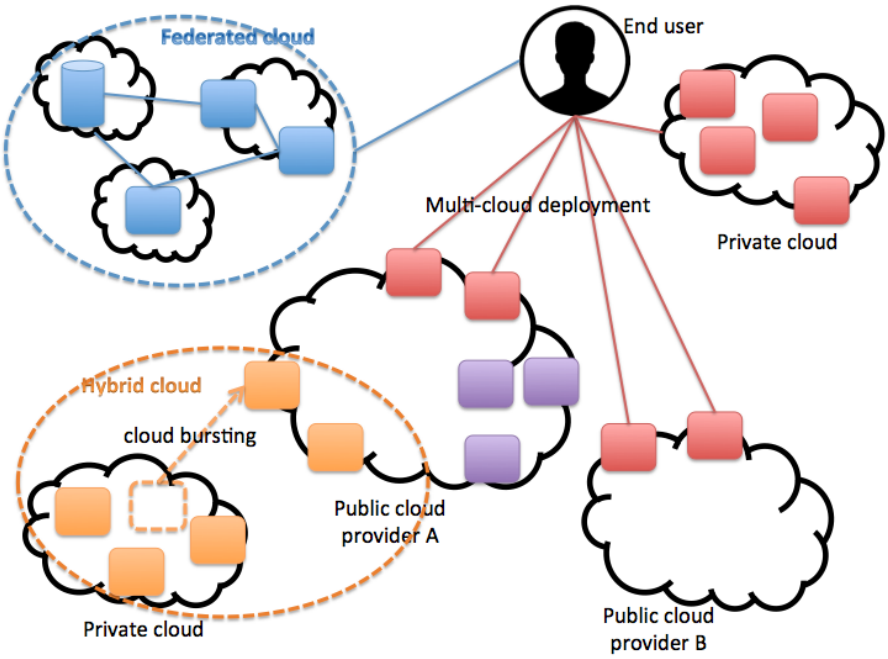


Figure 3.1: Multi-cloud scenarios.

For example, the increase in the demand for a service or the growth in scale of an application may result in the

need for additional capacity in the data center. Since actual resource utilization of many real-world application services vary unpredictably with time, a single cloud provider may be overburden by unexpected load leading to unreliable and interrupted services [1]. In order to handle this issue, a widespread choice by cloud providers is the *overprovisioning* of data center capacity. That is, maintain the computing infrastructure several times bigger than the average demand of the system, even if this strategy can lead to a waste of resources and consequently large expenses for cloud owners.

Conversely, if cloud providers were able to dynamically scale up or down their data center capacity relaying on interoperation of the clouds, they could overcome this issue saving a substantial amount of money. Cloud interoperability can help to handle the peak-load of services on the cloud through resource sharing, avoiding the cost of provisioning and administration of any additional hardware[10].

Public cloud providers usually relay on large scale data centers and can therefore easily give the perception of unlimited resources. In this scenario, one may argue that cloud providers never need immediate external additional capacity. However, this assumption does not hold in case of small-size private clouds and for those applications requiring expansion across geographically distributed resources to meet QoS requirements of their users [56].

Another important motivation for cloud interoperability is the necessity to improve availability, which is a key feature of cloud services. In fact, unexpected failures are always possible and can easily result in service interruption in a single cloud system, while a multiple cloud infrastructure can implement flexible mechanisms to relocate resources and continue the delivery of guaranteed service levels. If the physical allocation (on a specific cloud) of the application-level services is not transparent to the cutomers, this knowledge can also be used to intentionally build multiple cloud deployments in order to improve the availability and disaster recovery capabilities.

Furthermore, in order to meet the low-latency access requirement of some applications, a single cloud provider should have a data center in all geographic locations of the world. This is highly unlikely to happen but cloud federation can provide a similar solution allowing the cloud systems to complement each other by sharing their resources. As pointed out by Toosi et al. [1], “utilizing multiple clouds at the same

time is the only solution for satisfying the requirements of the geographically dispersed service consumers who require fast response time". Nevertheless, in this scenario, a further mechanism to dynamically coordinate load distribution among different cloud data centers is required. As some customers have specific restrictions about the legal boundaries in which their data can be hosted, cloud interoperability could also represent an opportunity for the provider to identify other providers able to meet the regulations due to the location of their data centers [1].

Finally, cloud interoperability can give a further improvement to the Green Computing area. The necessity to save energy by avoiding the problem of the idle capacity without losing the capability to respond to peaks in demand, can be managed through the leasing of underutilized computational power or on-demand purchasing additional resources from other providers. In this way, we would experience an overall increase in the efficiency of resource utilization.

In the following chapters, we consider – as a case study – the execution of data-intensive applications over multiple clouds. Indeed, the exponential increase in the use of mobile devices, the wide-spread employment of sensors across various domains and, in general, the trending evolution towards the Internet of things, is constantly creating large volumes of data that must be processed to extract knowledge. This pressing need for fast analysis of large amount of data calls the attention of the research community and fosters new challenges in the *big data* research area [57]. Since data-intensive applications are usually costly – in terms of CPU and memory utilization –, and often require the execution to satisfy constraints such as deadlines or reliability, facing huge and sometimes unpredictable amounts of input data, a lot of work has been done to simplify the distribution of computational load among several physical or virtual nodes and take advantage of parallelism.

For example, MapReduce [58] programming model and its popular open-source implementation Apache Hadoop [59] allow to transparently partition the input data-set into an arbitrary number of parts, each exclusively processed by a different computing node. Nevertheless, the fulfilling of a given time constraint requires a high degree of elasticity in resource provisioning: if an unexpected data peak appears, it is necessary to quickly provide additional computing resources

to maintain the execution time within the given deadline. In this scenario, a widespread choice is to relay on a cloud infrastructure to take advantage of its elasticity in virtual resource provisioning. Furthermore, the on-premise (company-owned) cloud can be combined with the resources of an off-premise (owned by a third party provider) cloud to further improve the elasticity of the virtual infrastructure. Indeed, many organizations are now focusing on a multiple cloud scenario to enable the execution of data-intensive applications and, in general, to take advantage of the numerous benefits that cloud interoperability can bring to both cloud provider and customer.

In the following, we mainly focus on the enhancements that cloud interoperability can bring to the big data research area if coupled with the autonomic ability to scale-up/-down the computing infrastructure when the resource demand changes. We firstly provide a classification of the cloud interoperability scenarios and a overview of main existing contributions to enable data-intensive computation over multiple clouds (Chapter 3.1). Later (Chapters 3.2 and 3.3), we describe our solution as well as the leveraged infrastructure and application-level tools. In Chapter 3.4, 3.5 and 3.6 we describe the adopted approaches in detail by focusing on the policies and models to enable MapReduce in multi-cloud environments. We provide our conclusions and a vision of possible evolutions of the works in this field in Chapter 3.7.

3.1 POSITIONING OUR CONTRIBUTION

As integration and aggregation of cloud services have recently received the attention of the research community, several different terms have been used to define cloud interoperability. A precise understanding of these terms and definitions, including differences and similarities, clarifies the position of our contribution.

The term Inter-cloud has been introduced by Cisco [60] as an interconnected global "cloud of clouds" that recalls the known term Internet, "network of networks". The Inter-cloud refers to a mesh of clouds that are unified by open standard protocols to provide a cloud interoperability. The Inter-cloud final goal is to create a web of computing element ubiquitously connected

together in a multi-provider infrastructure (similarly to the Internet model and telephone system).

The term cloud federation, on the other hand, refers a group of aggregated providers that share their resources in order to improve each other's services [61].

The terms Inter-cloud and cloud federation are often used interchangeably in the literature. The primary difference between the two is that the Inter-cloud is based on open interfaces (derived from standards), while the federation implies to make interoperable a provider's version of the interfaces. Therefore, cloud federation can be considered as a prerequisite toward the ultimate goal of the Inter-cloud, where interoperability of different cloud platforms is transparently achieved by users [1]. According to Chen et al. [62], interoperability can be obtained by adhering to published interface standards, or developing a broker of services that can "on the fly" convert one cloud interface into another.

As asserted by Toosi et al. [1], the "transition toward Inter-cloud has already started and it is an inevitable need for the future of cloud computing". In this regard, we present different scenarios for cloud interoperability in the next chapter.

3.1.1 Cloud interoperability scenarios

Since "cloud computing refers to both the applications delivered as services over the internet and the hardware and systems software in the data centers that provide those services" [63], cloud services can be sold by providing IaaS – like Amazon EC2 [6] – or at a higher level, realizing the PaaS and SaaS – like Google AppEngine [64]. When a cloud is available to the public in a pay-as-you-go manner, it is called *public* cloud, and when a cloud belongs to a company or organization and not made available to the public, it is called *private* cloud. Cloud environments include a multitude of independent, heterogeneous, private, and public clouds.

The main actors of cloud computing scenarios are cloud users and providers. Cloud users can be either software/application service providers, who have their service consumers or end-users utilizing the cloud computing services directly. Service providers offer their services using hardware resources provisioned by cloud providers. Different combinations of cloud providers and cloud users (service

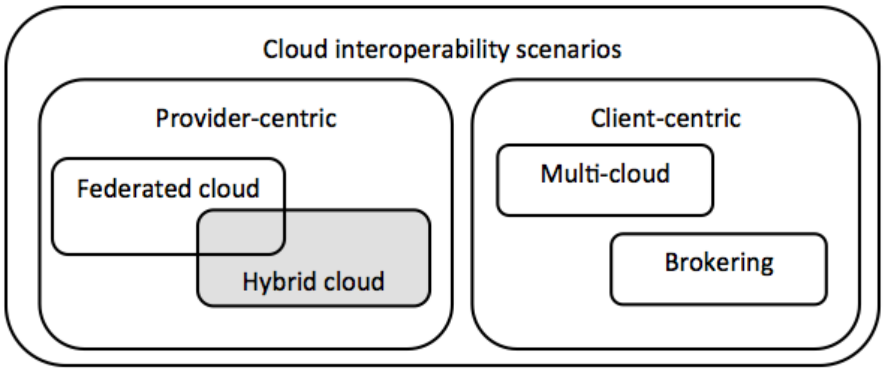


Figure 3.2: Classification of interoperability solutions for clouds as suggested in the work by Toosi et al [1]. This dissertation mainly focus on solutions to enable the hybrid cloud scenario in grey.

providers or end-users) give rise to several plausible use cases between clouds [65]:

- if cloud interconnection is performed at different levels of cloud stack layers, for example, a PaaS and IaaS provider, the scenario is often referred as *delegation* or *vertical federation*;
- if interconnection between clouds is at the same layer (e.g., IaaS to IaaS), it is called *horizontal federation*.

As pointed out by Villegas et al. [66], a federated cloud structure is a vertical stack analogous to the layered cloud service model. At each layer, a service request can be served either through local resources using delegation or by a partner cloud provider through federation.

An exhaustive classification of the different scenarios of cloud interoperation can be found in [1]. According to this classification, if cloud interoperability is realized by cloud providers, who adopt and implement standard interfaces, protocols, formats, and architectural components to facilitate collaboration, we have a provider-centric interoperability. Provider-centric scenarios are categorized as *hybrid* and *federated* cloud scenarios (as depicted in Figure 3.2). On the contrary, client-centric interoperability is not supported by cloud providers. The customers are required to initiate it by themselves or via third-party brokers. This kind of interoperability does not require an *a priori* business agreement among cloud providers and allows multiple cloud scenarios without (or with minimal) adoption of common interfaces and

protocols[1]. We consider *multicloud* and *aggregated service by broker* as client-centric interoperability scenarios.

In *multicloud* scenario in particular, the customers are responsible for the management of resources across multiple clouds. Service deployment, negotiation, and monitoring of each cloud provider during service operation are performed by end-user applications. In this case, the use of an adapter library with different APIs to run services on different clouds is required.

In *aggregated service by broker* scenario, a new actor, the broker, aggregates services from multiple cloud providers and offers to the final customers an integrated single entry point to the services. In this way, from the point of view of the customer an abstraction layer is created for the management of components deployed on different clouds.

The *federated cloud* scenario is a case of provider-centric approach to interoperability. Here, a group of cloud providers are federated and trade their surplus resources among each other to gain economies of scale and a higher level of efficiency. Therefore, the customer establishes a contract with a cloud provider that is a member of a federation, and the computing utility is delivered to him using resources of either one cloud provider or a combination of different cloud providers. The customer might be aware or unaware of the federation and his contract is with a single cloud provider.

In *hybrid cloud* architectures, an organization that owns its private cloud moves part of its virtual infrastructure to external cloud providers. This extension of a private cloud to combine local resources with those of a remote off-premise cloud allows the end-user applications to scale-up (cloud burst) through public clouds when the local infrastructure is insufficient. When the emergency is solved the customer can scale the application back down in the private cloud to minimize the cost of the public infrastructure utilization. Furthermore, this scenario can be extended if the organization offers capacity from its private cloud to others when that capacity is not needed for internal operations. In fact, as suggested by the overlapping areas in Figure 3.2, federated and hybrid scenario shows common aspects from the point of view of the technologies to enable interoperation.

In the following, we focus on the hybrid cloud model as a method to overcome the limits of a single on-premise cloud

infrastructure when dealing with heavy computing tasks such as data-intensive applications.

3.1.2 Enabling data-intensive applications over multiple clouds

MapReduce programming model is a widespread choice when dealing with data-intensive tasks because it simplifies the implementation of distributed application for data processing. According to this approach, the user must specify two functions: the *mapper* and the *reducer*. The *mapper* extracts from the given input data a series of intermediate *key/value* pairs, while the *reducer* merges all the intermediate values associated with the same key. The programs implemented according to this model can be automatically parallelized and easily executed on a distributed infrastructure [58].

MapReduce model is implemented and supported by several platforms [59, 67–69]. In this work, we opted for Apache Hadoop [59], one of the most used and popular frameworks for distributed computing.

MapReduce success is primarily due to its simple parallelization model, graceful degradation in case of partial system failures and good horizontal scalability. Nevertheless, data-intensive applications require a high degree of elasticity to deal with unpredictable data peaks or execution deadlines. This requirement has inevitably directed the attention of the big data research area towards the cloud. In particular, combining both on-premise and off-premise cloud infrastructures, the hybrid scenario is the ideal enabler for data-intensive applications. The rationale behind employing a hybrid approach can be manyfold, e.g. using off-premise resources for being able to guarantee a minimum QoS, satisfying a predefined deadline constraint, or partitioning the computation between on- and off-premise zones as mandated by security compliance requirement (for instance, if part of the data is not allowed to cross the boundary of the on-premise infrastructure).

In fact, among the various widely used and trending platforms that leverage the hybrid cloud delivery model [70, 71], data-intensive analytics at large scale is one of the most challenging cases, especially when large volumes of data are involved.

Conventional big data analytics frameworks – such as Hadoop [59] – assume physical co-location of IT resources with

high-speed interconnection among servers. This assumption is lifted in hybrid cloud environments comprising physically separated datacenters that are interconnected via a network that is at least an order of magnitude slower (either dedicated connectivity or the open Internet).

Furthermore, due to the size of the data being analyzed, most big data analytics techniques make heavy use of data locality by shipping the computation close to the data. This aspect is particularly challenging when boosting a private cloud with extra temporary VMs from a public IaaS cloud to finish a big data analytics job faster: the newly provisioned VMs do not hold any data and as such it is necessary to send large amounts of data over the slow link in order to be able to leverage the extra computational capability.

Therefore, we can claim that data movement is elevated to a major challenge in hybrid cloud big data analytics.

3.1.3 MapReduce over cloud environments: state of the art

Recently, a lot of work has focused on cloud computing for the execution of big data applications: as pointed out in [72], the relationship between big data and the cloud is very tight, because collecting and analyzing huge and variable volumes of data require infrastructures able to dynamically adapt their size and their computing power to the application needs. For this reason, several efforts to improve the MapReduce performance leverage cloud technologies [73–76], with the assumption of having unlimited computing resources. These efforts are complemented by works on storage elasticity [77, 78], which explore how to deal with the explosion of the costs related to the storage capacity and the Input/Output (I/O) bandwidth, and are highly relevant for MapReduce applications.

The work by Chen et al. [79] presents an accurate model for optimal resource provisioning useful to operate MapReduce applications in public clouds. Similarly, Palanisamy et al. [80] deal with optimizing the allocation of VMs executing MapReduce jobs in order to minimize the infrastructure cost in a cloud datacenter. In the same single-cloud scenario, Rizvandi et al. [81] focus on the automatic estimation of MapReduce configuration parameters, while Verma et al. [82] propose a resource allocation algorithm able to estimate the amount of resources required to meet MapReduce-specific performance

goals. However, these models assume an unlimited amount of available resources (as in the classic public cloud) and are not intended to address the challenges of the hybrid cloud scenario, which is the target environment of our work.

According to the classification in [1], our work mainly deals with the hybrid cloud approach for cloud interoperability, because the main motivation of our system is to allow cloud bursting to EC. However, our proposal could also be classified as a *Federation* mechanism for cloud aggregation because – as in federated clouds – the interoperation between clouds is completely transparent to end-users.

The choice of primarily relay on a small (e.g., private) cloud and then use the extra-capacity offered by a public cloud for opportunistic scale-out has been investigated by several authors in the past [83, 84].

The work in [85] and [86] focus on enabling cloud bursting through inter-cloud migration of VMs, which is generally a time and resource expensive mechanism. The system described in [86], in particular, optimizes the overhead of migration using an intelligent pre-copying mechanisms that proactively replicates VMs before the migration. Our work doesn't take into consideration the VM migration, but only dynamic instantiation of new compute nodes on EC, thus to avoid the unnecessary movement of the whole VM snapshot across the cloud boundaries. This technique is particularly suitable for the MapReduce model because the Hadoop provisioning and decommissioning mechanism intrinsically contributes to simplify the cloud bursting process.

The hybrid scenario is also investigated in the works by Zhang et al. [70, 87] by focusing on the workload factoring and management across federated clouds. More similarly to our approach, cloud bursting techniques has been adopted for scaling MapReduce applications in the work by Mattess et al. [88], which presents an online provisioning policy to meet a deadline for the Map phase. Differently from our approach, [88] only focuses on the prediction of the execution time for the map phase but does not take into account the resource utilization, nor provides an autonomic mechanism to reconfigure the cluster if a future deadline exceeding is predicted. Furthermore, [88] relies on the assumption that the distributed filesystem is persistently deployed and loaded with data throughout the execution of MapReduce workloads. As such, it does not capture the initial phase of cross-cloud data

distribution and data-balancing, a vital phase in high value hybrid cloud use cases, such as cloud-bursting.

Also the work presented by Kailasam et al. [89] deals with cloud bursting for big data applications. It proposes an extension of the MapReduce model to avoid the shortcomings of high latencies in inter-cloud data transfer: the computation inside IC follows the batch MapReduce model, while in EC a stream processing platform called Storm is used. The resulting system shows significant benefits. Differently from [89], we chose to keep complete transparency and uniformity with respect to the allocation of working nodes and their configuration.

Several recent efforts are focused on improving the performance of MapReduce frameworks for hybrid environments. HybridMR [90] proposes a solution for executing MapReduce in a hybrid desktop grids and external voluntary nodes, but does not consider expanding and shrinking a MapReduce setup dynamically. HadoopDB [91] proposes a a hybrid system comprising Hadoop and parallel database systems to yield the resilience, and scalability of Hadoop and the performance and efficiency of parallel databases. Similarly, hybrid scheduling techniques [92–94] uses GPUs to improve the performance of MapReduce applications in accelerator-enabled clusters. These techniques uses hybrid computing to improve the performance of MapReduce applications, but the hybrid aspect is on the computational side rather than the networking side.

Specifically to workload performance prediction and optimization in hybrid clouds, Van den Bossche et al. [95] have proposed a linear/integer programming model for relevant workloads, showing substantial improvement over naive executions. Albeit valuable from a bounds perspective, such models are typically hard to scale and also fail to capture framework-specific intricacies, such as data-movement and data locality. Imai et al. [96] explore hybrid cloud prediction patterns from the perspective of selection of resource units (VM sizing types) with prediction techniques. This work could be used to enrich the performance of our proposed models (depicted in Chapters 3.3 and 3.6) by selecting matching VM sizes for hosting MapReduce daemons.

The fallowing tables (3.1, 3.2 and 3.3) summarize the main features of the referred works in the field of VM management over multiple-clouds. Table 3.4 refers to our contributions presented in the next section.

Table 3.1: Main features of the works in the field of VM management over multiple-clouds

Contribution	Addressed problem	Cloud model	Cloud platform	MapReduce platform
Zhang et al. [70]	Workload factoring	hybrid	Amazon EC2 & private unmanaged	-
Suen et al. [71]	Intercloud migration	hybrid or federated	Amazon EC2 & Openstack	-
Tsai et al. [73]	MapReduce cluster scaling	public	-	Google MapReduce
Tian et al. [74]	MapReduce cluster provisioning	public	unmanaged	Hadoop
Gunarathne et al. [75]	MapReduce cluster provisioning	public	Microsoft Azure	Azure MapReduce
Zhang et al. [76]	large-scale data anonymization through MapReduce	public	Openstack	Hadoop
Nicolae et al. [77]	storage elasticity	public	unmanaged	Hadoop

Table 3.2: Main features of the works in the field of VM management over multiple-clouds

Contribution	Addressed problem	Cloud model	Cloud platform	MapReduce platform
Nicolae et al. [78]	storage elasticity	public	unmanaged	-
Chen et al. [79]	MapReduce cluster provisioning	public	unmanaged or Amazon EC2	Hadoop
Palanisamy et al. [80]	MapReduce cluster provisioning	public	Amazon EC2	Hadoop or Amazon Elastic MapReduce
Rizvandi et al. [81]	MapReduce parameter estimation	public	unmanaged	Hadoop
Cardosa et al. [83]	MapReduce over hybrid configuration	hybrid	PlaneLab & Amazon EC2	Hadoop
Nagin et al. [85]	Intercloud migration	hybrid	private unmanaged & Amazon EC2	-
Guo et al. [86]	Intercloud migration	hybrid	-	-
Zhang et al. [87]	workload factoring	hybrid	private unmanaged & Amazon EC2	-
Bicer et al. [84]	MapReduce over hybrid performance evaluation	hybrid	private unmanaged & Amazon EC2	-

Table 3.3: Main features of the works in the field of VM management over multiple-clouds

Contribution	Addressed problem	Cloud model	Cloud platform	MapReduce platform
Mattess et al. [88]	scaling MapReduce cluster to meet Map-phase deadline	hybrid	private unmanaged & Amazon EC2	Hadoop
Kailasam et al. [89]	boost MapReduce over hybrid	hybrid	both unmanaged	Hadoop
Sharma et al. [90]	MapReduce over hybrid provisioning	hybrid	both unmanaged	Hadoop
Abouzeid et al. [91]	boost MapReduce over hybrid	hybrid	Amazon EC2	Hadoop
Shirahata et al. [92]	MapReduce over CPU and GPU	hybrid architectures	unmanaged	Hadoop
Van den Bossche et al. [95]	workload scheduling optimization	hybrid	unmanaged	-
Imai et al. [96]	workload scheduling optimization	hybrid	Amazon	-

Table 3.4: Our contributions in the field of VM management over hybrid cloud

Contribution	Addressed problem	Cloud model	Cloud platform	MapReduce platform
Loreti et al. [97]	cost model of MapReduce over hybrid	hybrid	Openstack	Hadoop
Loreti et al. [98]	MapReduce over hybrid cloud bursting policy	hybrid	Openstack	Hadoop
Loreti et al. [99]	MapReduce over hybrid cloud bursting & scale-down policy	hybrid	Openstack	Hadoop
Loreti et al. [100]	iterative MapReduce over hybrid cost model & scaling policy	hybrid	Openstack	Hadoop

3.2 ENABLING TOOLS

In order to better clarify our solution for enabling data-processing over the hybrid cloud, we start by introducing the infrastructure and application level tools we relay on: OpenStack [5] (one of the most popular open-source platforms for cloud computing) and MapReduce [58] (a programming model for distributed data-intensive applications).

3.2.1 OpenStack architecture

OpenStack [5] is an open-source platform for cloud computing released under the terms of the Apache License. It was developed in 2010 during a project lead by Rackspace in collaboration with NASA. Similarly to other free open-source platforms in this field [101–103], it owes its notoriety not only to the economical benefits derived from the open-source license but also to the possibility to adapt the solution to specific requirements. This characteristic is particularly relevant in order to support the implementation of multi-cloud solutions. Furthermore, as open-source cloud platforms mostly support standard interfaces such as Open Grid Forum (OGF) [104] and Open Cloud Computing Interface (OCCI) [105], applications deployed on these architectures can be easily moved from one IaaS provider to another, without having to be modified [1].

OpenStack currently constitutes the basis of several public deployment – e.g., Rackspace public cloud [106], HP Helion cloud [107], etc. – and is extensively used by the research departments of large companies (e.g., IBM), thus revealing the maturity of this open-source product.

Thanks to its modular architecture, OpenStack is particularly handful when a customization of the cloud solution is required. It can be configured either to realize a small private cloud or to control large pools of compute, storage, and networking resources in big data centers. If some functionalities are not relevant to the considered domain, we can avoid the installation and configuration of the correspondent modules, saving disk space and preventing the risk of failures in unused portions of code.

OpenStack has a centralized architecture composed by a single cloud machine (the cloud *controller*) that orchestrates all the physical machines hosting the VMs (the cloud *compute* nodes). In large datacenters, OpenStack can also give support

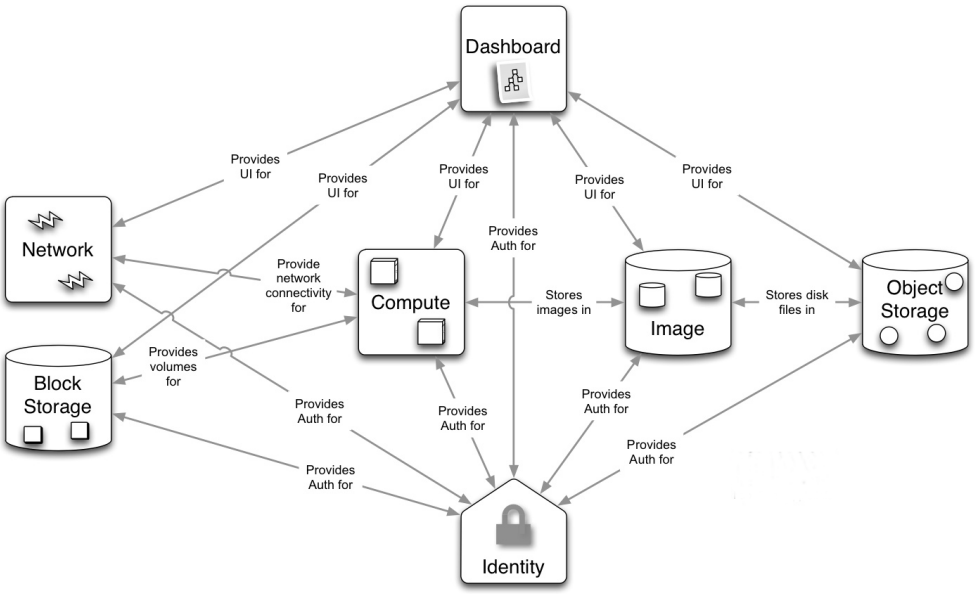


Figure 3.3: Schema of main OpenStack modules interactions [2].

to a multi-layer organization of the physical nodes, thus to realize a "tree" infrastructure. This kind of deployment is particularly useful when we deal with geographically distributed clouds that must coordinate to realize a single public service. In this case the traditional *controller-computes* architecture is replicated in each cloud and the controllers are coordinated by a single upper level controller.

The main OpenStack modules are depicted in Figure 3.3 and described in the following list.

- Horizon Dashboard enables the resource management through a web portal that acts like a command board. Similarly to the OpenStack project, the interface has a modular architecture, thus to show only the GLOBAL USER INTERFACE (GUI) of the installed components. For example, Horizon interacts with all the other modules to enable the creation of VMs, IP address assignment, management of VM images, access control, etc.
- Keystone Identity service is the authentication and authorization service. It manages the users, the tenants (groups of users that map a customer company) and their roles for each offered service. After the login with username and password, a token mechanism is used to grant or deny the access to specific services. Keystone is also responsible for storing the list of the endpoints

that must be involved for each service offered – e.g. the Horizon dashboard URL.

- Glance Image service provides support to memorization and discovery of images for VM instances. Thanks to this service is also possible to easily create a snapshot of the VM and store it into a database for future use.
- Nova Compute is the crucial module for the management and deployment of VMs. It interacts with the hypervisors of the physical machines (e.g., KVM, XenServer, etc.) and all the other modules to provide or delete VMs with specific functionalities. It also offers the possibility to configure the policy of the instance scheduler, by defining which allocation criteria should be taken into account when creating or migrating an instance.
- Neutron Networking enables the Internet connectivity on VM instances and allows the user to create several different network functionalities and configurations through the virtualization of network devices (switch, router, firewall, etc.).
- Cinder Block Storage service manages the creation, deletion and attachment to VM instances of storage volumes supporting different protocols (e.g., iSCSI, NFS, GlusterFS, etc.). Thanks to this module, the data memorized inside the virtual volumes can be saved as snapshots for future restore and attachment to other instances.
- Swift Object Storage service offers a distributed platform for data storage and discovery. It is not a traditional file system but a distributed object storage service, thanks to which the physical machines realizing the cloud can share static data, like VM images, backup archives, multimedia files, etc. Swift is able to guarantee redundancy without any central control system. This decentralized architecture contributes to improve the module scalability.
- Heat service implements an orchestration engine to launch multiple VMs eventually organized in clusters for the execution of a predefined application. The provisioning mechanism is based on templates in the form of text files that can be treated like code.
- Sahara service is the OpenStack module specific to data-processing. Leveraging the functionalities offered by Heat module, allow the user to rapidly create clusters

of VMs configured to execute various data-intensive execution models. It offers the possibility to install several MapReduce platforms on the newly provided VMs, easily launch jobs and scale-up/down the virtual infrastructure at runtime. All these operations can be executed by simply using the Horizon interface.
- Ceilometer service is a monitoring module. It periodically checks the utilization of virtual resources and stores the monitoring information in a specific database for subsequent retrieval and analysis.

Although not specifically designed for either interoperability or portability, OpenStack is very close to being a standard in the cloud ecosystem thanks to its open-source nature and modular architecture. For these reasons, we chose OpenStack as a base for the implementation of our architecture.

3.2.2 MapReduce theory and implementation

Over the last few years, the big-data research area has gained importance thanks to the increase in the use of mobile devices and the adoption of sensors networks in various fields. These technologies have created the pressing need for fast analysis of large volumes of data. The MapReduce programming model [58] and its open-source implementation through the Apache Hadoop runtime [59] have gained significant traction for the purpose of enabling data-intensive applications by distributing the computational load among several servers and taking advantage of parallelism.

MapReduce principals were firstly enunciated in 2004 in the work by Dean and Ghemawat [108] as a solution to simplify the execution of data-intensive applications over a collection of physical servers. Indeed, when the volumes of data to be processed are large, even the execution of simple algorithms (e.g. grep, URL access count, inverted index, etc.) can be highly time-consuming. Before the adoption of the MapReduce model, a popular solution for the problem was to reformulate the algorithm in a distributed form (i.e., in a form that can be executed by multiple servers in a coordinated way), divide and distribute the input data and, eventually, implement disaster recovery mechanisms to avoid that failures on a single node affect the whole computation. In these circumstances, even the implementation of simple algorithms turned into a difficult problem where concurrency must be carefully managed.

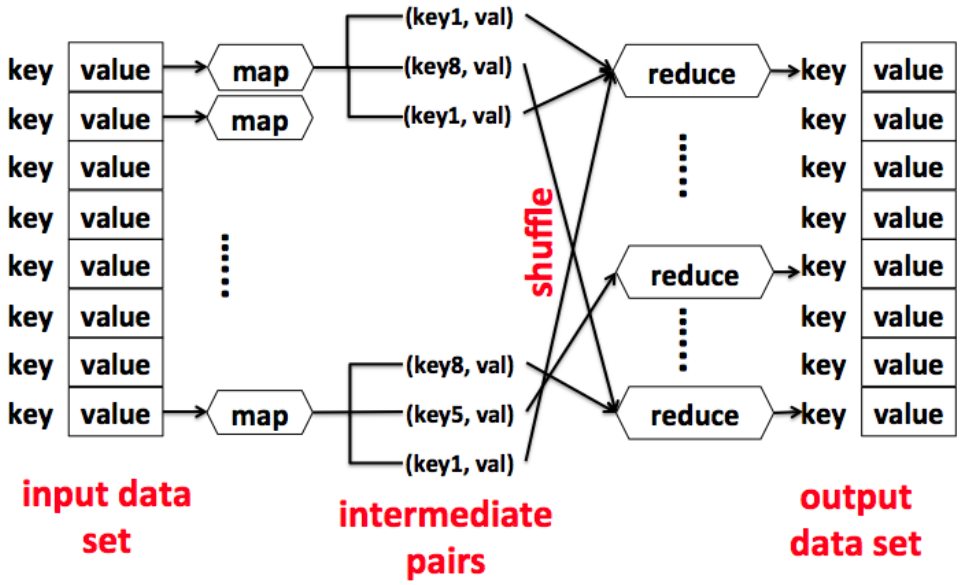


Figure 3.4: Execution of a generic MapReduce application. The map and reduce tasks can be distributed on different computing nodes.

MapReduce model suggests the user to reformulate her program in terms of two functions: *map* and *reduce*. As shown in Figure 3.4, the map function takes as input a portion of the data and emits a collection of <*key*, *value*> pairs, while the reduce function merges the different values associated with the same key. The output of the reduce function is again composed of <key, value> pairs. The program implemented according to this model have the advantage of being intrinsically parallelizable.

A classic example of application that can be easily implemented according to the MapReduce model is the *word count* problem. The goal is to count the number of occurrences of each word in a collection of documents. The solution schema of this case is depicted in Figure 3.5. Each map task takes as input a document of the collection (or a part of a single input document), analyze the content and, for each occurrence of the word w, emits a <w, 1> pair as output. These intermediate pairs are then partitioned by key and sent to the reduce tasks (i.e., every reducer receives all the pairs with the same key) that sums all the different values associated with the same key. The output dataset is therefore composed of <w, n> pairs, where w is a word and n is number of occurrences of w in the documents.

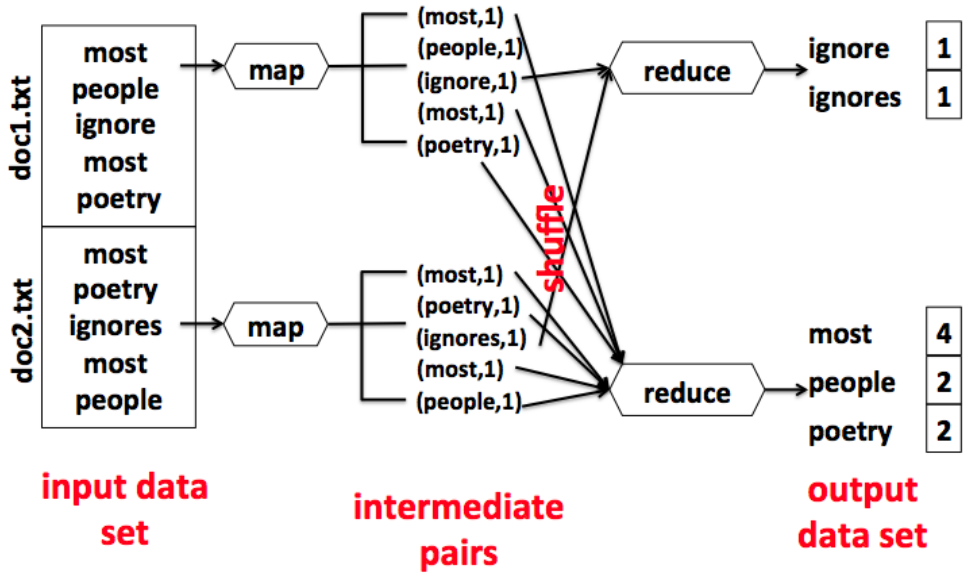


Figure 3.5: Execution schema of a word count application implemented according to the MapReduce model.

Algorithm 8 shows the map and reduce functions through pseudo-code. Since each map task works on a subset of the input data and each reduce task works on a key (by merging its values), these function do not share any data and can be executed in parallel, possibly on different machines. The only constraint is that the reduce tasks should execute after the map output is produced. This characteristic makes the MapReduce model intrinsically parallelizable. Further details about the deployment of MapReduce tasks are illustrated in Figure 3.6.

As suggested in [108], MapReduce model can be used to parallelize several other common problems of big data analysis:

- *Grep* procedure – i.e., filtering the rows of a document that contains a specific string given as input – can be realized by submitting to each map task a portion of the input text. The map performs the grep operation, while the reduce tasks are identity functions.
- URL *access frequency*: given a log containing a list of recently visited URLs, count the number of accesses for each URL. This operation is similar to the word count example and can be implemented in the same way.
- *Inverted index* procedure: given a list of documents, we want to extract for each word the ordered list of documents containing it. Each document is parsed by a

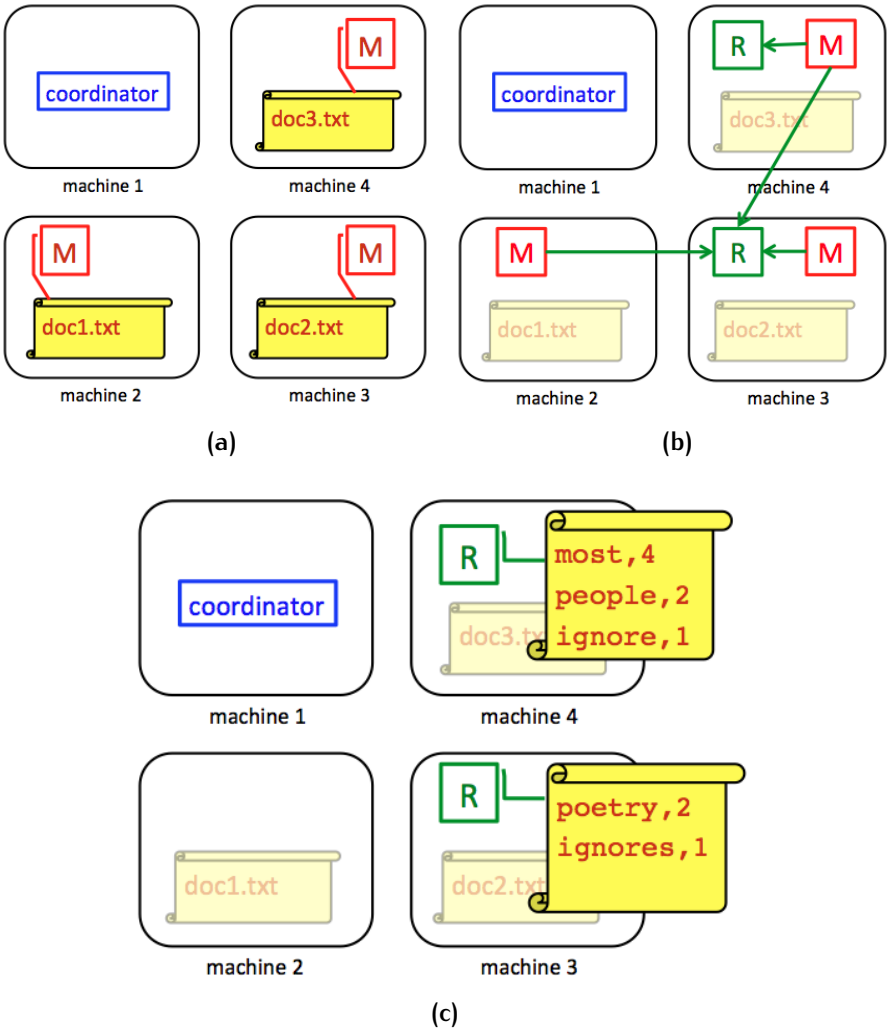


Figure 3.6: Example of task deployment over a four-node architecture for a word count application. The allocation of the tasks is controlled by a coordinator process. The input dataset is partitioned across the nodes and each map task is responsible for the execution over a particular subset of the input (Figure 3.6a). The intermediate pairs are then sent to the reduce tasks (shuffle phase) as in Figure 3.6b. Finally the reducers process merges the received pairs and emit the output (Figure 3.6c).

Algorithm 8: Word count map and reduce functions

```
  /* key:document name. value:document content      */
1 function map(String key, String value)
2     foreach word ∈ value do
3         EmitIntermediate(word,"1") ;
4     end

  /* key: a word. values: a list of counts          */
5 function reduce(String key, Iterator values)
6     int result = 0 ;
7     foreach v ∈ values do
8         result += ParseInt(v);
9         Emit(key, AsString(result)) ;
10    end
```

map task (or split to several map tasks if the size is relevant), such that for every word *w* in the document *docY*, the map emits a <*w, docY*> pair. Each reduce task takes as input all the pairs associated to the same key *w*, sorts the document names and emits a <*w, list(docName)*> pair.

- *Sort* procedure: the map tasks extract the ordering feature from each record of the input data, save it in the key and emits a list of <*key, record*> pairs, while the reduce tasks are identity functions. This procedure is realized thanks to the *ordering guarantee* feature – i.e., each reducer must process and emit its pairs in key order – and a custom partitioning function described in the following.

Furthermore, MapReduce can also be used to implement procedures that requests more than one map and one reduce phase, the so-called *Iterative MapReduce* problems, such as K-means and Iterative grep. Other examples of programs implemented according to the MapReduce model can be found in [58].

As suggested in [108], if the operation executed in the reduce phase is commutative and associative (as for the sum in the word count algorithm), it can be partially anticipated by executing it on the same machines of the map tasks. Another function, the *combiner*, is therefore admitted after the map phase. In the word count problem for example, this function takes as input all the <*w,1*> pairs, calculates the sum *S* of the

occurrences and emits <*w*,*S*>. Then, the reduce task finalize the sum and emits <*w*,*sum(S)*>.

When possible, the introduction of the combiner function is particularly useful because it shrinks the volumes of data sent from map to reduce tasks and partially relieve the work of the subsequent reduce phase.

Partitioning function and shuffle phase

As shown in Figure 3.4, between map and reduce tasks a particular phase, called *shuffle phase*, is needed in order to correctly send the list of intermediate pairs with the same key to the same reduce task.

This operation needs to be performed in a fast way, balancing the amount of data that must be processed by each reducer and without introducing further complexity.

The solution proposed by Dean and Ghemawat [108] is to relay a simple hash function of the keys. The destination reduce task D_R for each key k is therefore determined by the output of the following expression:

$$D_R = (hashcode(k)\&MaxInteger)\%N_R \tag{3.1}$$

where N_R is the number of the reduce task in the architecture and $MaxInteger$ is the maximum integer value supported by the language. The bitwise AND operation in Equation 3.1 is necessary to always obtain positive values, while the module operation (%) produces a value in the interval $[0, N_R - 1]$, thus to suggest the destination reducer for the key k. Thanks to the hash function properties, this operation always produces the same result for each key and has the further advantage of being very fast to execute. Furthermore, the hash function guarantees a certain degree of balancing in the distribution of the <key,values> pairs.

When the program to be implemented through MapReduce is the sort function, the default hash partitioning must be substituted with a partitioning based on the first byte of the key. According to the ordering guarantee principle indeed, each reducer processes (and emits) its pairs in key order. This custom partitioning function also guarantees that no other ordering operation is needed after the reduce phase.

Apache Hadoop

Apache Hadoop [59] is an open-source platform developed by the Apache Software Foundation to enable the execution of MapReduce jobs. It is characterized by a relatively simple COMMAND LINE INTERFACE (CLI) and two main components with master-slave architecture: HADOOP DISTRIBUTED FILE SYSTEM (HDFS) and MapReduce Runtime.

HDFS is a distributed file system whose content is not, generally, replicated but only split between the computing nodes. The master machine runs a *Namenode* daemon that coordinates a collection of *Datenode* daemons executing on the slave nodes. Thanks to this architecture, when a file is copied in HDFS, it is transparently split into blocks distributed to the slaves. The Hadoop CLI offers a collection of shell-like commands to manage the distributed file system and always shows the elements inside as if they were memorized over a single storage device. To improve the recovery ability in case of node failures, HDFS can also be configured to autonomously manage the replication of the memorized data.

MapReduce Runtime is the Hadoop component responsible for the execution of MapReduce programs. It generates and coordinates the map/reduce processes by means of a master-slave architecture. The master machine executes a *JobTracker* daemon that assigns specific tasks to the slave machines by interacting with the *TaskTraker* daemon (one for each slave) and generally preferring a *data-local* computation. Figure 3.7 clarifies the allocation of the daemons to the nodes and the general map/reduce task assignment strategy.

The number of map tasks N_M to be launched is determined automatically by the Hadoop platform depending on the volume of the input data. Generally a map task is created to process a 64MB chunk of input data. Nevertheless, the user can control the number of tasks that can be executed concurrently on the cluster. This value is expressed by configuring the number of map/reduce *slots* hosted by each computing node. A *slot* can be seen as a container where a map/reduce task is placed after being launched. For example, on a dual-core machine with hyperthreading a popular solution is to specify four map slots ($S_M = 4$) and four reduce slots ($S_R = 4$), because 4 is the maximum number of tasks that can be concurrently executed by the hardware without context switches. Actually, $S_M = S_R$ because the reducers use the output of the mappers

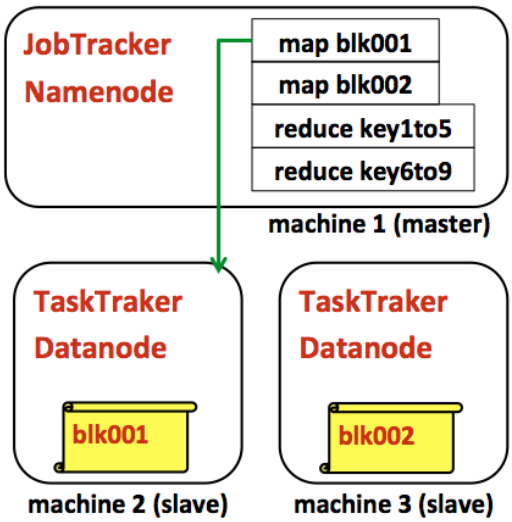


Figure 3.7: When the execution command is issued to the Hadoop cluster through the CLI, the JobTracker on the master node is involved. It interacts with the Namenode daemon to create a number of map/reduce tasks consistent with the number of data blocks to be processed. The JobTracker also allocates the tasks preferring a data-local computation. For example, the map working on block 001 is likely to sent to machine 2 containing that block on its portion of HDFS.

and, therefore, the executions of these two phases do not overlap.

The number of reduce task N_R to be launched is controlled by the user with a specific option of the CLI. Hadoop developers suggests two possible values (obtained through statistical analysis [59]) for N_R depending on the hardware characteristics. In particular, if the computing nodes have similar performance, the number of reduce tasks to be launched should be set to $N_R = 0.95 \times S_R$. In case of cores with different performance, Hadoop developers suggests $N_R = 1.75 \times S_R$.

The rational behind these two values is intuitively in the attempt to obtain the maximum degree of parallelism in the execution. In case of similar cores performance, the value is close to 1 because this will assure to allocate a reduce task to each reduce slot thus to complete the reduce phase in one round. When the core performance are different, the execution time for a given task can vary depending on the slot/core to which the task is assigned. The multiply factor 1.75 makes

N_R higher than S_R, so that while the slower cores continue to execute the reduce tasks of the first round, the faster ones have already completed their task and can start a second round of reducers.

MapReduce fault tolerance and elasticity

Thanks to its distributed architecture MapReduce programming model is particularly useful when a certain level of fault tolerance is required. Indeed, independently from the specific implemented algorithm, two different error situations can be identified and (partially) solved:

- if the master node identifies a fault on a slave – e.g., the salve is no longer responding – but the data are accessible (or replicated elsewhere in the distributed file system), the master can restart the map/reduce task of the faulty slave on another machine;
- if the master identifies errors on some data blocks, can decide to skip them and continue.

These fault tolerance mechanisms are implemented on all main MapReduce platforms and contribute to improve the QoS of the data-parallel approach.

Alongside these mechanisms, Hadoop also implements strategies to improve the scalability of the computing cluster at runtime. If a new slave is added to the infrastructure while some MapReduce job is executing, Hadoop is able to assign pending tasks to it, thus to relieve the other machines and speed up the computation.

This elastic behavior is particularly useful when Hadoop is executed on a scalable infrastructure i.e., an infrastructure that allows the user to easily add or remove computing resources to the cluster, such as the cloud. In this environment VMs can be provisioned (or de-provisioned) on demand, to support the need for additional (or less) computing resource in a Hadoop cluster.

3.3 FRAMEWORK ARCHITECTURE

Our contribution to enable MapReduce over hybrid cloud is a framework architecture [48] based on two separated cloud installations:

- the on-premise IC, owned and managed by a private company;
- and the off-premise EC, a collection of resources owned by a cloud provider and rented to customers according to a predefined price plan.

Having their own cloud management software and offering their virtualized resources to final users (e.g., customers, company employees, etc...), both IC and EC implement the cloud paradigm at IAAS level.

As a case study, we consider the execution of data-intensive applications over clusters of VMs initially deployed on IC. Since the private cloud can be used by multiple users for different goals at the same time, it is likely to happen that a physical machine simultaneously hosts VMs designed for different application domains.

In this scenario, if a physical node hosting a VM for data-processing becomes overloaded in terms of CPU, memory or disk utilization (e.g., as a result of other computations carried on the same physical machine by VMs belonging to other application domains/projects), the performance of the hostedVM may dramatically decrease, thus slowing down the whole virtual cluster for data-processing.

In this case, if another less loaded physical machine is available on-premise, the best solution would be to move the VM on that physical node. The progresses in the techniques for VM migration has brought the possibility to move a VM from a physical machine to another without need to suspend it and suffering a minimal downtime. The fault tolerance features offered by all MapReduce platforms can help to overcome this downtime an make the data analysis continue.

However, the private cloud has a finite amount of resources and it may happen that all the physical machines in IC are too loaded to receive the VM: in this case, we can provide additional resources on EC and perform a redistribution of the computational load. Automating this mechanism – by arranging an autonomous infrastructure able to detect the occurrence of a critical condition on a physical machine and react to that by providing additional off-premise resources to the application level cluster involved – can represent a key advantage for data-processing computation on hybrid environments.

To this purpose we defined a platform for the autonomic management of VMs in hybrid clouds: SYSTEM FOR HYBRID

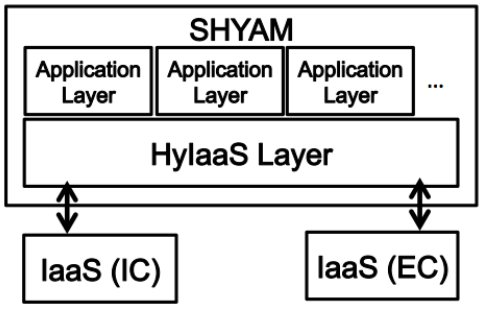


Figure 3.8: Layer architecture of SHYAM system.

CLUSTERS WITH AUTONOMIC MANAGEMENT (SHYAM). In particular, we designed and implemented a software layer able to manage virtual clusters using both on-premise (i.e., computing nodes in a private IC) and off-premise (i.e. in a public EC) hardware resources. The system is able to dynamically react to load peaks – due, for instance, to VM contention on shared computing nodes – by redistributing the VMs on less loaded nodes (either migrating inside IC or crossing the cloud boundaries towards EC).

As shown in Figure3.8, the key component of SHYAM is HYBRID INFRASTRUCTURE AS A SERVICE (HYLAAS), a software layer that enables the integration between IC and EC infrastructures. The layer interacts with both on- and off-premise cloud with the goal of providing hybrid clusters of VMs. Each cluster is dedicated to the execution of a particular distributed application (e.g., distributed data-processing). If there are enough resources available, all the VMs of a cluster are allocated on-premise to minimize the costs introduced by the public cloud and the latency of data transferred between the virtual nodes. If on-premise resources are not sufficient to host all the VMs, a part is provisioned on IC and the others on EC. This partitioning should be transparent to the final user of the virtual cluster, allowing her to access all the VMs in the same way, regardless to the physical allocation. We call *hybrid cluster* the result of this operation.

HYLAAS is also responsible for autonomously handling to changes in the current utilization level of the on-premise physical machines hosting the VMs of the cluster. To avoid the application slowdown due to the poor performance of these VMs, HYLAAS layer is in charge of dynamically spawning new VMs on EC and providing them to the above Application layer (Figure 3.8). This layer is responsible for installing and

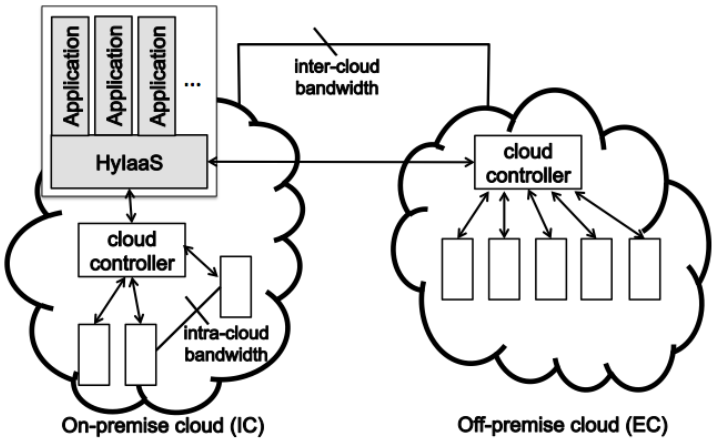


Figure 3.9: Hybrid cloud scenario. SHYAM is an on-premise software component able to collect information about the current status of IC and dynamically add off-premise resources if needed.

configuring a specific distributed application on the newly provided VMs.

SHYAM's main goal is to unify on- and off-premise resources. As shown in Figure 3.9, it must be installed on IC, so that it can collect monitoring information about the utilization level of the on-premise machines. According to a specific user-defined policy, the HyIaaS layer can perform cloud bursting toward EC by translating generic spawning and scale-down requests into specific off-premise provisioning and de-provisioning commands.

We assume both IC and EC have a centralized architecture, as such SHYAM makes them able to cooperate by communicating with the central controllers of IC and EC.

In the following, we will use the term *compute nodes* to refer all the physical machines (of IC or EC) able to host VMs and not in charge of any cloud management task.

Therefore, HyIaaS layer consists of three components (depicted in Figure 3.10): the Monitoring Collector, the Logic and the Translation component.

- The Monitoring Collector is in charge of fetching information about the current resource utilization level of the on-premise *compute node*.
- The Logic component uses the information read by Monitoring Collector and implements a custom-defined spawning policy. Given the current status of the on-premise cluster and additional constraints possibly

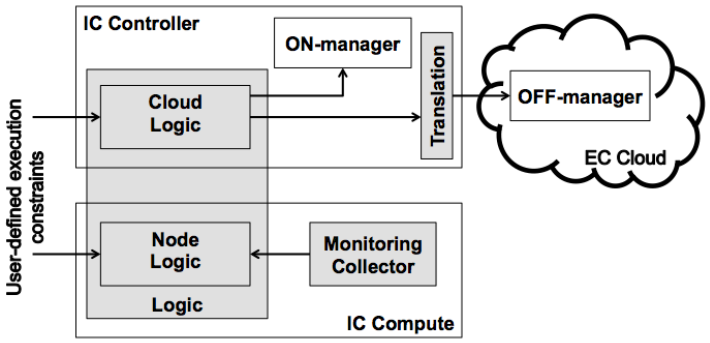


Figure 3.10: Hybrid Infrastructure as a Service layer. The subcomponents are displayed in grey.

introduced by the customer (e.g., deadline for the execution of a certain job), the output of the Logic component is a new allocation of the VMs over the physical nodes, possibly including new VMs spawned off-premise.
- The Translation component converts the on-premise commands into the API specific to the off-premise cloud platform.

The Logic component has been split into two subcomponents:

- the Node Logic (one for each *compute*), responsible for analyzing the monitoring data from the Monitoring Collector, detecting if a critical situation occurred on that physical machine (e.g., the *compute node* is to loaded) and sending notifications to the Cloud Logic;
- the Cloud Logic (installed on IC's controller node), in charge of autonomously taking spawning/migration decisions given the received monitoring alerts from Node Logic.

The alerts from Node Logic and the policy of Cloud Logic can be defined by the IC system administrator. The rational behind splitting the Logic component into two parts is to minimize the amount of information exchanged between the on-premise cloud controller and the physical nodes hosting VMs: the Node Logic sends notifications to the Cloud Logic only if a critical condition at node-level is detected. Having a wider vision of the state of the cloud, Cloud Logic can combine the received information to implement a more elaborate policy.

If the new VM allocation produced by the Logic involves EC, the Translation component is used to convert the directives into EC-specific APIs.

As a case study, we focus on the data-intensive scenario, where the applicative load can be distributed among several computing nodes. In order to control the complexity of parallelization, we adopt the MapReduce model [58] and its open-source implementation platform Apache Hadoop [59].

We execute the Hadoop workload over a virtual cluster that can be deployed on the hybrid cloud (partitioned between IC and EC) in case the on-premise resources are not enough. Therefore, the first application layer we implement for SHYAM is responsible for installing and configuring Hadoop on the newly provided VMs and allows us to evaluate the performance of operating MapReduce in a hybrid cloud setup.

3.3.1 General Cost Model

In the following, we investigate the cost model of a generic application running on the hybrid cluster. The model we obtain is used for the implementation of the Logic component to guide the dynamic resource provisioning mechanism.

Although we expect the cost model to mainly depend on the specific application-level workload executed by the virtual cluster, we can nevertheless point out some general relation that must be specialized depending on the particular application.

Considering all the execution tasks and input data initially distributed over X virtual nodes on IC and a predefined deadline T for the computation to be completed, we want to investigate under which hypothesis we can take advantage from spawning and how many VMs we need to provide on EC in order to satisfy the deadline constraint. The number Y of off-premise VMs to be spawned can be expressed as follows:

$$Y = f_A(D, X, L, T), \tag{3.2}$$

where D is the amount of data to be processed, L is the inter-cloud bandwidth and f_A is a function strictly related to the running application A. The deadline T must be compared to the predicted total time T_{tot_H} needed for the computation to take place on the hybrid cluster. As we start from a fully on-premise cluster of VMs, T_{tot_H} can be expressed as follows:

$$T_{tot\_H} = T_p + T_c + T_e, \tag{3.3}$$

where T_p is the time to have Y VMs provided on EC up and running, T_c is the time to configure these newly provided VMs and T_e is the effective time to execute the job on the resulting hybrid cluster.

Depending on the EC IaaS layer only, the provisioning time T_p can be estimated regardless to the specific application running on the hybrid cluster. When a spawning command is issued to EC, the scheduler of the off-premise cloud infrastructure must determine which physical machine can host each newly provided VM. Intuitively, we expect the provisioning time to be as follows:

$$T_p = o(Y) + t_{s\_ov}, \tag{3.4}$$

where $o(Y)$ is a component of T_p directly proportional to the number Y of spawned VMs and is strictly connected to the EC infrastructure performances. On the other hand, t_{s_ov} is a constant overhead introduced by the EC scheduler. T_p can be seen as an application-independent lower bound for T_{tot_H} because no computation can start on the off-premise VMs before the provisioning operation is completed.

Unlike T_p, the configuration and execution times (T_c and T_e) depend on the application-level tasks operated by the hybrid cluster. In the following section we describe T_c and T_e with reference to the particular execution scenario of MapReduce data-intensive applications.

3.3.2 Hybrid MapReduce Cost Model

Focusing on Hadoop implementation of MapReduce, the Job-Tracker on the master node assigns jobs to the workers according to the part of data currently allocated on the worker's portion of HDFS.

Having no data initially allocated on the newly provided off-premise workers, they will be scarcely useful for the computation, because the Job-Tracker will not assign any task to them. Therefore, in case of Hadoop application, the configuration time T_c in Equation 3.3 must include the time to perform the distribution of the data (and the consequent task allocation) across all the virtual nodes.

Hadoop platform offers a data balancing feature called Hadoop Balancer [59] that can be requested on-demand with a specific command or continuously executed by a daemon process. Whenever needed, the Balancer analyze and compare

the utilization of the portion of HDFS on each worker. If the utilization differs from the average such that it exceeds a tolerance interval (default set to [-10,10]%), the balancing process starts to move data blocks from the workers with higher HDFS utilization to the underutilized ones.

We call "balancing time" T_b the amount of time needed for this operation and we express T_c as follows:

$$T_c = T_b + t_{c\_ov} \tag{3.5}$$

where t_{c_ov} is the constant overhead introduced by generic configurations (e.g., changes in the configuration files, daemon start-up, etc.)

In a hybrid scenario, T_b is strictly influenced by the bandwidth L between IC and EC and can be expressed as:

$$T_b = \frac{\delta(X, Y, D)}{L} + \tau(X, Y, D, L) \tag{3.6}$$

where δ is a function expressing the amount of data crossing the on-/off-premise boundaries and τ addresses the time requested by Hadoop Balancer to detect which blocks of data must be moved and perform additional data transfer.

While it is rather simple to estimate the minimum amount of data we need to transfer from on- to off-premise cloud (to have input data balanced across all the VMs), it is harder to determine how much time is needed by Hadoop Balancer as it performs additional data movement. Hence, τ must be estimated through testing. However, this estimation has the advantage of being independent from the specific MapReduce workload going to be issued to Hadoop framework.

As regards the T_e parameter in Equation 3.3, we can note that the execution phase of MapReduce application over a Hadoop hybrid cluster has no conceptual differences from the execution on a set of on-premise VMs (assuming the same number of virtual servers with the same computing power in both the scenarios). Nevertheless, it is crucial to consider that in the hybrid case, Y off-premise VMs are reachable through a lower bandwidth medium.

We expect the execution time $T_{e\_H}(Z)$ in a hybrid scenario with Z VMs to be higher than the execution time $T_{e\_I}(Z)$ on a fully on-premise cluster of Z co-located VMs, resulting in the following:

$$T_{e\_H}(Z) \geqslant T_{e\_I}(Z) \tag{3.7}$$

In the case of a Hadoop application, the *mapper* tasks mainly work on local data, while the *reducers* collect input data from *mappers*' output. For this reason the delay of hybrid execution is mainly due to Hadoop's shuffle phase (i.e., when the intermediate output from the *mappers* is sent to the *reducers*). Since the amount of data shuffled depends on the workload and the particular data set involved, the execution time T_e in Equation 3.3 can only be experimentally estimated.

3.3.3 Implementation details

We implemented HyIaaS by extending OpenStack Sahara [109] component to allow cluster scaling operations in a hybrid scenario.

As described in Chapter 3.2.1, OpenStack [5] is an open source platform for cloud computing with a modular architecture and Sahara is the OpenStack module specific to data processing.

OpenStack Sahara allow the user to quickly deploy, configure and scale virtual clusters dedicated to data intensive applications like MapReduce. We modified Sahara scaling mechanism to allow the spawning of new VMs on a remote cloud.

The Monitoring Collector component is a simple daemon process running on each compute node and collecting the percentage of physical resource utilization. It checks the CPU, RAM and disk usage on the compute nodes and sends this information to the Logic component where it is used to detect the need for new resources. The monitoring data can be further coupled with those collected by the OpenStack Ceilometer module [110] (i.e., utilization of virtual resources on each VM).

When the virtual cluster needs to be scaled by providing new off-premise VMs, the command is issued through the Translation component to EC. In our test scenario, the off-premise cloud runs another OpenStack installation, therefore the Translation component simply forward the provisioning command to EC's OpenStack component in charge of VM management – i.e., OpenStack Nova Compute module.

The MapReduce Application layer finally configures Hadoop and launches its daemons by connecting to the newly provided VMs.

Name	VCPUs	RAM(GB)	Disk(GB)
m1.small	1	2	20
m1.medium	2	4	40
m1.large	4	8	80
m1.xlarge	8	16	160

Table 3.5: Characteristics of the VMs considered for T_p evaluation depicted in Figure 3.11a

3.3.4 Experimental Results

We relay on SHYAM to conduct a first set of tests in order to better understand Hadoop behavior when executed on a hybrid cluster. Further in this dissertation (Chapters 3.4 and 3.5), we describe and evaluate two different policies we implemented for the Logic component with the aim to improve the hybrid cluster performance.

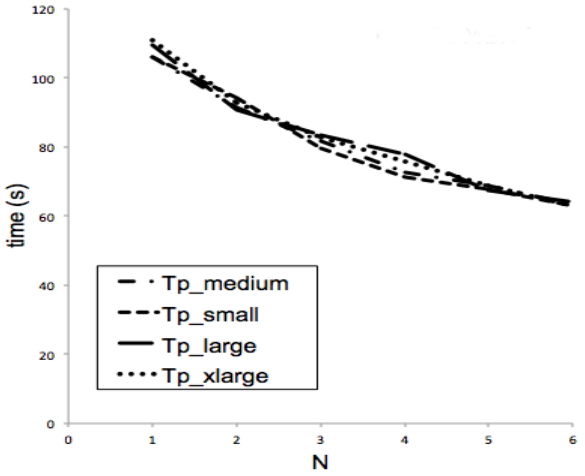
Our setup is composed of two OpenStack clouds to emulate IC and EC. The on-premise cloud has five physical machines, each one with a Intel Core Duo CPU (3.06 GHz), 4GB RAM and 225GB Hard Disk Drive (HDD). EC is composed of three physical machines, each one with 32 cores Opteron 6376 (1.4 GHz), 32GB RAM and 2.3TB HDD.

On both IC and EC we provide VMs with two Virtual Central Processing Units (VCPUs), 4GB RAM and 20GB of disk. The intra-cloud bandwidth of IC and EC is 1000Mbit/s while the inter-cloud bandwidth L is at most 100Mbit/s.

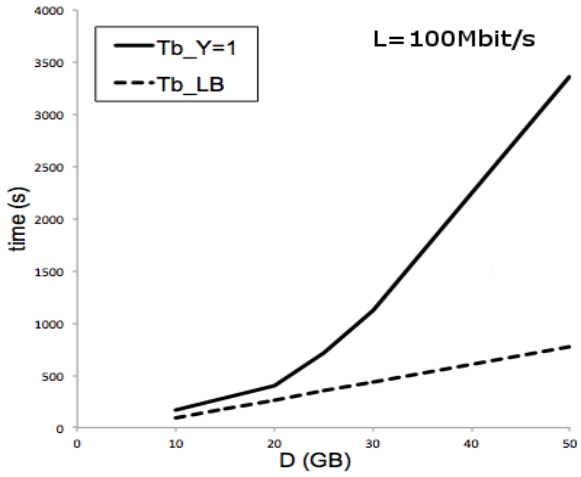
Our initial scenario is composed of five VMs (one *master* and four *workers*) allocated on IC. In order to characterize the computation performance on the hybrid cluster, we separately study the times in Equation 3.3.

As regards the time T_p to provide a new VM on EC, we can easily verify in Figure 3.11a that it is independent from the characteristics (number of VCPUs, RAM and disk size) of the specific VM spawned (detailed in Table 3.5).

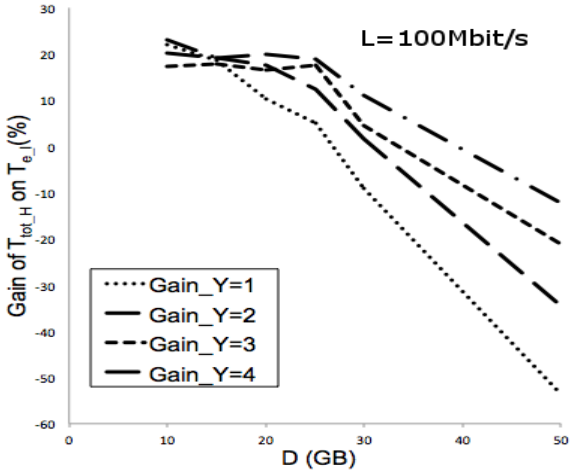
Figure 3.11a shows the provisioning time for a single VM as we vary the number N of VMs spawned at a time on EC. As we expected, the trend of the curve suggests that there is a constant overhead caused by the cloud provisioning mechanism, thus verifying the assumption of Equation 3.4.



(a)



(b)



(c)

Figure 3.11: Performance of HyIaaS in a hybrid scenario.

We assume to have four on-premise VMs already configured to be Hadoop workers and provided with a certain amount of data D on HDFS. We use HyIaaS to add a new VM to the cluster by spawning it on EC.

The new VM is configured to be an Hadoop worker but, since it will initially have no data on its portion of HDFS, it will be scarcely useful to improve the performance of the existing cluster. A data balancing phase is therefore necessary.

Figure 3.11b shows the time $T_{bY=1}$ to balance the cluster after the provisioning of a single ($Y = 1$) off-premise VM for increasing amount of data. According to Hadoop default configuration, we assume a node balanced if it is loaded as the average of the cluster considering a 10% tolerance threshold.

Having a fixed value L=100Mbit/s for inter-cloud bandwidth, we can calculate the time T_{b_LB} to send the minimum amount of data to the off-premise VM as follows:

$$T_{b\_LB} = \frac{Y}{L}\left(\frac{D}{X+Y} - \frac{C}{100}t\right) \tag{3.8}$$

where X and Y are the number of on- and off-premise VMs respectively ($X = 4$ and $Y = 1$ in this testbed), D is the total amount of data, C is the disk capacity of each VM provided and t is the balancing threshold (10% in our setup).

T_{b_LB} is linear in the total amount D of data involved but, as we can see in Figure 3.11b, for high values of D, it is not the main component of the balancing time. This is due to additional balancer computation and additional data to be transferred inter- and intra-cloud. Considering Equation 3.6, we can therefore conclude that T_b is non-linear in the amount of data D and its value is mainly due to the contribute of τ.

On top of T_p and T_b we must consider the time T_e to execute a specific MapReduce application. We chose the word count application [59] as a benchmark workload for the following test and we apply it on wikipedia datasets of different size D [111]. We force the hybrid cluster to have a single *reducer* task physically allocated on one of the on-premise VMs. With this configuration, after the *mapper* phase (mainly involving local data), the off-premise *mappers*' output must cross the cloud boundaries to be processed by the single *reducer*.

We compare the time to execute on a fully on-premise cluster (4 VMs) with the total time T_{tot_H} to deal with the same amount of data over a hybrid cluster with increasing number

of off-premise VMs. The values on the y-axis of Figure 3.11c are calculated as follows:

$$Gain(\%) = \frac{T_{e\_I} - T_{tot\_H}}{T_{e\_I}} 100 \tag{3.9}$$

Figure 3.11c underlines that we can obtain a significant gain in time for low amount of data but for high values of D the execution over a lower number of co-located nodes is effectively less time consuming. To fully understand Figure 3.11c, we investigate the T_e contribute to T_{tot_H} separately.

Figure 3.12 compares the time T_e_I to execute wordcount over 4 on-premise nodes with that of a hybrid cluster (including 4 IC VMs and up to 4 EC VMs).

In case of a limited inter-cloud bandwidth (L=10Mbit/s in Figure 3.12a), we see that there is no advantage in adding off-premise nodes to the cluster because the data exchange during the shuffle phase is too time consuming.

On the other hand, if the inter-cloud bandwidth is higher (L=100Mbit/s as in Figure 3.12b), we can see a significant gain in execution time when the off-premise nodes are added to the virtual cluster. As shown in Figure 3.12b, when the bandwidth is not a bottleneck for the system, the more is the number of VMs provided on EC, the higher is the gain in the execution time. However, is relevant to notice that, while over a fully on-premise cluster T_e is linear in D, on a hybrid cluster this relation is no longer linear and for high values of D a fully on-premise execution can save more time.

3.3.5 Discussion

In this chapter we presented HyIaaS, a software component to allow VM management and configuration in a hybrid cloud scenario. We illustrated the architecture of the HyIaaS component and its internal structure. Finally, we tested our system by executing a Hadoop data-intensive application on hybrid clusters, where some of the worker nodes are located off-premise and might be reachable through a higher latency medium. Our results show that, although the total cost to execute a benchmark application over a hybrid cluster is strongly influenced by the workload, some components of this cost model are independent from the application running and can therefore be *a priori* estimated.

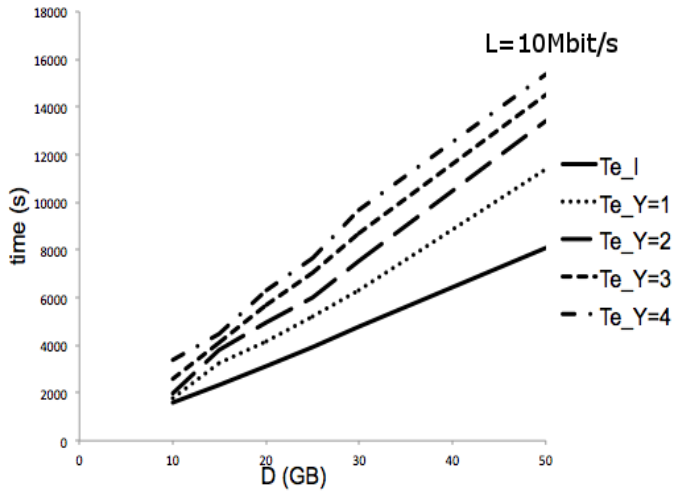


Figure 3.12: Comparison of times T_e to execute on IC and EC varying the volumes of data D and the number Y of off-premise spawned VMs. 3.12a shows the case of a limited inter-cloud bandwidth (L=10Mbit/s), while 3.12b illustrates the behavior with a higher bandwidth (L=100Mbit/s)

This work represents a first step towards the implementation of an autonomic "Hybrid Infrastructure as a Service" system, able to forecast the computing resources needed by each on-premise application and to autonomously request them to EC. We used HyIaaS for the quick deploy of hybrid clusters of VMs aiming to execute a MapReduce workload by means of the Hadoop framework.

The execution time model needs to be refined by analyzing other MapReduce workloads and studying the connections with the volumes of data exchanged during the Hadoop shuffle phase. Since the processing time of a MapReduce application in a hybrid cluster is also influenced by the number and the allocation (on- or off-premise) of the reducers, further investigations of the performance are needed when more then one reduce task is running.

Nevertheless, the results of this work are useful to clarify the trends of execution times for MapReduce applications over an interconnected multiple cloud scenario. The study shows that the main obstacle to the final goal of enabling MapReduce over an hybrid cluster of VMs is the limitation introduced when the inter-cloud bandwidth is saturated. Indeed, this factor is responsible of the evolution in the execution time from a linear trend (proportional to the volumes of data involved) on a fully on-premise cluster to a non-linear trend on a hybrid environment. It is therefore necessary to contain the amount of data flowing through the on-/off-premise boundaries, eventually avoiding to appeal to Hadoop Balancer process for the initial redistribution of the input data.

Focusing on the scenario described in Chapter 3.3, we continue this dissertation by providing two management policies for HyIaaS Logic component aiming to both perform cloud bursting when needed and contain the volumes of data spawned off-premise.

3.4 THE SPAN POLICY

We consider the scenario of a private cloud used for MapReduce data-processing and for various other goals at the same time. In this case, it is likely to happen that a physical machine simultaneously hosts VMs designed for different application domains. If a compute node hosting a MapReduce worker VM becomes overloaded as a result of

other computations carried on the same physical machine by VMs belonging to other projects, the performance of the MapReduce worker may dramatically decrease, thus slowing down the whole virtual cluster for data-processing.

For this reason, we implemented a first example of policy for the Logic component executed on every compute node of IC: the SPAN policy (Algorithm 9). The work included in this chapter was published in [98].

SPAN aims to maintain the load of each compute node under parametric threshold THR_U. It periodically checks the resource utilization of the compute node h (line 2 in Algorithm 9). If the load exceeds THR_U, the procedure selects to move a subset of the VMs currently on h (line 3 in Algorithm 9). The $selectToMove$ function is implemented according to MoM algorithm from Beloglazov et al. [39]. This policy ensures to always move the minimum number of VMs that brings h utilization back under THR_U.

For each vm selected, if there is another on-premise node that can host the VM, a migration is performed (line 7 in Algorithm 9). Otherwise, if no IC's *compute* node can host the VM, a new one is spawned off-premise and the specific application level operation is performed ($swap()$ in line 10 of Algorithm 9).

We chose Hadoop as an example of distributed data-processing platform. For this reason, *swap* procedure (Algorithm 10) simply consists of performing an Hadoop installation on vm_{new} and including it in the cluster while the old on-premise vm is excluded by performing Hadoop decommissioning procedure [59]. This operation automatically copies to the other workers of the cluster the portion of HDFS data on vm and marks it to be excluded from task assignment. However, Hadoop does not really delete the data on the decommissioned worker: they remain on the old vm unless it is actually remove. Hence, we perform $ONmanager.remove(vm)$ in the SPAN algorithm once the *swap* procedure has competed (line 11 in Algorithm 9).

Focusing on the $cl.include(vm_{new})$ operation in line 3 of Algorithm 10, we must consider that Hadoop's Job-Tracker (running on the *master* node) assigns jobs to the workers according to the part of data currently allocated on the worker's portion of HDFS. Having no data initially allocated on the newly included off-premise workers, they will be scarcely

Algorithm 9: SPAN policy

input: h, $ONmanager$, $OFFmanager$, THR_U, Δt

```
1  while true do
2      if h.getUtil() > THR_U then
3          vmsToMove = selectToMove(h.getVMs()) ;
4          foreach vm ∈ vmsToMove do
5              d = ONmanager.getAnotherAllocation(vm) ;
6              if d != null then
7                  ONmanager.migrate(vm, d) ;
8              else
9                  vm_new = OFFmanager.provideLike(vm);
10                 swap(vm, vm_new) ;
11                 vm_new = ONmanager.remove(vm);
12             end
13         end
14     end
15     sleep(Δt);
16 end
```

useful for the computation, because the Job-Tracker will not assign any task to them.

A first idea to solve this problem, could be to provide vm_{new} on EC, maintain the old vm on IC and simply launch Hadoop Balancer process [112], which is in charge of equally redistributing data across the workers. However, it is possible to verify that the balancing mechanism is highly time-consuming [113] because it does not simply move the data from the overloaded IC worker towards the newly provided off-premise VM, but also tries to balance the amount of data across all the workers, causing further balancing

Algorithm 10: $swap(vm, vm_{new})$ procedure for Hadoop virtual cluster

input: vm, vm_{new}

```
1 installMR(vm_new) ;
2 cl = vm.getVirtualCluster() ;
3 cl.include(vm_new) ;
4 cl.decommission(vm) ;
```

computation and additional data to be transferred inter- and intra-cloud.

Another solution cloud be to perform an inter-cloud migration by transferring off-premise a snapshot of the overloaded VM. This operation would ensure to move the minimum portion of HDFS to EC, but will also cause other parts of the VM (operating system, configurations, applications, etc.) to be transferred. Furthermore, given the intrinsic heterogeneity of the hybrid cloud environment, the inter-cloud live mobility is currently a challenging field [85], and can have poor performance due to the large size or the VM snapshot involved and the limited inter-cloud bandwidth.

For this reason, our solution is to relay on a well known Hadoop behavior: when the on-premise vm is decommissioned and its data are replicated, Hadoop prefers the workers with low utilization of HDFS as destinations. Initially having 0% HDFS utilization, off-premise vm_{new} is likely to be preferred and no other data balancing is needed to give vm_{new} an effective role in computation. Therefore, $cl.decommission(vm)$ in line 4 of Algorithm 10 is enough to trigger the data replication process and avoid the drawbacks of Hadoop data balancing process. It also produces the benefits of an inter-cloud VM migration (i.e., the minimum portion of HDFS is moved to EC) without performing the whole VM snapshot transfer.

3.4.1 Experimental results

In order to test the performance of SPAN policy, we relay on the same infrastructure used for the validation of the cost model (Chapter 3.3.4).

In this scenario, we assume to have four on-premise worker VMs already configured to run Hadoop jobs and provided with a certain amount of data D on HDFS.

We consider the time to execute a word count Hadoop job [108] over Wikipedia datasets of different size D [111]. Figure 3.13a compares the execution time trends of three scenarios. The first one (Te_I in Figure 3.13a) represents the ideal situation of having each Hadoop VM allocated on an on-premise dedicated physical machine. Since no other physical or virtual load is affecting the execution, we can obtain good performance (execution time is linear in D).

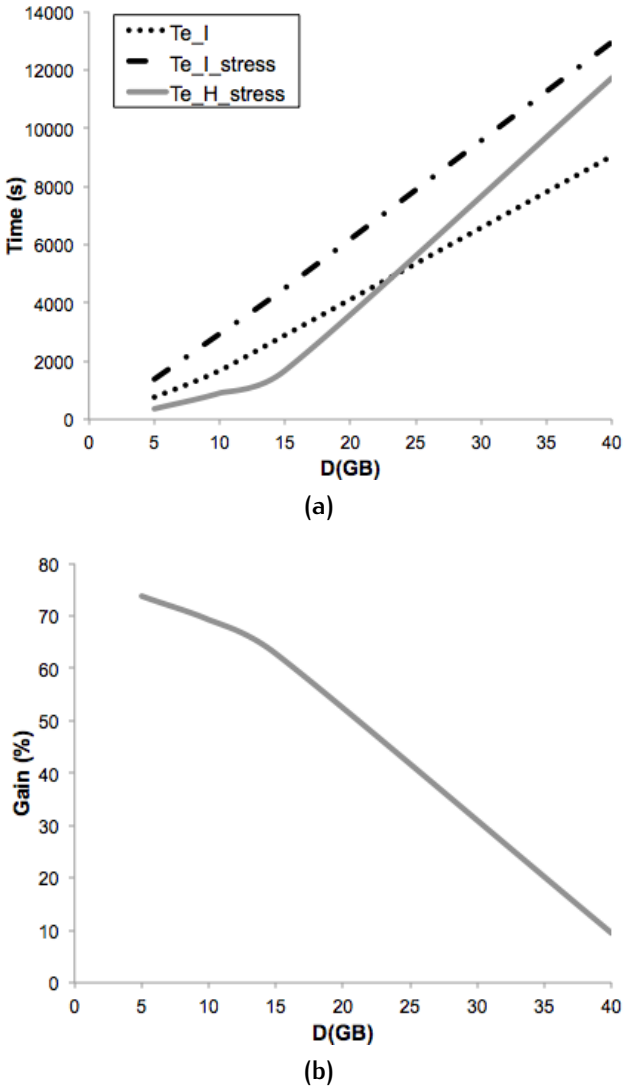


Figure 3.13: Performance of HyIaaS in a hybrid scenario. Figure 3.11a shows the time to provide new off-premise VMs with different characteristics. Figure 3.13a compares the time to perform a Hadoop word count workload on a fully on-premise cluster - with (Te_I) or without (Te_I_stress) a stressing condition on a physical node -, with the performance on a hybrid cluster created by the HyIaaS layer. Figure3.13b shows the percentage gain obtained by our solution.

The second scenario (Te_I_{stress} in Figure 3.13a) shows the performance degradation when one of the four Hadoop workers is running on a overloaded physical machine and no VM redistribution mechanism is adopted. As we can see in Figure 3.13a, the execution time is considerably higher when

compared to Te_I because the VM on the stressed physical node sensibly slows down the whole distributed computation.

The third scenario (Te_H_{stress} in Figure 3.13a) repeats the second scenario and uses SHYAM infrastructure with SPAN policy. THR_U is fixed at 90%. In this case, a new VM is spawned off-premise and the on-premise worker running on the stressed machine is decommissioned (i.e., excluded from Hadoop cluster after its data have been copied on other worker nodes). This operation causes a part of data to cross on-/off-premise boundaries.

As we can see in Figure 3.13a, the adoption of SPAN policy can considerably improve the performance for low values of D. However, (as already noticed in Chapter 3.3.4) the trends show that in the hybrid scenario the execution time is no longer linear in the volume of data involved and, for high values of D, the time gain determined by the off-premise resources is considerably reduced. This is mainly due to data movement across the on-/off-premise boundaries, which is usually over a higher latency medium when compared to a fully on-premise computation.

Figure 3.13b shows the gain in execution time obtained with SHYAM. Although for high volumes of data crossing on-/off-premise boundaries the gain is low, the graph suggests that the autonomic provisioning and configuration of VMs on EC can represent a good solution to face critical conditions of stress in private clouds.

3.4.2 Discussion

Relaying on SHYAM system, we developed a new policy for the management of data-processing clusters of VMs in a hybrid cloud environment.

We evaluate SPAN policy performance by executing a Hadoop application on a virtual cluster and stressing one of the IC's physical machines. In the given scenario, SHYAM autonomously spawns new VMs on EC and configures them as workers of the Hadoop cluster.

Our results show that the hybrid cluster obtained can sensibly improve the performance of a benchmark Hadoop word count application. However, for high volumes of data crossing on-/off-premise boundaries, the performance of the hybrid cluster decreases as the inter-cloud bandwidth is saturated. Since this drawback could be also influenced by the

kind of application executed (e.g., word count in our case study), we need to further investigate SHYAM performance with different Hadoop workloads.

Furthermore, in SPAN policy we trigger the spawning/migration mechanism if the physical machine's CPU utilization exceeds THR_U. In the following chapter, we modify the policy to take into account the utilization of other resources (RAM and disk.).

In case of spawning new VMs towards EC, SPAN approach lacks a mechanism for "bringing back" on IC the off-premise VMs once the critical condition is solved. Therefore, as a natural evolution of SPAN, the policy in the next chapter implements a symmetric threshold mechanism to detect underloaded hosts in IC.

Finally, as SPAN policy is actually performed only on the compute nodes, it lacks a mechanism to coordinate the whole infrastructure. This can result in inconsistent evolutions of the algorithm, specially if we deal with further complicated policies. Such kind of situations are avoided in the following by introducing a policy central controller that coordinates the reactions of the compute nodes to critical condition.

3.5 THE HYMR POLICY

Given the limits of SPAN policy depicted in the previous chapter, we improved the HyIaaS Logic component policy to address the scale-down issue and take into account other hardware resources in addition to CPU utilization.

The result is Hybrid MapReduce (HyMR) [99], a policy in charge of the autonomic scaling of Hadoop clusters over the hybrid cloud. Similarly to SPAN, this policy also performs application level tasks, such as installing and configuring Hadoop on the newly provided VMs. HyMR is split into two parts: the HyMR Node and HyMR Cloud policies, as suggested by the Logic component implementation (Figure 3.10).

HyMR Node policy (Algorithm 11) is executed on every compute node of IC and aims to maintain the resource utilization between two parametric thresholds. It periodically checks the CPU, RAM and disk utilization of the compute node h (line 3 in Algorithm 11) and, if one of these values exceeds the correspondent threshold, the procedure selects to move a subset of the VMs currently on h (line 5 in Algorithm 11).

Algorithm 11: HyMR Node policy

input: h, CloudL, ONmanager, OFFmanager, td_U, tc_U, tr_U, td_D, tc_D, tr_D, Δt

```
1  while true do
2      (d, c, r) = h.getDiskCpuRamUtil();
3      if d > td_U or c > tc_U or r > tr_U then
4          CloudL.notifyCritical(c, r, d, h) ;
5          vmsToMove = selectToMove(h.getVMs()) ;
6          foreach vm ∈ vmsToMove do
7              h_new = ONmanager.migrate(vm) ;
8              if h_new = h then
9                  vm_new = OFFmanager.provideLike(vm);
10                 swap(vm, vm_new) ;
11                 if d > td_U then
12                     vm_new = ONmanager.remove(vm) ;
13                 end
14             end
15         end
16         (d, c, r) = h.getDiskCpuRamUtil() if
           d < td_U and c < tc_U and r < tr_U then
17             CloudL.solved(c, r, d, h) ;
18         end
19     else if (d < td_D and c < tc_D and r < tr_D) and
       OFFmanager.VMs.length > 0 and
       CloudL.getScaleDownPermit(h) then
20         vmsToMove = selectToMove(OFFmanager.VMs) ;
21         foreach vm ∈ vmsToMove do
22             vm_new = ONmanager.provideLikeOn(vm, h);
23             swap(vm, vm_new) ;
24             vm_new = OFFmanager.remove(vm);
25         end
26     end
27     sleep(Δt);
28 end
```

The *selectToMove* function is implemented according to the MoM algorithm from Beloglazov et al. [39]. This policy ensures to always move the minimum number of VMs that brings h utilization back under the exceeded threshold.

Furthermore, if we assume that the VMs are "small" – in terms of CPU, RAM and disk characteristics – when compared to the physical machine hardware and their resource utilization does not substantially obscillate, the Minimization of Migration policy also guarantees to avoid glitches in the scale-up/-down mechanism, i.e., constantly moving VMs back and forth between IC and EC.

For each vm selected, if there is another on-premise node that can host the VM, a migration is performed (line 7 in Algorithm 11). Otherwise, if no IC's compute node can host the VM, a new VM is spawned off-premise and the *swap* procedure is called in order to reconfigure the Hadoop cluster (Algorithm 10). As clarified in the previous chapter, the *swap* procedure trigger the data replication process avoiding the drawbacks of Hadoop data balancing process and produces the benefits of an inter-cloud VM migration (i.e., the minimum portion of HDFS is moved to EC) without performing the whole VM snapshot transfer. Note that, *swap* operation automatically copies the portion of HDFS currently on vm to the other workers of the cluster and marks vm to be excluded from task assignment. However, Hadoop does not really delete the data on the decommissioned worker: they remain on vm in case it will be included back in the cluster in the future.

For this reason, after the *swap* procedure, if the critical condition that caused the spawning was a disk overloading, we need to instruct the on-premise cloud controller to actually remove the vm (line 12 in Algorithm 11). On the other hand, if the critical condition was related to RAM or CPU overloading, we prefer to maintain vm and its data on IC (although it has been decommissioned) to speed up future scale-down operations.

When the Monitoring Collector component detects that either the CPU, RAM or disk utilization of the compute node is under the corresponding threshold and some VMs of the current virtual cluster are allocated on EC, an analogous set of operations is performed (lines 19-25) to bring back all (or part of) the off-premise VMs, thus to minimize the cost of public cloud resource utilization.

In this scale-down scenario, if a copy of the data on the off-premise VMs going to be decommissioned is also available on-premise, the operation can be performed fastly. We refer it as Fast Scale Down (FSD). On the contrary, if the portion of HDFS off-premise is not replicated on IC – e.g., if we were obliged to perform a $remove(vm)$ to satisfy the disk utilization constraint as in line 12 of Algorithm 11 –, the scale down will cause all the data on the off-premise VM to be copied on-premise before the decommissioning process can end. We refer this case as Complete Scale Down (CSD) in the following.

Since HyMR Node policy is executed in a decentralized way on each compute node, we need a mechanism to coordinate scale-up and -down decisions in order to avoid the underloaded physical hosts of IC to receive off-premise VMs when some other IC's host is overloaded (and would therefore benefit from an intra-cloud migration). This mechanism is realized by the HyMR Cloud policy (running on the cloud controller), which receives the notification of critical condition occurred and solved from each physical hosts of the cloud through $notifyCritical()$ and $solved()$ operations in line 4 and 17 of Algorithm 11. The Cloud policy temporary stores this information and uses it to grant or deny the scale-down permission to underloaded hosts ($getScaleDownPermit(h)$ in line 19)

3.5.1 Experimental results

The performance of the cluster are evaluated on the infrastructure described in Chapter 3.3.4 by executing three different Hadoop workloads: word count, inverted index (over Wikipedia datasets of different size D [111]) and tera sort (over a collection of rows randomly generated with teragen application [59]).

In order to obtain the best performance for the reduce phase in a hybrid setup, we chose to launch 8 reduce tasks (one for each core in the cluster) in all the experiments, making sure that they are executed both on- and off-premise, as suggested by the results of the work in [88].

When the Monitoring Collector installed on a physical machine detects a critical condition according to HyMR policy, one or more VMs are spawned off-premise. As proven in Chapter 3.3.4 (and detailed in [46]) the provisioning time is linear in the number of provisioned VMs (a part from a

constant overhead introduced by HyIaaS) and independent from their characteristics. Therefore, in the following, we overlook provisioning and investigate the execution and decomissioning time only.

Graphs in Figure 3.14a, 3.14b and 3.14c respectively show the performance of word count, inverted index and terasort. In particular, each graph compares the execution time trends of three scenarios: Te_I, Te_I_{stress} and Te_H_{stress}.

The first one (Te_I) represents the ideal situation of having each Hadoop VM allocated on an on-premise dedicated physical machine. Since no other physical or virtual load is affecting the execution, we can obtain an execution time linear in D for all the workloads.

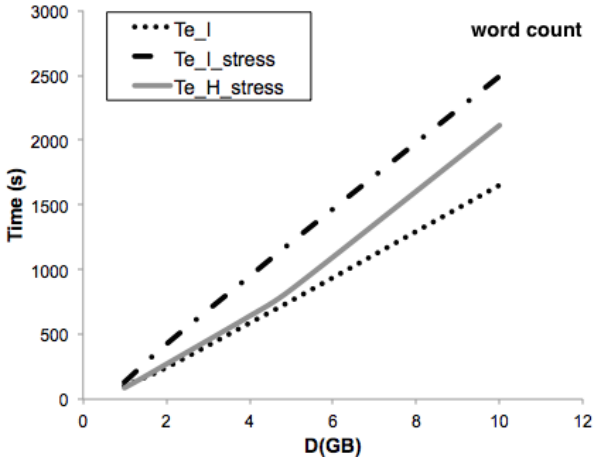
The second scenario (Te_I_{stress}) shows the performance degradation when one of the four Hadoop workers is running on a overloaded physical machine and no VM redistribution mechanism is adopted. As we can see in Figure 3.14a, 3.14b and 3.14c, the execution time is considerably higher when compared to Te_I (particularly for word count and inverted index) because the VM on the stressed physical node sensibly slows down the whole distributed computation.

The third scenario (Te_H_{stress}) repeats the second scenario and adopts HyIaaS infrastructure with HyMR policy. tc_U, tr_U and td_U are set at 90%, while tc_D, tr_D and td_D are 10%. In this case, a new VM is spawned off-premise and the on-premise worker running on the stressed machine is decommissioned (i.e., excluded from Hadoop cluster after its data have been copied on other worker nodes). This operation causes a part of data to cross the on-/off-premise boundaries.

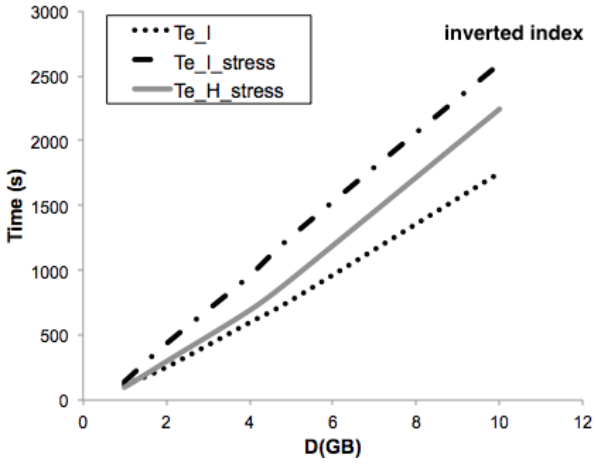
All the stress tests in 3.14 are conducted by artificially stressing the RAM of the physical machine but analogous results can be obtained by stressing the RAM or disk.

Figure 3.15 shows the gain in execution time obtained by our solution (Te_H_{stress} compared to Te_I_{stress}) in case of one physical machine stressed. This graph is useful to better clarify the trends of Figure 3.14a, 3.14b and 3.14c. For all the workloads, the gain decreases as we augment the volume of input data D. This is mainly due to data movement across the on-/off-premise boundaries, which is usually over a higher latency medium when compared to a fully on-premise computation.

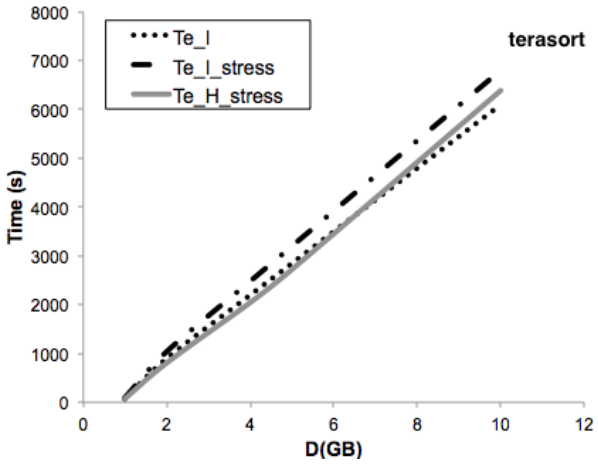
In particular, both word count and inverted index show a stronger gain when compared to terasort. This is due to



(a)



(b)



(c)

Figure 3.14: Performance of HyMR policy and HyIaaS system with different MapReduce workloads.

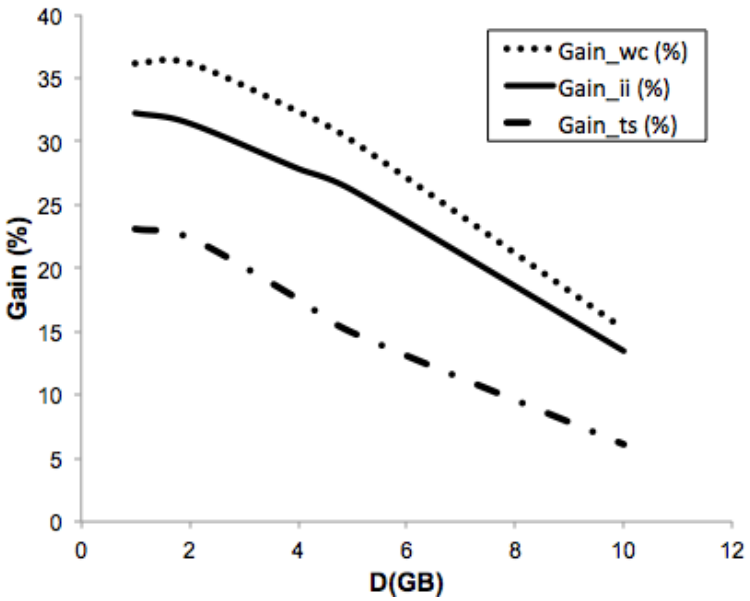


Figure 3.15: HyMR gain in execution time for word count, inverted index and tera sort workloads.

different volumes S of data exchanged during Hadoop shuffle phase (when intermediate data are sent from mappers to reducers [59]). Effectively, while the mappers usually work on local blocks of data, the reducers collect information from other workers and are therefore more influenced by the network latency.

Given the standard implementation of word count (with *Combiner* function) and inverted index in [108], it is easy to understand that the intermediate keys generate a reduced amount S of data to be shuffled (and possibly crossing cloud boundaries) when compared to the input data D. On the other hand, terasort implements the mapper as an identity function resulting in S to be equal to D.

Since in a hybrid scenario a part of the shuffled data S must cross the on-/off-premise boundaries facing a higher latency, the difference between word count/inverted index and terasort results in different trends of the gain function in Figure 3.15.

Finally, we evaluate the scale-down performance of HyMR policy in Figure 3.16 considering both FSD and CSD processes. As we expected, since the CSD causes all the portion of HDFS on the decommissioning node to cross the cloud boundaries, the process is much more time consuming than FSD.

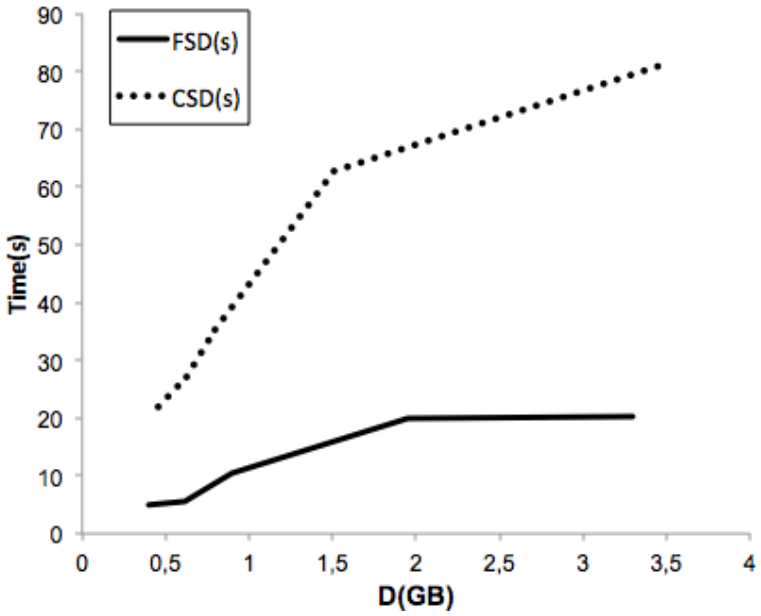


Figure 3.16: HyMR scale-down performance.

3.5.2 Discussion

In this chapter we presented HyMR, an evolution of SPAN policy for HyIaaS component that enables VM cluster autonomic scaling in a hybrid cloud scenario dedicated to MapReduce workload execution. We evaluated the policy performance in the same scenario illustrated in Chapter 3.4 (i.e., by executing a Hadoop data-intensive application on a virtual cluster and stressing one of IC's physical machines).

HyMR policy autonomously spawns new VMs on EC and configures them as workers of the Hadoop cluster. It is also able to scale-down the cluster by bringing back the VMs previously spawned towards EC once the on-premise critical condition is solved. The double threshold mechanism, the introduction of the Cloud Logic algorithm and the adoption of the MoM [17] contribute to reduce glitches in the scale-up/-down process.

Albeit for high volumes of data crossing on-/off-premise boundaries, the performance of the hybrid cluster inevitably decreases as the inter-cloud bandwidth is saturated, our results show that the hybrid cluster obtained can sensibly improve the performance of benchmark Hadoop applications. The percentage of gain determined by SHYAM with HyMR policy is influenced by the MapReduce workload executed as it

determines the volumes of data crossing the on-/off-premise boundaries.

In HyMR policy, we trigger the spawning/migration mechanism as a consequence of a threshold constraint violation in the percentage of resource utilization of every IC's physical machine. However, the policy could be easily modified to take into account more complex constraints (e.g., the speed in resource utilization change).

For the future, SHYAM system would benefit from a further evaluation of the strength of our approach through testing on a larger scale (i.e., more than 10 workers and 100GB input data).

From the infrastructure point of view, while HyIaaS layer is mainly focused on OpenStack cloud, we plan to adapt our solution to OCCI standards [105], in order to further improve the system interoperability with other cloud platforms.

3.6 ITERATIVE MAP REDUCE OVER THE HYBRID CLOUD

Another applicative scenario is represented by Iterative Map Reduce workloads. This specific class of data-intensive applications iteratively reuses the same map and reduce functions multiple times with slight alterations such as change of input and output files. Each iteration proceeds using a simple loop condition and can use the previous iteration's output as its input.

Iterative MapReduce is particularly suitable for hybrid cloud big data analytics. As the management of data volumes is the more influencing element of data-processing in a hybrid environment, having a method to treat the input data that is significantly different from the classical MapReduce scenario, the execution of iterative MapReduce on hybrid clouds worths to be treated separately.

For this class of applications, data locality can be leveraged over and over again once the input data was replicated off-premise. However, given the large initial overhead of the data movement, an efficient solution that facilitates data locality is non-trivial.

Furthermore, since the extra off-premise resources incur pay-as-you-go costs, it is crucial to estimate the performance gains in advance, in order to be able to decide whether it is

worthwhile to commit any extra resources at all, and, if so, how many of them in order to achieve the desired cost-performance trade-off. This work aims to address both these directions.

We propose a novel technique that minimizes data movement over the inter-cloud network and thus guarantees elevated levels of data locality, while preserving cross-cloud data replication. We achieve this is a completely transparent fashion, without invasive changes to the MapReduce framework or the underlying storage layer by adapting existing features to the hybrid setup (Chapter 3.6.2).

Furthermore, we propose a performance prediction methodology that combines analytical modeling with micro-benchmarking to estimate time-to-solution in a hybrid setup, including any data movement and computation (Chapter 3.6.3).

Finally, we evaluate our approach in a series of experiments that involve two representative real-life iterative MapReduce applications exhibiting a highly intensive map phase that processes large input datasets. Our experiments demonstrate both the ability to achieve strong scalability using our data movement technique, as well as small prediction errors (Chapter 3.6.4). The content of this chapter was published in [100].

3.6.1 Challenges of data locality in hybrid IaaS clouds

MapReduce applications typically exhibit a high degree of data parallelism: massive amounts of data are transformed in parallel fashion during the map phase, after which they are aggregated in a reduce phase. This approach puts a high burden on the storage layer: it needs to serve a large number of concurrent read requests corresponding to the input data of the map phase, as well as a large number of concurrent write requests corresponding to the output of the reduce phase.

In this context, using a conventional distributed file system that is decoupled from the MapReduce runtime is not enough to deal with such highly concurrent I/O access patterns: this would incur a massive amount of network traffic, overwhelming the networking infrastructure and offsetting the benefits of storing the data in a distributed fashion.

For this reason, a key design choice of MapReduce is the ability to take advantage of data locality: the storage layer is colocated with the MapReduce runtime on the same nodes

and is specifically designed to expose the location of the data blocks, effectively enabling the scheduler to bring the computation close to the data and avoid a majority of the storage-related network traffic. By replacing the nodes with VMs, a similar configuration that can efficiently exploit data locality can be obtained in an IaaS cloud as well.

However, in a hybrid cloud setup, there are two major challenges. First (as underlined in the previous chapters), the storage layer and all data is deployed initially only on-premise. Thus, when additional off-premise VMs are provisioned from the external cloud provider to boost the initial setup, they cannot benefit out-of-the-box from data locality and need to fetch/write their data to/from the on-premise VMs.

Second, the link between the on-premise infrastructure and the external cloud provider is typically of limited capacity. Thus, off-premise VMs that need to communicate with on-premise VMs create a network bottleneck much faster than the case when all VMs are located within the same cloud.

These two challenges are even more exacerbated in the context of iterative applications: in many cases, a majority of the input data needed for the first iteration will be needed for the subsequent iterations (such data is called the invariant). Thus, adopting a naive solution where the off-premise VMs read the input data from the on-premise VMs over and over again over a weak link is not feasible.

Furthermore, for the data that changes from iteration to iteration, off-premise VMs need to constantly write their output remotely, then read it back in the subsequent iteration, again over the weak link. Given these circumstances, exploring a better solution that improves the ability to take advantage of data locality in a hybrid setup is critical.

3.6.2 Asynchronous data rebalancing technique

This chapter describes our proposal to enable efficient execution of iterative MapReduce jobs in a hybrid IaaS cloud setup. It focuses on defining a strategy to address the technical challenges mentioned in the previous Chapter.

At first sight, the problem of avoiding remote data transfers over the weak link seems to be easily addressable by using a conventional caching solution: the invariant data and the newly written data can simply be stored locally on the off-premise VMs for faster subsequent access. However, adopting such a

caching strategy is non-trivial, because it needs to integrate well into the whole MapReduce framework.

More specifically, since the scheduling of tasks is deeply linked with the data locality, the MapReduce scheduler will prefer on-premise VMs over off-premise VMs, which leads to a scenario where the off-premise VMs are underutilized.

Furthermore, even if the scheduler would not exhibit such preference and would rather aim for load balancing, it is not enough to simply cache the data blocks off-premise and expose their location, because the storage layers fills other roles as well: replication support for resilience and high availability, load balancing of the data distribution, etc. Thus, in order to scale and properly take advantage of all these features, it is important to extend the storage layer beyond the on-premise VMs and re-balance the data blocks so that they are spread both over the on-premise VMs as well as the off-premise VMs.

Also important are other non-functional aspects: users prefer to use a standard MapReduce distribution (e.g. Hadoop) that was tested and tuned in their on-premise cloud, rather than switch to a dedicated solution specifically written for a hybrid setup. Furthermore, switching to a custom storage layer may not always be feasible: for example, if a huge amount of data is already stored in a regular on-premise MapReduce deployment, the overhead of migrating to a custom storage layer might offset the benefits of enabling the hybrid support altogether.

For these reasons, we propose a non-invasive solution that solves the aforementioned issues without deviating from the standard storage layer. Our key idea is to leverage rack awareness, a feature typically implemented in production-ready MapReduce storage layers, such as HDFS [114]. Originally intended as a mechanism to enhance fault tolerance, rack awareness enables the user to specify for each HDFS node that is part of the deployment a logical group, typically corresponding to a physical rack of the cluster where Hadoop is deployed. Using this information, HDFS replicates each data block at least once outside of the group where it was written, under the assumption that such a behavior improves the ability to resist catastrophic failures where a whole rack would fail at once.

In our context, we leverage rack awareness from a novel perspective. Specifically, we create two logical groups: one for the on-premise VMs and another for the off-premise VMs.

Thus, when new off-premise VMs are provisioned to boost the capability of the already running on-premise VMs, we extend the HDFS deployment in a rack-aware fashion on the off-premise VMs.

Using this approach, whenever an off-premise VM writes a new data block, it actually writes both local copies (solving the locality issue) as well as at least a remote copy, which enables efficient storage elasticity: off-premise VMs can be simply killed as desired without having to worry about transferring the data back to the on-premise side.

The only remaining issue is that the HDFS data nodes running on the off-premise VMs are initially empty, which prompts the need to re-balance the initial data blocks in order achieve load balancing and enable the scheduler to fully take advantage of the off-premise VMs.

However, re-balancing has its own overhead and as such is subject to a trade-off: at one extreme one can wait until all invariant data is balanced, which enables a maximum acceleration of the iterations from the beginning; at the other extreme one can run the re-balancing asynchronously, which eliminates the initial overhead at the cost of gradual acceleration of the iterations as the data balancing progresses.

We opted for the second option, since the initial overhead of re-balancing is significant and the ability to overlap the computation with the data transfers is crucial. While more elaborate balancing strategies (e.g. wait until a certain number of blocks was transferred off-premise then switch to the asynchronous strategy) are possible to explore, this is outside the scope of this work.

3.6.3 Performance prediction model

In some cases, using a hybrid cloud is a functional requirement: there are simply not enough resources on-premise to run the application with the desired level of complexity. However, most of the time, users are interested in a hybrid solution because they intend to accelerate their application by renting extra off-premise VMs. Since a hybrid solution incurs additional costs, it is important to understand how much the hybrid solution can accelerate the application, given a number of extra off-premise VMs. Ideally users would like to have the answer in advance, in order be able to decide apriori whether it is worthwhile to commit the extra off-premise VMs or not. In this

chapter, we propose a performance prediction methodology that addresses this issue.

Assumptions

We assume a MapReduce deployment that initially spans N on-premise VMs where all initial invariant data is distributed. These N on-premise VMs are complemented by M extra off-premise VMs, where we extend the MapReduce deployment using the asynchronous rack-aware rebalancing strategy mentioned in the previous chapter and then run the iterative application on the resulting hybrid setup. For simplicity, we assume the on-premise VMs are identical in capabilities to the off-premise VMs. Furthermore, we assume that the user has access to the historical traces of the application or can estimate important MapReduce metrics: total number of map/reduce tasks (p_M and p_R); total number of map/reduce slots (k_M and k_R); average map/reduce/shuffle duration (A_M, A_R, A_S), average data/shuffle sizes per map/reduce task (D_M, D_R, D_S). Also, we assume that the iterative applications exhibit a well-defined behavior: the number of iterations (I) is known in advance and the map/reduce tasks do not change in terms of number, amount of input data and computational complexity from one iteration to another.

Performance model for on-premise jobs

Using these metrics, techniques to estimate the runtime of MapReduce jobs on a single cluster have been proposed before and can be used in our case for the N on-premise VMs. In particular, Verma et. al. propose a model based on the make-span theorem [115], which states that for a greedy assignment of p tasks on k workers, the lower and upper bound for the execution time is $p \cdot A/k$ and, respectively, $(p-1) \cdot A/k + \lambda$, with A the average execution time of the tasks and λ the execution time of the slowest task. Intuitively, the lower bound corresponds to an ideal scenario where there is perfect load balancing, while the upper bound corresponds to a worst case scenario where the slowest task is scheduled last, after all other $p-1$ tasks finished in at most $(p-1) \cdot A/k$ time. Since MapReduce does not overlap the map phase with the reduce phase, both can be treated separately using the make-span theorem. For simplicity, we focus in this work on

the lower bound only. The upper bound can be estimated in a similar fashion.

$$T_M^{low} = A_M \cdot \frac{p_M}{k_M} \qquad T_R^{low} = A_R \cdot \frac{p_R}{k_R} \tag{3.10}$$

More complexity is introduced by the shuffle phase, for which the first wave overlaps with the map phase and thus the resulting overhead needs to be considered separately (denoted A_{S1}). For the rest of the shuffle waves ($p_R/k_R - 1$), the make-span theorem can be applied as usual:

$$T_S^{low} = A_S \cdot (\frac{p_R}{k_R} - 1) + A_{S1} \tag{3.11}$$

Thus, the estimated completion time for a single iteration is:

$$T^{low} = T_M^{low} + T_R^{low} + T_S^{low} \tag{3.12}$$

Considering all iterations, the total estimated completion time is:

$$T^{low} = \sum_{i=1}^{I} (T_{M_i}^{low} + T_{R_i}^{low} + T_{S_i}^{low}) \tag{3.13}$$

Performance model for hybrid jobs

In a hybrid setup, two important aspects affect the estimations discussed above: (1) while the asynchronous rebalancing progresses in the background, it generates extra overhead, which will slow down the map/shuffle/reduce (2) due to the weak link between the on-premise and off-premise VMs, the data transfer during the shuffle may experience a slowdown.

Due to the fact that (2) is highly complex and dependent on the nature of the application, we focus in this work on (1), leaving (2) as future work. Thus, we propose to amend the equations above such that they reflect the rebalancing aspect. Specifically, two important factors characterize this aspect. First, while the rebalancing is in progress, the mappers become slower due to the additional background activity. We denote this slowdown as α. It remains greater than 1 while the rebalance is in progress and equals 1 after the rebalance is complete. Second, as more data is transferred off-premise during the rebalancing, more locality can be exploited by

the scheduler, which effectively translates to more mappers that are scheduled off-premise. For simplicity, we adopt a simple heuristic to account for this effect that assumes only rack-local mappers will be executed the scheduler. This roughly corresponds to real life: only a negligible fraction of mappers are not scheduled rack-local. Furthermore, we make another simplifying assumption: all mappers are scheduled at the beginning of the iteration. Under these circumstances, the total number of parallel mappers during iteration i (denoted k_{M_i}) depends on the progress of the rebalancing at the beginning of the iteration, ranging from the map slots available on-premise only (k_{M_1}) to the map slots available both on-premise and off-premise after the rebalancing is complete.

For the rest of this chapter, we refer to α and k_{M_i} as the *hybrid rebalance factors*. Thus, for the hybrid case, the estimated completion time for the map phase is:

$$T_M^{low} = \sum_{i=1}^{I} \alpha \cdot A_M \cdot \frac{p_M}{k_{M_i}} \tag{3.14}$$

$$T_M^{up} = \sum_{i=1}^{I} \alpha \cdot A_M \cdot \frac{p_M - 1}{k_{M_i}} + \lambda \tag{3.15}$$

Methodology to leverage the hybrid performance model

In order to make use of the hybrid performance model introduced above for actual predictions, we need to estimate the hybrid rebalance factors. However, due to the complex inter-play between the system, the virtualization layer and the MapReduce framework that depends on a variety of parameters (i.e., point-to-point bandwidth between VMs, aggregated bandwidth of the weak link between on-premise and off-premise VMs, I/O pressure on the local storage, etc.), it is not easy to determine them analytically.

Thus, we propose to establish them experimentally, by using a series of micro-benchmarks that are executed on the hybrid setup independently of the application. More specifically, given N on-premise VMs and a desired number of M off-premise VMs, we create a similar setup as if running the actual application (i.e. same data size, number of mappers, etc.). However, instead of running the application, we run

an I/O intensive benchmark that approximates the application behavior for the duration of the re-balancing.

To obtain α, we simply divide the result of the hybrid benchmark by the baseline (i.e. same I/O intensive benchmark running on-premise only). To obtain k_{M_i} , we correlate the rebalance progress observed during the I/O benchmark with the moment when each iteration starts for the real application as follows:

$$k_{M_i} = min(k_M^{max}, B_M(T_{M_{i-1}})/D_M) \tag{3.16}$$

In the equation above, $B_M(t)$ denotes the amount of data transferred off-premise at moment t, k_M^{max} denotes the total number of mapper slots available from the $N + M$ VMs both on-premise and off-premise, while D_M denotes the data size processed by each mapper. By convention, $T_{M_0} = 0$.

Note that it is not necessary to run the I/O intensive micro-benchmarks for before running every real application: these results can be cached and reused later if the off-premise setup is unchanged (e.g. same type of VMs, same aggregated throughput between the off-premise and on-premise nodes). Since in many cases it is possible to use historical micro-benchmark results to calculate the hybrid rebalance factors, we differentiate the running of the I/O micro-benchmarks from the actual calculation of the factors, which we henceforth refer to as *micro-calibration*.

Once the micro-calibration is done, T^{low} and T^{up} can be estimated as described in Chapter 3.6.3. To find an optimal configuration, one can simply take a set of representative values for M and calculate T^{low} for each M. Armed with the knowledge of how long the execution time is likely to be for a variable M, it is easy to estimate whether a speed-up is possible in the first place, and, if so, how much extra cost would be necessary to achieve it. Furthermore, since we target iterative applications, we can estimate the completion times for an arbitrary number of iterations, which aids in choosing the right trade-off between cost and precision of results.

3.6.4 Evaluation

Experimental setup

The experiments for this work were performed on the Kinton testbed of the HPC&A group based at Universitat Jaume I.

It consists of 8 nodes, all of which are interconnected with 1 Gbps network links and split into two groups: four nodes feature an Intel Xeon X3430 CPU (4 Cores), HDD local storage of 500 GB, and 4 GB of RAM. These less powerful nodes (henceforth called thin) are used for management tasks. The other four nodes feature two Intel Xeon E5-2630v3 (2 x 8 Cores), HDD local storage of 1 TB, and 64 GB of RAM. These more powerful nodes (henceforth called fat) are used to host the VMs.

In order to get as close as possible to a real-life hybrid cloud, we configure two separate IaaS clouds based on OpenStack Icehouse and QEMU/KVM 0.12.1 as the hypervisor. One of the OpenStack deployments acts as the on-premise cloud, while the other one acts the off-premise cloud. A fully-featured OpenStack deployment requires two management nodes: one *controller node* that manages the compute nodes where the VMs are hosted and one *network node* that manages the cloud networking, which is managed separately due to the complexity of the networking technologies involved.

More specifically, in a typical configuration based on neutron (the standard OpenStack network management service), there are three conceptually separated communication domains: the management network (i.e., used for control messages and administrative traffic), the internal network (i.e., traffic between the VM instances using private IP addresses) and the external network (i.e., traffic between the VM instances and the outside of the cloud). In this configuration, the VM instances are configured to directly communicate with each other via the links of their compute node hosts. However, all communication with the outside of the cloud is routed through the network node, which is equipped with three NICs, each dedicated to a communication domain. Thus, in a real-life hybrid cloud setup that involves two OpenStack deployments, any communication between on-premise and off-premise VMs will pass through the network nodes, which become the weak link (i.e., total aggregated throughput between all on-premise and off-premise VMs is 1 Gbps).

For our experiments, we created a new VM flavor: i2.xlarge. This flavor features 4 VCPUs, HDD local storage of 100 GB and 16 GB of RAM. Thus, each compute node has the capacity to host 4 VMs simultaneously. Since two VMs that are co-located on the same compute node can communicate at much higher rate that two VMs that are hosted on different compute nodes, we

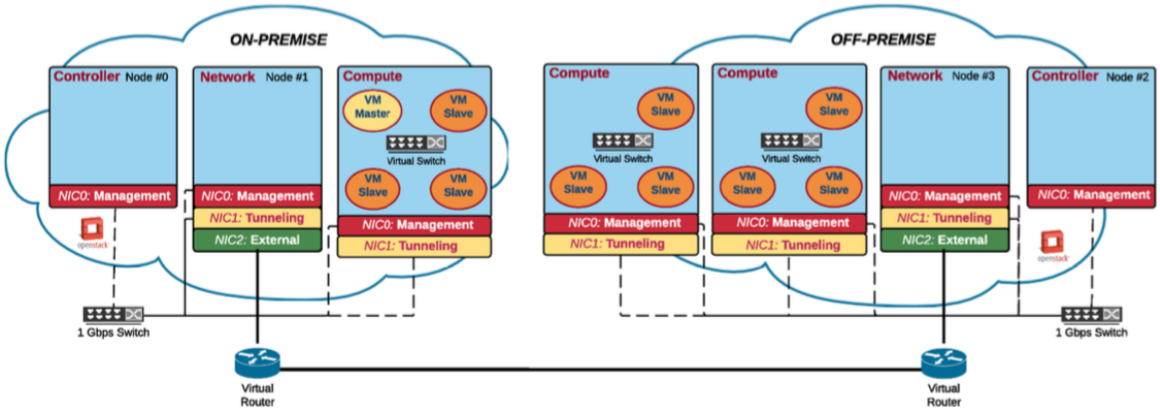


Figure 3.17: Hybrid IaaS OpenStack cloud example: one fat node on-premise and two fat nodes off-premise

limit the network capacity of this flavor to 1 Gbps to obtain a close-to-uniform environment where all VMs can communicate with each other at the same rate, regardless where they are located. This setup is illustrated in Figure 3.17, using one fat on-premise node and two fat off-premise nodes. On the on-premise part we provision 4 VMs in which Hadoop version 2.6.0 was deployed. One of these VMs is used as Hadoop master and the others as Hadoop slaves. Specifically, each Hadoop slave is configured both as a HDFS DataNode and as a YARN slave, with enough capacity to run 4 mappers and reducers simultaneously. On the off-premise part, we provision 4 VMs on each fat node, with a variable number of fat nodes ranging from one to three. In order to extend the Hadoop deployment over the off-premise VMs, we start the relevant services (i.e., HDFS DataNodes and YARN runtime) on the off-premise VMs. These services will report to the master, which integrates them into the Hadoop deployment. Rack-awareness is achieved by creating two groups corresponding to the on-premise an off-premise VMs and assigning each HDFS DataNode to the appropriate group.

Overview

We run extensive experiments with two real-life MapReduce iterative applications, described in greater detail in Chapter 3.6.2. Both applications exhibit a reduction phase that involves a negligible amount of data compared with the map phase, which is a frequent real-life scenario that emphasizes the map phase. The goal of these experiments is two-fold: (1) to

demonstrate the feasibility of our re-balancing proposal; (2) to validate the hybrid performance prediction model introduced in Chapter 3.6.3. against the results observed in real life.

First, we run a series of I/O intensive benchmarks that correspond to the micro-calibration mentioned in Chapter 3.6.3. To this end, we rely on the *TestDFSIO* microbenchmark, which is a standard Hadoop tool that measures the HDFS read and write throughput under concurrency.

Second, we run a series of experiments that study the strong scalability of the application on a single OpenStack cloud. Since there is no weak link in this setup, these experiments reveal the maximum theoretical potential for speed-up in a hybrid setup. We refer to this series of experiments as *Baseline*. Then, we run the same experiments in a hybrid setup, where we fix the number of on-premise VMs and vary the number of off-premise VMs. We refer to these experiments as *Hybrid-real*. We discuss these results in comparison with Baseline to address goal (1). Forth, based on the results from *Baseline* experiments and the *micro-calibration*, we extract the relevant application metrics and compute the hybrid rebalancing factors to estimate T_M^{low} and T_M^{up} using the equations described in Chapter 3.6.3. Since the reduce phase is negligible compared with the map phase, $T^{low} \approx T_M^{low}$ and $T^{up} \approx T_M^{up}$. We then discuss these results in relationship with Hybrid-real to address goal (2).

Micro-calibration

In this chapter we illustrate how to perform the micro-calibration. To demonstrate how to reuse the results of the micro-benchmarking for multiple applications, we fix the application input data at 20 GB and the HDFS chunk size (corresponding to the size of data per mapper D_M) at 64 MB, which means a total of 300 map tasks (p_M) are needed.

First, the data is generated by running TestDFSIO in write mode in an HDFS deployment spanning 3 on-premise VM instances. After the initial data was written, the HDFS deployment is extended by a variable number of additional off-premise VM instances. Then, the hybrid rebalancing is started at the same time with another TestDFSIO that runs this time in read mode. While the experiment is running, we monitor the amount of data that accumulates off-premise during the rebalancing. We record both this progress and the metrics reported by TestDFSIO, which is run repeatedly in an iterative fashion until the rebalancing is complete.

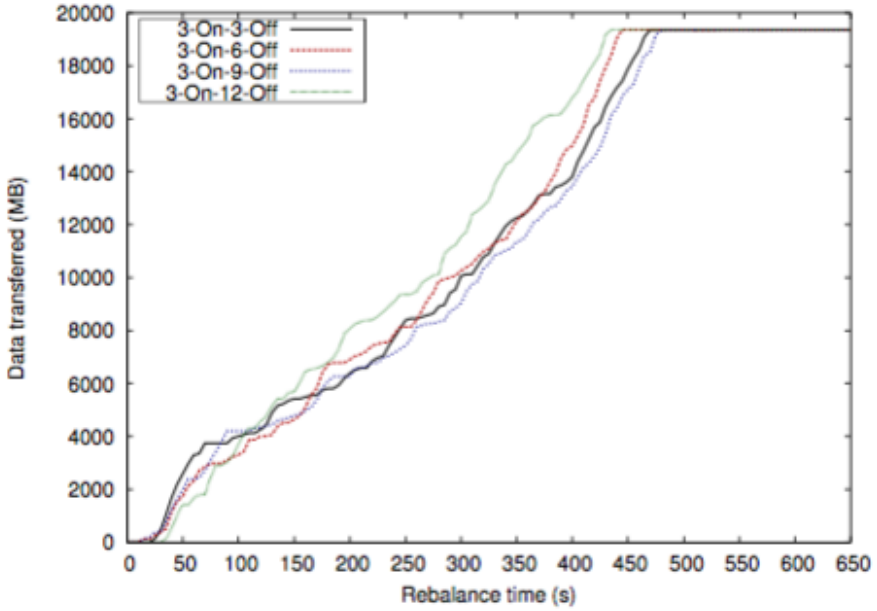


Figure 3.18: TestDFIO micro-calibration: rebalance progress for 20 GB total data.

The results of the rebalancing progress are depicted in Figure 3.18. As can be observed, the off-premise HDFS data accumulates steadily in all configurations. Furthermore, there is little difference between the various hybrid configurations, which enables an estimation of $B_M(t)$ (introduced in Chapter 3.6.4) even when micro-benchmarking results are not available for a particular configuration.

The average completion time per concurrent read iteration for TestDFSIO is illustrated in Table 3.6. By convention, we denote a configuration with N on-premise VMs and M off-premise VMs as N-on-M-off. 3-on-0-off is the baseline for which no re-balancing is present. Both the baseline and the hybrid TestDFSIO experiments are repeated 10 times. These results are then used to calculate α, included in the table. As can be observed, the re-balancing introduces significant background overhead that reduces the concurrent read throughput and lowers the overall completion time per iteration by up to 75%. Also, interesting to observe is that α remains very close for all hybrid configuration except 3-on-12-off. Thus, the previous observation about a rough estimation being possible even when no micro-benchmarks are available for a particular configuration holds for α too.

Configuration	Time / iteration	α
3-on-0-off	276s	N/A
3-on-3-off	472s	1.70
3-on-6-off	471s	1.70
3-on-9-off	485s	1.75
3-on-12-off	416s	1.5

Table 3.6: TestDFSIO Agerage completion time per iteration

KMeans

Our next series of experiments focus on K-Means [116], a widely used application in a multitude of contexts: vector quantization in signal processing, cluster analysis in data mining, pattern classification and feature extraction for machine learning, etc. K-Means partitions a set of multidimensional vectors into k sets, such that the sum of squares of distances between all vectors from the same set and their mean is minimized. This is typically done by using iterative refinement: at each step the new means are calculated based on the results from the previous iteration, until they remain unchanged (with respect to a small epsilon). K-Means was shown to be efficiently parallelizable and scales well using MapReduce [117], which makes it a popular tool to analyze large quantities of data at large scale.

For the purpose of this work, we use the MapReduce K-Means implementation that is part of the Mahout 0.10 collection of machine learning algorithms. This implementation generates only a minimal amount of intermediate data at each iteration (i.e., the mean for each of the k sets), however it typically analyses a large amount of input data that remains unchanged between the iterations. Thus, it is classified as a map-intensive job. We generate 20 GB worth of input that is processed in 10 iterations. The data is generated using the data generator included in Mahout and is uploaded to HDFS before starting each experiment. For comparison, the shuffle data for each iteration is in the order of several MB, which is why we can consider the reduction phase negligible (i.e., $T^{low} \approx T_M^{low}$ and $T^{up} \approx T_M^{up}$).

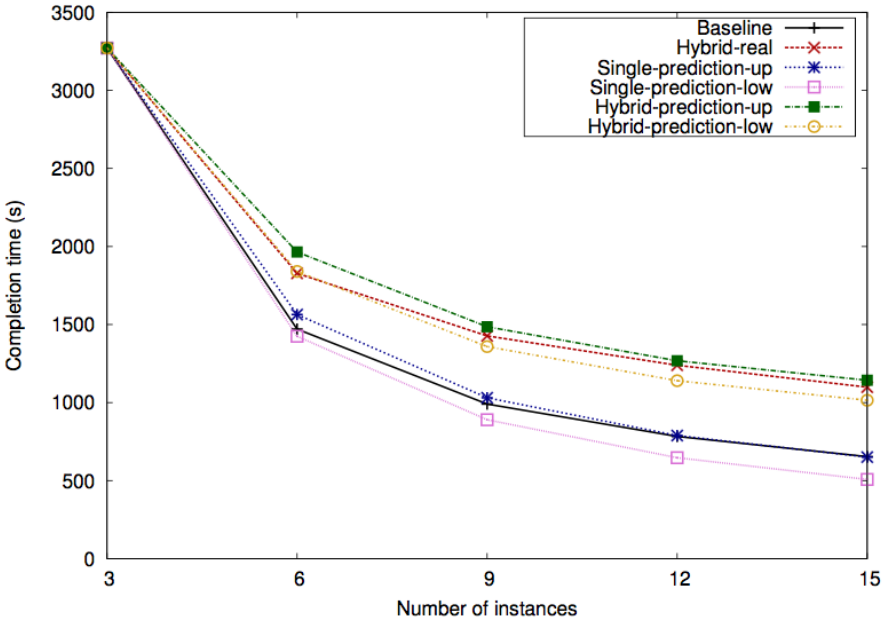


Figure 3.19: K-Means of a 20GB dataset. Strong scalability: predicted vs. real total completion time for 10 iterations for a single cloud and a hybrid cloud setup. The measured completion time observed on the single cloud is the Baseline. Lower is better.

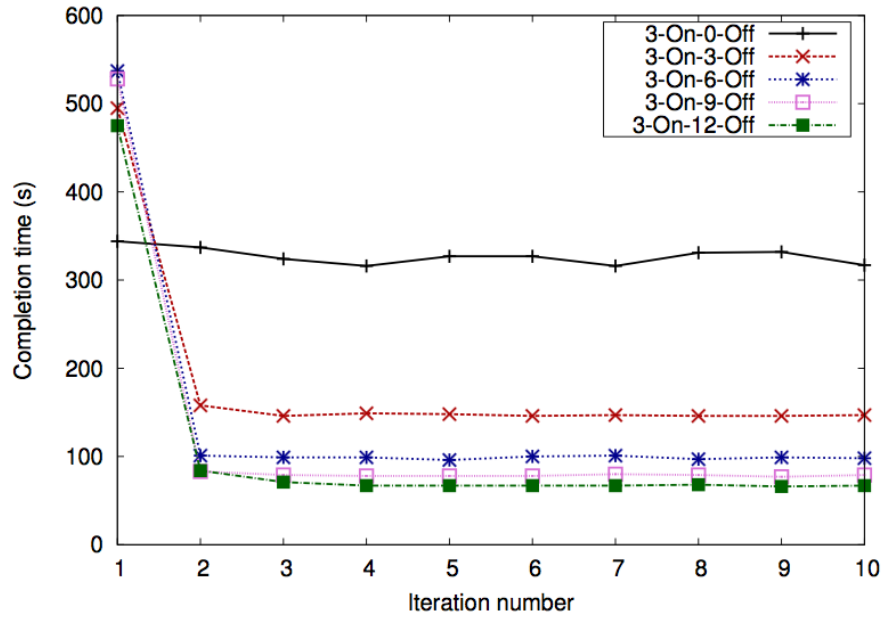


Figure 3.20: K-Means of a 20GB dataset. Iteration analysis: completion time per iteration for an increasing number of off-premise VM instances. Lower is better.

First, we run the Baseline experiment by deploying a single OpenStack cloud where we vary the number of VMs allocated to the Hadoop deployment. As can be observed in Figure 3.19, with an increasing size of the Hadoop deployment, K-Means experiences a steady drop in the total completion time, confirming its potential to achieve strong scalability. Furthermore, applying the performance model for on-premise jobs introduced in Chapter 3.6.3 reveals a good estimation of the total completion time: the Baseline stays within the lower (Single-prediction-low) and upper (Single-prediction-up prediction bounds at all times. Furthermore, there is almost a perfect overlap between Single-prediction-up and Baseline, while Single-prediction-low provides an over-optimistic estimation that deviates by at most 20%.

Next, we run the Hybrid-real experiment, where we deploy a hybrid setup consisting of 3 on-premise VMs and a variable number of off-premise VMs (X axis depicts total number of VMs). Initially, Hadoop is deployed only on the on-premise VMs and is extended as described in the Chapter 3.6.4, with the asynchronous rebalancing and the application being started simultaneously. The total completion time can be observed in Figure 3.19. Interesting to note is the drop in completion time with increasing number of off-premise VMs. As expected, the rebalancing overhead in the hybrid case has a negative impact on the strong scalability when compared with Baseline (up to of 40% increase in execution time), however the scalability trend is clearly visible, confirming the viability of adopting our proposal to extend iterative MapReduce jobs using additional VMs leased from an off-premise cloud. Furthermore, by using the relevant application metrics extracted from the Baseline experiments (i.e., A_M, λ) and the micro-calibration results from Chapter 3.6.4 in the equations described in Chapter 3.6.3, we obtain the lower (Hybrid-prediction-low) and upper (Hybrid-prediction-up) total estimated hybrid completion time. As can be observed, we can see again a good prediction: the real result stays within the lower and upper bound, while the error is at most 8% for the lower bound and 4% for the upper bound.

To understand these results better, we zoom in Figure 3.20 on the completion time per iteration. We use the same N-on-M-off notation for each configuration as explained in Chapter 3.6.4. As expected, for the 3-on-0-off case, the completion time per

iteration remains constant. However, in the 3-on-M-off cases, a large gap between the first and the rest of the iterations is visible. This is explained by the fact that the re-balancing finishes during the first iteration, such that beginning with the second iteration, the data locality can be fully exploited. Since the invariant input data is reused at each iteration, most of the increase in the total completion time is due to the first iteration. Thus, if more than 10 iterations are needed, this initial overhead will be better amortized.

Iterative Grep

The second application we evaluate is iterative grep, which is a popular analytics tool for large unstructured text. Iterative grep consists of a set of independent grep jobs that find all string matches of a given regular expression and sorts them according to the number of matches. The iterative nature is exhibited in the fact that the input data remains the same, but the regular expression changes as a refinement of the previous iteration. For example, one may want to count how many times a certain concept is present in the Wikipedia articles, and, depending on the result, prepare the next regular expression in order to find correlations with another concept. Since the regular expression is typically an exact pattern, the output of the mappers is very simple and consists of a small number of key-value pairs that are reduced to a single key-value pair. Thus, it can be classified as a typical map-intensive MapReduce job.

For the purpose of this work, we use the standard grep implementation that comes with the Hadoop distribution. We use 20 GB worth of Wikipedia articles as input data and 10 keywords to run 10 iterations over this input data, which is uploaded to HDFS before each experiment. The shuffle data for each iteration is less than one MB, which is why we can consider the reduction phase negligible (i.e., $T^{low} \approx T_M^{low}$ and $T^{up} \approx T_M^{up}$).

As can be observed in Figure 3.21, for the Baseline experiment (measured total completion time for a single cloud), there is again evidence of strong scalability. This is understandable, since grep is almost embarrassingly parallel. However, there is a slight degradation of scalability for an increasing number of VMs, due to the increasing overhead of parallelization. Applying the performance model for on-premise jobs (Chapter 3.6.4), we observe the following

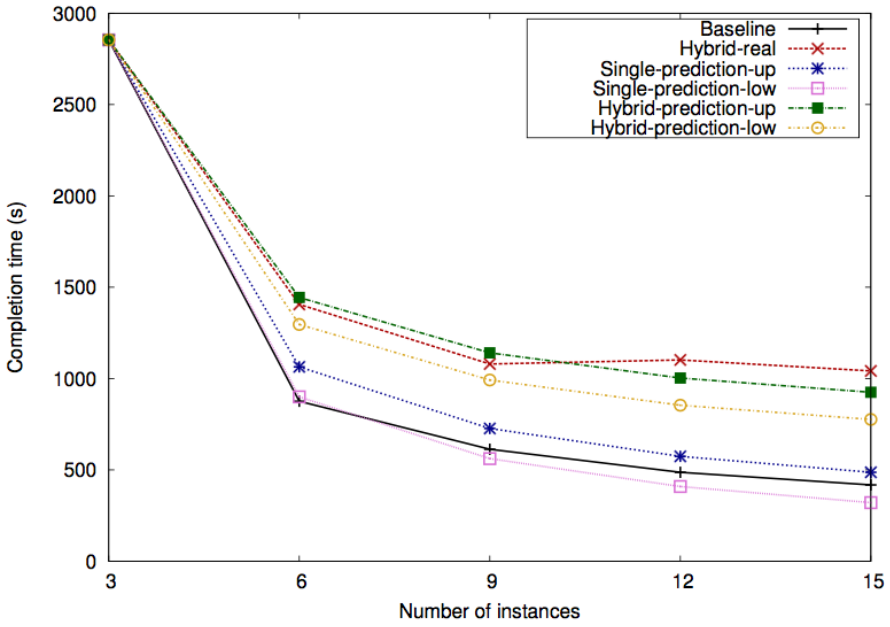


Figure 3.21: IGrep of a 20GB dataset. Strong scalability: predicted vs. real total completion time for 10 iterations for a single cloud and a hybrid cloud setup. The measured completion time observed on the single cloud is the Baseline. Lower is better.

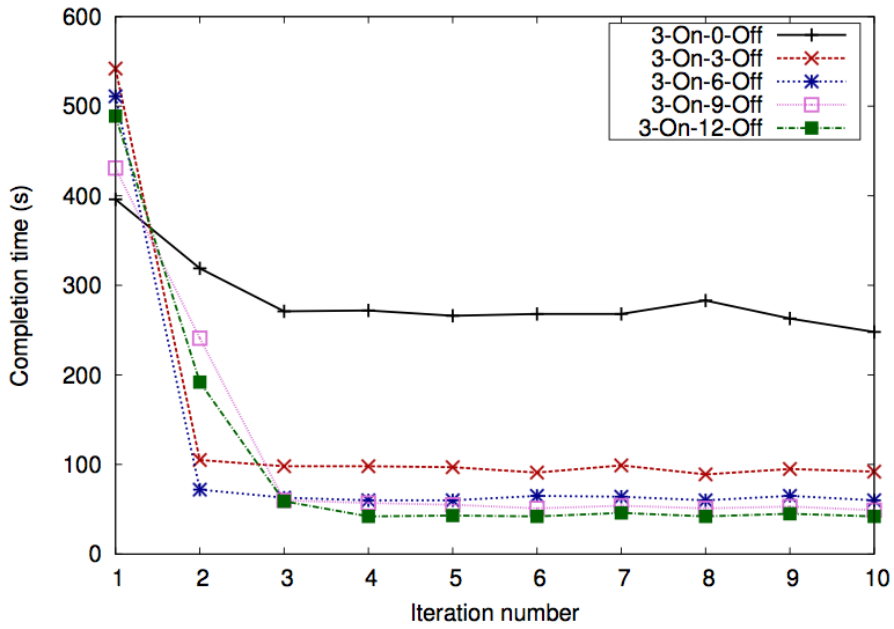


Figure 3.22: IGrep of a 20GB dataset. Iteration analysis: completion time per iteration for an increasing number of off-premise VM instances. Lower is better.

estimations for the total completion time: the lower bound (Single-prediction-low) under-estimates by up to 24% and the upper bound over-estimates by up to 15%, which places the measured total completion time within the lower and upper bound.

For the Hybrid-real experiment, we deploy a hybrid setup that keeps 3 on-premise VMs and adds a variable number of off-premise VMs. The total completion time can be observed in Figure 3.21 (total number of VMs on X axis). Again, we observe a drop in completion time with increasing number of off-premise VMs, which confirms the viability of adopting our re-balancing proposal. Furthermore, the lower (Hybrid-prediction-low) and upper (Hybrid-prediction-up) estimated hybrid completion time keep the measured result within their limits up until 6 off-premise VMs. However, when increasing the number of off-premise VMs beyond 6, both the lower and upper bound under-estimate the measured completion time: by up to 25% and 12% respectively.

These results are better understood by analyzing the per iteration completion times, which are depicted in Figure 3.22. Surprisingly, for the 3-on-0-off case, the completion time per iteration remains constant only after a few iterations, which hints at possible OS caching effects for the input data read from HDFS at each iteration. In the hybrid configurations, it can be observed that the rebalancing does not finish during the first iteration for the case when a large number of off-premise VMs is used (i.e. 9 and 12). This increased complexity may explain why the hybrid estimations exhibit larger errors than in the case of K-Means. In fact, when compared with the single cloud estimations, it can be observed that the errors are similar in magnitude, which hints that the hybrid aspect was accurately factored into the estimation.

3.6.5 Discussion

This work contributed to the autonomic management of VMs over hybrid clouds with a novel proposal that addresses this challenge for iterative MapReduce applications. It transparently manages data movements asynchronously in an efficient fashion without invasive changes to the MapReduce framework or the underlying storage layer. At the same time, it is able to predict the runtime of the application for a variety of hybrid configurations, by combining analytical modeling

with micro-calibration. The results using two real-life iterative MapReduce applications show excellent hybrid scalability potential that follows a similar trend as the single-site scalability except for an initial overhead during the first few iterations, whose impact on the overall execution time is diminished with increasing number of iterations.

Furthermore, our prediction of the execution time for a hybrid setup matches the accuracy of the techniques used in single-site setups, with maximum upper/lower bound errors of 4%/8% and, respectively, 12%/25%.

Encouraged by these results, we plan to broaden the scope of our work in future efforts. In particular, we focused on map-intensive applications where the reduce phase is negligible in comparison. Thus, one interesting direction is to complement the current work with an analysis of reduce-intensive jobs in a hybrid setup: study of the weak link and interferences with the rebalancing, refined prediction equations, etc.

3.7 CONCLUSIONS AND FUTURE WORK

In this chapter we highlighted the numerous kinds of cloud interoperability scenarios and we particularly focused on the hybrid environment as a promising mechanism to leverage both on-premise private and off-premise public clouds.

Hybrid cloud opens an entire new horizon in the big data analytics landscape, effectively enabling on-premise resource owners to extend complex workloads beyond the capacity of their infrastructure by leasing off-premise resources. However, the need to transfer large data sizes from the IC toward off-premise through a higher latency medium poses a difficult challenge to the ability to exploit data locality efficiently.

In this scenario, the infrastructure management must be combined with a sapient application-level strategy to transfer off-premise the minimum amount of data that enable a consistent off-premise computation without saturating the limited inter-cloud bandwidth.

This work particularly focused on MapReduce applications and contributed to the autonomic hybrid infrastructure management in this field by proposing a layered framework architecture to enable the dynamic provisioning and

configuration of off-premise VMs that extend the capacity of the private cloud.

We also proposed two different policies (SPAN and HyMR) to enable the execution of a classical MapReduce job when on-premise physical machines are in a stressed condition and a solution to boost iterative MapReduce leveraging the off-premise resources.

The proposed layered architecture needs to be extended and adapted to OCCI standards [105], in order to further improve the system interoperability with other cloud platforms.

As the map phase generally has a simpler and more standardizable execution schema when compared to the reduce phase – thanks to the possibility to leverage more the data locality –, several works in this field focus on the analysis and performance prediction of the mappers. Nevertheless, the state of the art would benefit from studying of the reduce phase in a hybrid setup. Therefore, the prediction model in Chapter 3.6 needs to be extended/refined to evaluate the performance of reduce-intensive jobs when some of the data are transferred through a limited bandwidth.

Finally, considering a classic MapReduce computation over hybrid cloud, the management policies depicted in Chapters 3.4 and 3.5 would benefit from the definition of a more complex strategy to anticipate the identification of a stressing condition going to occur on the physical nodes. For this reason, in the next chapter, we overlook the information about the current utilization level of physical machine resources (coming for the monitoring component of our management system) and focus on what can be inferred by analyzing the runtime behavior of the application.

4 MONITORING THE ARCHITECTURE WITH PROCESS MINING

DISTRIBUTED architectures, such as MapReduce, are providing technical answers to the challenge of big data analytics. Especially when coupled with a cloud infrastructure, these solutions can be enriched of additional features, such as the dynamic provisioning of computational nodes – e.g., to ensure that processing tasks are performed within a temporal deadline or to deal with data peaks. As deepen in the previous chapters, good results can be achieved at infrastructure level by monitoring the resource utilization and spawning new VMs – eventually on an off-premise public cloud – to boost the execution.

Nevertheless, the factors that can slow down a MapReduce computation are not only related to the limited amount of resources available but they can also be connected with the specificities of the infrastructure configuration or processed data itself. For example, the time to execute a MapReduce job on the same volume of data can vary depending on the percentage of data local computation obtainable, the quality of the link between the computing nodes, the content of the processed data itself, etc.

For this reason, the autonomic management of a (hybrid) cloud infrastructure can also benefit from the implementation of advanced monitoring systems taking into account not only the physical resource utilization, but also domain specific information derived by the application level computation.

The applicative software running over the cloud is an example of system that must comply to a set of contract terms and functional and non-functional requirements specified by a SLA. The complexity of the resulting overall system, as well as the dynamism and flexibility of the involved processes, often require an on-line operational support checking compliance. Such monitor should detect when the overall system deviates from the expected behavior, and raise an alert notification immediately, possibly suggesting/executing specific recovery actions. This run-time monitoring/verification aspect – i.e. the capability of determining during the execution if the system

exhibits some particular behavior, possibly compliance with the process model we have in mind – is still matter of an intense research effort in the emergent Process Mining [118] area. As pointed out by Van Der Aalst et al. in [118], applying Process Mining techniques in such an online setting creates additional challenges in terms of computing power and data quality.

Starting point for Process Mining is an event log. We assume that in the architecture going to be analyzed it is possible to sequentially record events. Each event refers to an activity (i.e., a well-defined step in some process/task) and it is related to a particular process instance. Note that, in case of a distributed computation, we also need extra information such as, for instance, the resource/node executing, initiating and finishing the process/task, the timestamp of the event, or other data elements.

While, in an cloud architecture, several tools exists for performing a generic, low-level monitoring task [110, 119], we advocate also the use of an application/process oriented monitoring tool in the context of Process Mining in order to run-time check the conformance of the overall system. Essentially, the goal of the work presented in this chapter is to apply the well-known Process Mining techniques to the monitoring of complex distributed applications, such as MapReduce in a cloud environment.

As suggested by the previous chapters, if we assume that the performance of the overall computing infrastructure is stable and a minimum QoS is guaranteed, MapReduce parallelization model makes relatively simple to estimate a job execution time by on-line checking the execution time of each task in which the application has been split. This estimation can be compared to the deadline and used to predict the need for scaling the architecture [79, 88, 100].

Nevertheless, the initial assumptions may be not always satisfied and the execution time may differ from what expected depending on either architectural factors (e.g., the variability in the performance of the machines involved in the computation or the fluctuation of the bandwidth between the nodes), or domain-specific factors (e.g., a task is slowed down due to the input data content or location). This unpredictable behavior could be run-time corrected if the execution relays on an elastic set of computational resources as that provided by cloud computing systems.

In this chapter, we focus on the step of monitoring MapReduce applications, to detect situations where additional resources are needed to meet the deadlines. To this end, we exploit some techniques and tools developed in the research field of Business Process Management [120]: in particular, we focus on declarative languages and tools for monitoring the execution of business processes. Inside SHYAM framework, we introduce a logic-based monitor able to detect possible delays, and trigger recovery actions such as the dynamic provisioning of further resources.

Since MapReduce applications typically operate in dynamic, complex and interconnected environments demanding high flexibility, a detailed and complete description of their behavior seems to be very difficult, while the elicitation of the (minimal) set of behavioral constraints/properties that must be respected to correctly execute the process, and that cannot be directly incorporated at design time into the system can be more realistic and useful.

Therefore, in this context, we will adopt the Event Calculus [121], logic-based formalism for representing actions and their effects, in particular we relay on a verification framework based on constraints, called MOBUCON EC (Monitoring business constraints with Event Calculus [122]), able to dynamically monitor streams of events characterizing the process executions (i.e., running cases) and check whether the constraints of interest are currently satisfied or not. MOBUCON is an extension of the constraint-based Declare language [123] and is data aware. It allows us to specify properties of the system to be monitored using a logic-based syntax involving time constraints and task data. The Event Calculus formalization has been proven a successful choice for dealing with runtime verification and monitoring, thanks to its high expressiveness and the existence of reactive, incremental reasoners [122].

This chapter presents an on-line monitoring system to check the compliance of each node of a distributed infrastructure for data processing running on a cloud environment. The resulting information is used for taking scaling decisions and dynamically recovering from critical situations with a best effort approach by means of an underlying previously implemented infrastructure layer. This could be considered as a first step towards a MapReduce engine with autonomic features either in run-time detecting undesired task behaviors,

or in handling such events with dynamic provisioning of computational resources in an hybrid cloud scenario.

Chapter 4.1 provides a first introduction to the Event Calculus formalization and MOBUCON framework. Later, in Chapter 4.2, we describe the implemented architecture in detail, while in Chapter 4.3, we focus on the data-intensive use case scenario and we provide a first evaluation of the performance of our contribution. Chapter 4.4 summarize the novelties introduced by our approach and suggests future related work. The content of this chapter is also presented in the work [124].

4.1 POSITIONING OUR CONTRIBUTION

As regards the use of Event Calculus for verification and monitoring, several examples can be found in litterature in different application domains but we are not aware of any work applying it to the monitoring of MapReduce jobs in a cloud environment. Event Calculus has been used in various fields to verify the compliance of a system to user-defined behavioral properties. For example, [125], [126] exploit ad-hoc event processing algorithms to manipulate events and fluents, written in JAVA. Differently from MOBUCON , they do not have an underlying formal basis, and they cannot take advantage of the expressiveness and computational power of logic programming.

Several authors – [127], [128] – have investigated the use of temporal logics – LINEAR TEMPORAL LOGIC (LTL) in particular – as a declarative language for specifying properties to be verified at runtime. Nevertheless, these approaches lack the support of quantitative time constraints, non-atomic activities with identifier-based correlation, and data-aware conditions. These characteristics – supported by MOBUCON – are instead very important in our application domain.

Our approach takes inspiration from the work by Mattess et al. [88], which presents an online provisioning policy to meet a deadline for the Map phase. In order to check the compliance of the application, [88] takes into account the execution time of each map tasks but, differently from our approach, the deadline compliance prediction for the Map phase is computed with a traditional approach to monitoring, which introduces complexity in the implementation and tuning, whereas our

solution can benefit from a simple enunciation of the system properties relaying on the declarative approach offered by MOBUCON .

4.2 MAPREDUCE AUTO-SCALING ENGINE

We relay on HYLAAS system presented in Chapter 3.3 to develop MapReduce Auto-scaling Engine, an application-level software component to online detect user-defined critical situations in a MapReduce environment. The resulting framework architecture autonomously react by providing or removing resources according to high-level rules definable in declarative language.

consists of three main subcomponents (grey blocks in Figure 4.1): the Monitoring, Recovery and Platform Interface. These elements interacts with the MapReduce platform to detect and react to anomalous sequences of events in the execution flow.

The Monitoring component takes as input a high-level specification of the system properties describing the expected behavior of a MapReduce workflow and the on-line sequence of events from the MapReduce platform's log. Given these input data, the Monitoring component is able to rise alerts whenever the execution flow violates user-defined constraints. The alarms are evaluated by the Recovery component in order to estimate how many computational nodes must be provisioned (or de-provisioned) to face the critical condition according to user-defined rules taken as input.

Finally, the Platform Interface is in charge of translating the requests for new MapReduce nodes into VM provisioning requests to the infrastructure manager. The Platform Interface is also responsible for the installation of MapReduce-specific software on the newly provided VMs. The output of this subcomponent is a new configuration of the computing cluster with a different number of working nodes.

As shown in Figure 4.1, MapReduce Auto-scaling Engine relays HYLAAS component presented in Chapter 3.3 for the provisioning of VMs [113]. This layer encapsulates the cloud functionality and interacts with different infrastructures to realize a hybrid cloud: if the resources of the company-owned cloud are no longer enough, HYLAAS redirects the scale-up request to an off-premise public cloud. Therefore, thanks to

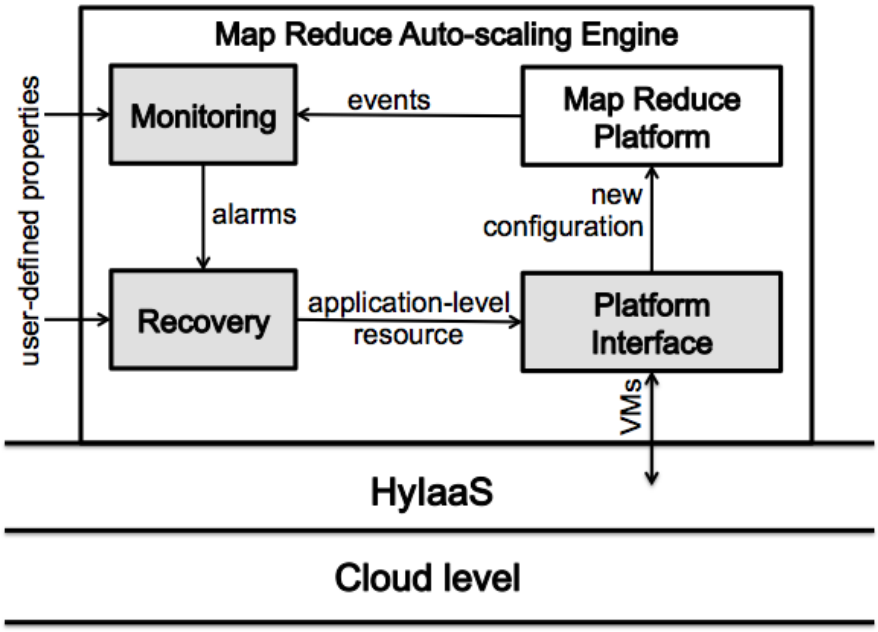


Figure 4.1: Framework architecture of MapReduce Auto-scaling Engine

HylaaS, the resulting cluster for MapReduce computation can be composed by VMs physically deployed on different clouds.

The hybrid nature of the resulting cluster is often very useful (especially if the on-premise cloud has limited capacity) but can also further exacerbate the problem of MapReduce performance prediction. If part of the computing nodes is available through a higher latency, the execution time can be substantially afflicted by the allocation of the tasks and the amount of information they trade with each other. Despite the complexity of the scenario, we want the monitoring system to offer a simple interface for the elicitation of the properties to be respected. Nonetheless, it should be able to rapidly identify critical situations. To this end, we apply the Mobucon framework to the monitoring component and benefit from the application of well-known Process Mining techniques to our environment.

4.2.1 Monitoring the system execution w.r.t. declarative constraints

Monitoring complex processes such as MapReduce approaches in dynamic and hybrid clouds has two fundamental requirements: on one hand, there is the need of a language

expressive enough to capture the complexity of the process and to express the key properties that should be monitored. Of course, for practical applications, such language should come already equipped with sound algorithms and reasoning tools. On the other hand, any monitor must produce results in a timely fashion, being the analysis carried out on the fly, typically during the system execution.

Declarative languages are one of the solutions proposed in the field of Business Process Management to answer the above requirements. In particular, they have been adopted to model business rules and loosely-structured processes, mediating between support and flexibility.

Among the many proposals, we focused on the Declare language [123], a graphical, declarative language for the specification of activities and constraints. The Declare language has been extended with temporal deadlines and data-aware constructs in [122, 129], where also the MOBUCON tool has been presented, together with some figures about its performances in a run-time context.

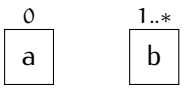
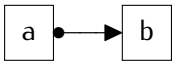
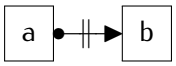
Declare is a graphical language focused on *activities* (representing atomic units of work), and *constraints*, which model expectations about the (non) execution of activities. Constraints range from classical sequence patterns to loose relations, prohibitions and cardinality constraints. They are grouped into four families:

- *existence* constraints, used to constrain the number of times an activity must/can be executed;
- *choice* constraints, requiring the execution of some activities selecting them among a set of available alternatives;
- *relation* constraints, expecting the execution of some activity when some other activity has been executed;
- *negation* constraints, forbidding the execution of some activity when some other activity has been executed.

Table 4.1 shows few simple Declare constraints.

The Declare language provides a number of advantages: being inherently declarative and open, it supports the modeler in the elicitation of the (minimal) set of behavioral *constraints* that must be respected by the process execution. Acceptable execution courses are not explicitly enumerated, but rather, they are implicitly defined by the execution traces that comply with all the constraints. In this sense, Declare is indeed a

Table 4.1: Some Declare constraints

	ABSENCE The target activity a cannot be executed EXISTENCE Activity b must be executed at least once
	RESPONSE Every time the source activity a is executed, the target activity b must be executed after a
	PRECEDENCE Every time the source activity b is executed, a must have been executed before
	NEGATION RESPONSE Every time the source activity a is executed, b cannot be executed afterwards

notable example of *flexibility by design*. Moreover, Declare (and its extensions) supports temporal deadlines and data-aware constraints, thus making it a powerful modeling tool. The MOBUCON tool fully supports the Declare language; moreover, being based on a Java implementation of the Event Calculus formalism [130], it provides a further level of adaptability: the system modeler can directly exploit the Event Calculus – as in [131] – or the Java layer underneath for a fully customizable monitoring. Finally, MOBUCON and the extended Declare support both atomic and non-atomic activities.

4.3 USE CASE SCENARIO

The architecture shown in Figure 4.1 has been implemented and analyzed using a testbed framework. In particular, a simulation approach has been adopted to create specific situations, and to verify the run-time behavior of the whole architecture. To this end, synthetic data has been generated, with the aim of stressing the MapReduce implementation.

4.3.1 Testbed architecture and data

As in the works presented in Chapter 3.3, we opted for Apache Hadoop [59] framework for MapReduce.

Our Hadoop testbed is composed of 4 VMs: 1 master and 3 worker nodes. Each VM has 2 VCPUs, 4GB RAM and 20GB disk. At the cloud level we use 5 physical machines, each one with a Intel Core Duo CPU (3.06 GHz), 4GB RAM and 225GB HDD.

We allocated a VM on each compute node. Since a dual core machine (without hyperthreading) can concurrently execute at most two tasks, we assigned two map slot (and two reduce slots) to each worker. Our MapReduce platform can therefore execute up to six concurrent tasks. See Chapter 3.2.2 for further details about Hadoop architecture.

As for the task type, we opted for a word count job, often used as a benchmark for assessing performances of a MapReduce implementation. In our scenario, we prepared a collection of 20 input files of 5MB each. Consequently, Hadoop MapReduce Runtime launches 20 mappers to analyze the input data. In this testbed we would like to complete the map phase in 20 minutes, so every map task should not exceed one minute execution.

According to the default Hadoop configuration, the output of all these mappers is analyzed by a single reducer.

In order to emulate the critical condition of some tasks showing an anomalous behavior, we artificially modified 8 input files, so has to simulate a dramatic increase of the time required to complete the task. The mappers analyzing these blocks resulted to be 6 times slower than the normal ones.

4.3.2 Properties to be monitored

In this work we mainly focus on time-constrained data insight: the aim is to identify as soon as possible the critical situation of the MapReduce execution going to complete after a predefined deadline. Practically speaking, this correspond to situations where the total execution time of the MapReduce is expected to stay within some (business-related) deadline: e.g., banks and financial bodies require to perform analyses of financial transactions during night hours, and to provide outcomes at the next work shift.

The MOBUCON framework already provides a model of activities execution, where a number of properties to be monitored are already directly supported. In particular, a support for non-atomic activities execution is proposed within the MOBUCON framework, where for each start of execution of a specific ID, a subsequent end of execution (with same ID) is expected. This feature has been particularly useful during the verification of our testbed, to identify a number of exceptions and worker faults due to problems and issues not directly related to the MapReduce approach. For example,

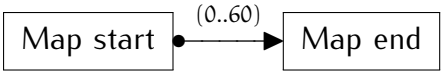


Figure 4.2: Declare Response constraint, with a metric temporal deadline

during our experiments we ignored fault events generated by power shortages of some of the PC composing the cloud. The out-of-the-box support offered by MOBUCON was exploited to identify these situations and rule them out.

To detect problematic mappers, we decided to monitor a very simple Declare property between the start and the end of the elaboration of each mapper. Declare augmented with metric temporal deadlines as in [122] was exploited to this end, and the constraint shown in Figure 4.2 illustrates the *Response* constraint we specified in MOBUCON. It simply states that after an event *Map start*, a corresponding event *Map end* should be observed, within zero and 60 seconds [1].

Notice that MOBUCON correlates different events on the basis of the *case*: i.e., it requires that every observed event belongs to a specific case, identified by a single case ID. To fulfill such requirement, we fed the MOBUCON monitor with the events logged by the Hadoop stack, and exploited the Map identifier (assigned by Hadoop to each mapper) as a *case ID*. This automatically ensures that each *Map start* event is indeed matched with the corresponding *Map end* event.

The constraint shown in Figure 4.2 allows us to detect mappers that are taking too much time to compute their task. The deadline set to 60 seconds has been chosen on the basis of the total completion time we want to respect while analyzing the simulation data. Naturally, some knowledge about the application domain is required to properly set such deadline. Mappers that violate the deadline are those that, unfortunately, were assigned a long task. This indeed would not be a problem for a single mapper. However it could become a problem if all the mappers get stuck on long tasks: having all the mappers busy on long tasks might undermine the completion of the whole bunch of data within a certain deadline. Note that, if the user doesn't have any knowledge of the volume of data to be processed – and consequently, the number of map tasks to be launched is not known *a priori* –, this methodology allow him

1 MOBUCON accepts deadlines at different time units. In this work we opted for expressing the time unit in terms of seconds, although depending on the application domain minutes or milliseconds might be better choices.

to still detect anomalies in the data that can require additional resources to speed up the computation. For example, the deadline for each map task can be computed at execution time by taking into account the average completion time for each completed mapper.

Besides supporting the monitoring of Declare constraints, MOBUCON supports also the definition of user-specific properties. We exploited this feature and expressed a further property by means of the Event Calculus language. The property, that we named *long_execution_maps*, aims to capture all the mappers that have already violated the deadline, and that are still *active* (i.e., a start event has been seen fo that mapper, and no end event has yet been observed). Such definition is given in terms of an Event Calculus axiom:

$$
\begin{aligned}
&\mathtt{initiates}(\\
&\qquad \mathtt{deadline\_expired}(A, \mathrm{ID}), \\
&\qquad \mathtt{status}(\mathtt{i}(\mathrm{ID}, \mathtt{long\_execution\_maps}), \mathtt{too\_long}), \\
&\qquad \mathrm{T} \\
&) \quad \leftarrow \\
&\qquad \mathtt{holds\_at}(\mathtt{status}(\mathtt{i}(\mathrm{ID}, \mathit{waiting\_task}), \mathtt{pend}), \mathrm{T}), \\
&\qquad \mathtt{holds\_at}(\mathtt{status}(\mathtt{i}(\mathrm{ID}, A), \mathit{active}), \mathrm{T}).
\end{aligned}
$$

We do not provide here all the details about the axiom (the interested reader can refer to [130] for an introduction to Event Calculus). Intuitively, the axiom specifies that at any time instant T, the happening of the event $\mathtt{deadline\_expired}(A, \mathrm{ID})$ initiates the property $\mathtt{long\_execution\_maps}$ with value $\mathtt{too\_long}$ for the mapper ID, if that mapper was still active and there was a constraint $\mathit{waiting\_task}$ still not fulfilled. The $\mathit{waiting\_task}$ constraint is indeed the response constraint we discussed in Figure 4.2.

With the $\mathtt{long\_execution\_maps}$ property we can determine within the MOBUCON monitor which are the mappers that got stuck on some task. However, to establish if a problem occur to the overall system, we should aggregate this information, and consider for each time instant *how many* mappers are stuck w.r.t. the total number of available mappers. Exploiting the MOBUCON feature of supporting also a *healthiness function*, we provided the following function:

$$
\text{System health} = 1 - \frac{\#\mathtt{long\_execution\_maps}}{\#\mathtt{total\_maps\_available}} \tag{4.1}
$$

In other words, the system health is expressed as the fraction of mappers that are not busy with a long task, over the total number of launched mappers. The lower the value, the higher the risk that the overall Hadoop framework gets stuck and violates some business deadline.

In order to make the health function more responsive, we can define a window of map task to be considered in the computation of system health.

4.3.3 The output from the MOBUCON monitor

In Figure 4.3 we show what happens when we analyze a run of the Hadoop architecture described in Chapter 4.3.1, with respect to the properties discussed in Chapter 4.3.2.

Figure 4.3 is composed of four strips, representing the evolution of different properties during the execution. From top to bottom of the figure we have: the health function, graphical representation of the Declare constraint, *long_execution_maps* property and description of the events occurred in each time interval. In the latter in particular (bottom part of Figure 4.3) the observer events has starting labels `ts` or `tc` to represent the start and the completion of a task, respectively. There are also a number of events starting with the label `time`: these events represent the ticking of a reference clock, used by MOBUCON to establish when deadlines are expired.

The health function on top of Figure 4.3 is the one defined in Eq. 4.1: indeed, the system healthiness dramatically decreased when six over seven of the first mappers launched in our testbed got stuck in a long execution task. The *long_execution_maps* strip (third strip from the top in Figure 4.3) further clarifies the intervals during which the long map tasks exceed their time deadline.

Finally, the Declare response constraint strip (second strip from the top in Figure 4.3) shows the status of each mapper: when the mapper is executing, the status is named *pending* and it is indicated with a yellow bar. As soon as there is information about the violation of a deadline (because of a tick event from the reference clock), the horizontal bar representing the status switched from *pending* to *violated*, and the color is changed from bright yellow to red. Notice that once violated (red color), the response constraint remains as such: indeed, this is a consequence of the Declare semantics where no compensation mechanisms are considered.

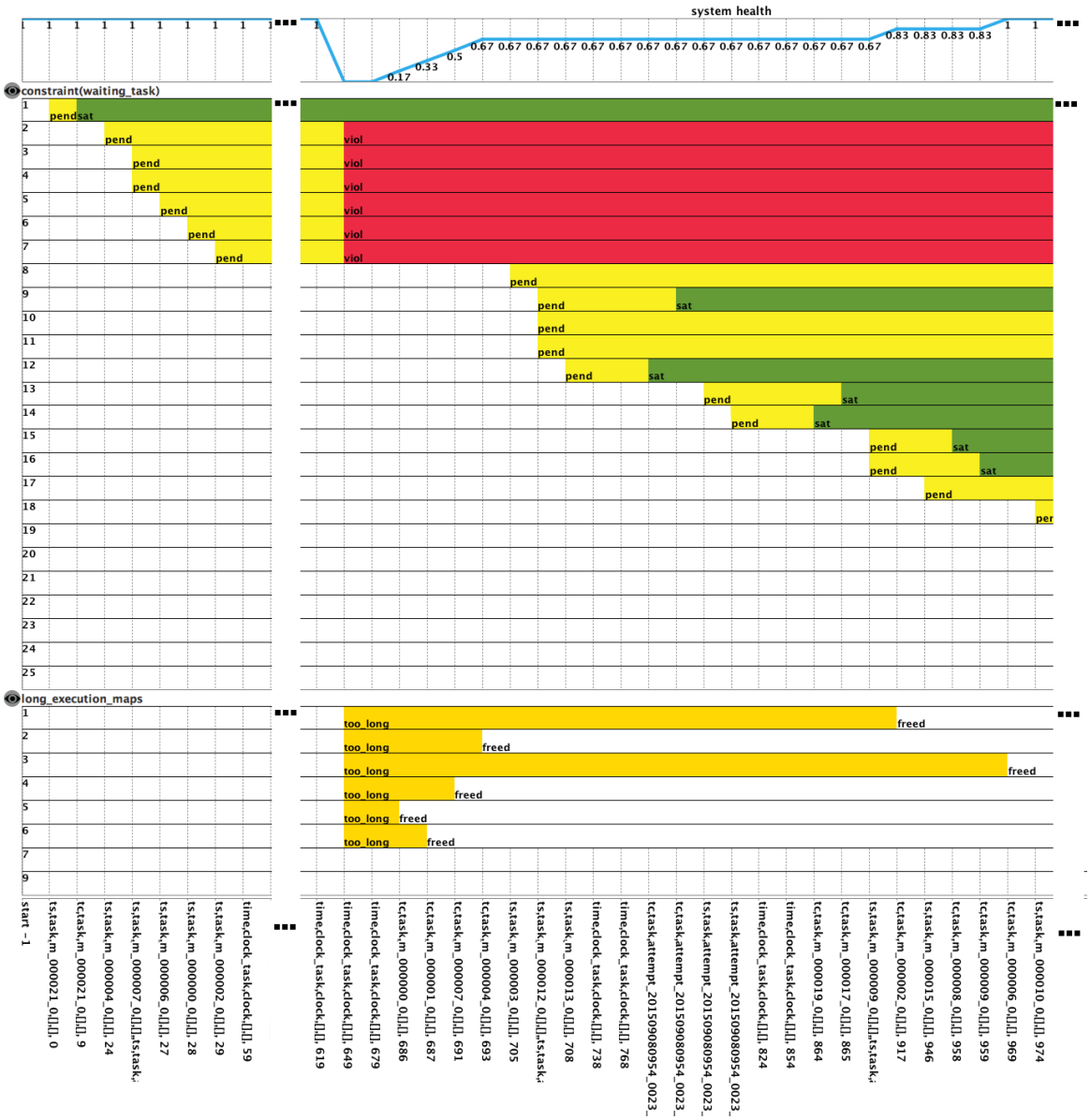


Figure 4.3: The output of the MOBUCON monitor for the execution of word count job on the given testbed.

For reasons of space, we provide in Figure 4.4 the evolution of our test (subsequent to what shown in Figure 4.3). As expected, the total number of mapper violating the deadline constraint is 8, as we provided 8 modified files in the input dataset. MOBUCON is therefore able to suddenly and efficiently identify any anomaly in the Hadoop execution (according to simple user-defined constraints).

The health function values in the output of MOBUCON monitor can be used to determine when a recovery action is needed. The intervention can be dynamically triggered by

a simple threshold mechanism over the health function or by a more complex user-defined policy (e.g., implementing an hysteresis cycle), possibly specified with a declarative approach.

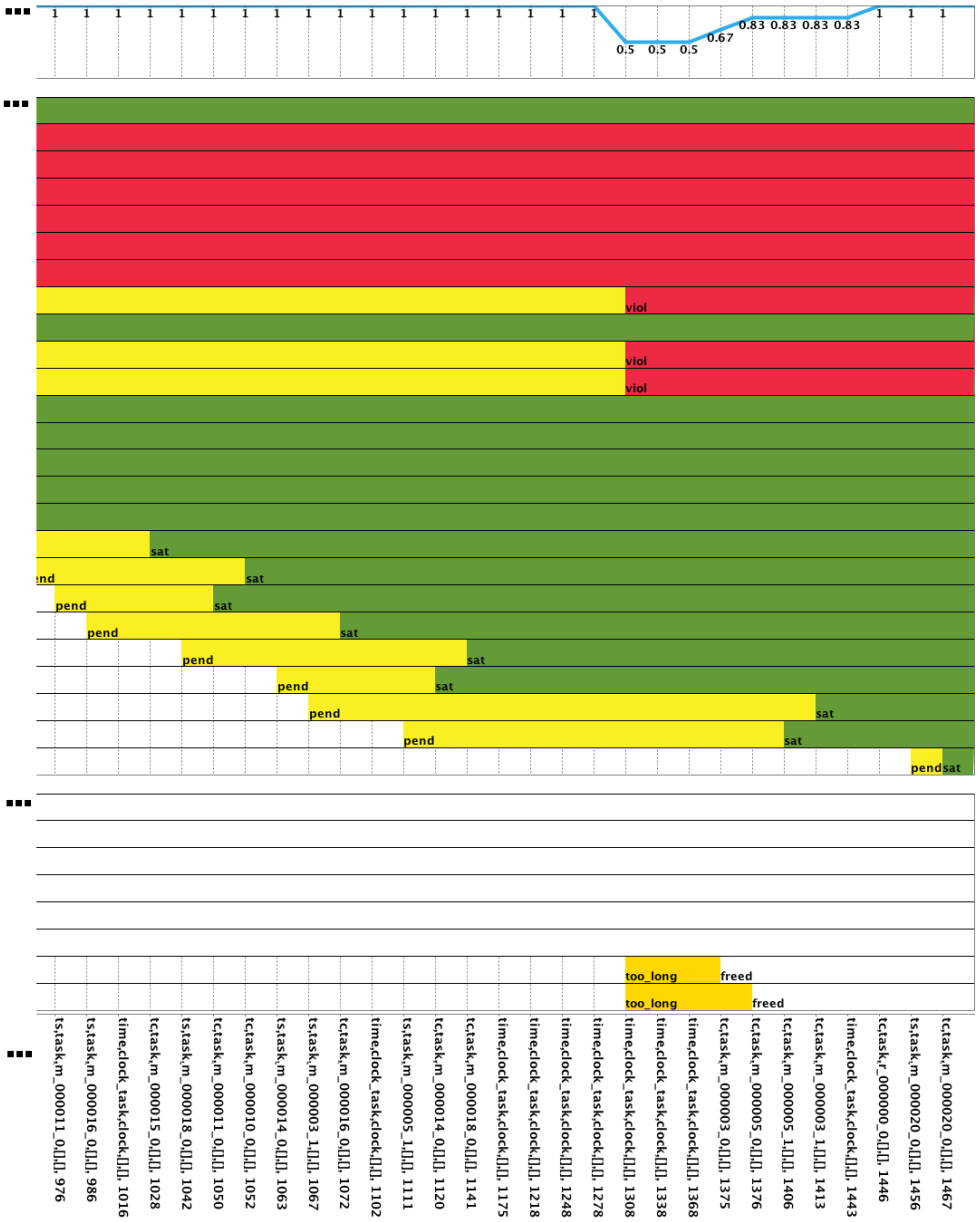


Figure 4.4: Output of the MOBUCON monitor subsequent to Figure 4.3.

Once the number of additional Hadoop workers needed is determined, MapReduce Auto-scaling Engine relays on HYLAAS

for the provisioning of VMs over a single public cloud or federated hybrid environment.

4.4 CONCLUSIONS AND FUTURE WORK

In this chapter, we presented a framework architecture that encapsulates an application level platform for data-processing. The system lends the MapReduce infrastructure the ability to autonomously check the execution, detecting bottlenecks and constraint violations through Business Process Management techniques with a *best effort* approach.

Focusing on *activities* and *constraints*, the use of Declare language has shown significant advantages in the monitoring system implementation and customization.

Although this work represents just a first step towards an auto-scaling engine for MapReduce, its declarative approach to the monitoring issue shows promising results, both regarding the reactivity to critical conditions and the simplification in monitoring constraint definition.

For the future, we plan to employ the defined framework architecture to test various diagnosis and recovery policies and verify the efficacy of the overall auto-scaling engine in a wider scenario (i.e., with a higher number of MapReduce workers involved).

Finally, particular attention must be given to the hybrid cloud scenario, where the HyIaaS component is employed to transparently perform VM provisioning either on an on-premise internal or an off-premise public cloud. In case of a hybrid deploy, several additional constraints can be taken into account by expressing them through declarative constraints. For example, the case of the limited inter-cloud bandwidth slowing-down the computation can be easily detected by the declarative monitoring engine that processes Hadoop logs thus on-line identifying the incidence of non-data-local computations.

While the monitoring policy implemented in this work is rather simple and can calculated on-the-fly during MapReduce job execution, we expect the introduction of additional constraints to further increase the computational complexity of the resulting monitoring and recovery policies. Therefore, an accurate study of the performance of our approach will be necessary. Nevertheless, we believe that a declarative approach

to the problem can contribute to significantly simplify the the work of the system administrator as he implements the monitoring policy by simply enunciating the set of properties the system must be compliant to.

5 CONCLUSIONS AND FUTURE DIRECTIONS

A lot of effort – in the academic as well as industrial research field – is concentrated on the autonomic management of virtual infrastructures and further encouraged by the rising need for enabling data-intensive applications leveraging a multiple-cloud environment.

5.1 SUMMARY

In this dissertation, we have presented our work on policies and mechanisms to enable the autonomic management of virtual infrastructures in cloud environments.

We initially focused on the single-cloud scenario, were the need for VM management is mostly determined by the fulfillment of two contrasting targets: restrain power waste and avoid performance degradation due to limited physical resource contention. After introducing the main works conducted in this filed, we presented our contribution to autonomic VM management: a decentralized solution for cloud virtual infrastructure, in which the physical hosts of the datacenter are able to self-organize and reach a global VM reallocation plan, according to a specific predefined goal. The implemented policies show good performance for various data centers dimensions in terms of both number of migrations requested and number of messages exchanged in order to converge to a common reallocation plan. Nevertheless, although very promising to improve the scalability of a single cloud scenario, decentralized solutions have shown substantial shortcomings from the performance point of view when compared to centralized solutions.

This finding has also influenced our latter research when dealing with autonomic management in the hybrid cloud scenario. Aiming to combine the virtual infrastructure administration strategies of both on- and off-premise cloud, the developed framework architecture presented in Chapter 3.3

has indeed a centralized nature. In fact, as regards the policy implementation, the proposed architecture partially distribute on the cloud compute nodes the monitoring and management logic duties (by arranging the correspondent components on each physical node). Nevertheless, we introduced a central coordination of the migration/spawning decisions, thus to prevent inconsistent evolutions of the strategy.

In this work, we also focused on the topic of data-intensive applications and, in particular, on the strategies to enable MapReduce execution over the hybrid cloud. The challenges of this field are due to the complexity of both the applicative scenario and the infrastructure level specificities.

For this reason, we focused on different policies aiming to autonomously manage the virtual clusters avoiding further complexity. A declarative approach in particular, has been employed in Chapter 4 in order to simplify the monitoring constraint definition for a MapReduce application running on a virtual infrastructure. The adopted approach has shown interesting and promising results.

5.2 CONCLUSIONS

The autonomic management of virtual resources in cloud datacenters is intrinsically complicated by the scale of the infrastructure, the variety of resources that need to be monitored and the contrasting goals (load balancing versus consolidation) to be pursued.

Decentralized solutions can be a good way to face this complexity by partitioning the management duties among different machines of the datacenter. Each node executes simple operations – according to a predefined policy – resulting in a complex emergent self-managing behavior of the infrastructure as a whole. Albeit the scalability and reliability of the system is improved when compered to centralized management architectures, our research has highlighted inevitable performance drawbacks in distributed solutions.

For this reason, when the scenario is further complicated by the presence of multiple interoperable clouds (such as in a hybrid cloud scenario), a combination of both centralized and distributed coordination techniques can represent a good choice to manage the complexity without compromising the performance.

Nevertheless, data-intensive tests conducted on hybrid infrastructures have highlighted the necessity to enrich the management system with application level monitoring information to *a priori* evaluate the performance improvements derived from VM migration/spawning decisions. The intricacies of taking into account both low- and high-level monitoring informations make essential the introduction of novel techniques – such as those borrowed from Business Process Management (BPM) area – to simplify the elicitation of the constraints to be runtime fulfilled by the system. Our research has shown that declarative methods can bring substantial improvements in this regard.

5.3 FUTURE RESEARCH DIRECTIONS

In this dissertation, we have explored several important research aspects of the autonomic infrastructure management problem but we have also identified many still open research questions worth pursuing. In this final chapter, we summarize the primary directions that, in our opinion, should drive future research efforts:

- *Improve cloud interoperability.* The cooperation between multiple clouds in order to offer an integrated resource provisioning service raises many more challenges than cloud computing. Beyond the challenge of enabling the cooperation through the adoption of common interfaces, other issues need to be faced. For example, the publication, discovery and selection of the services and resources offered by cooperating clouds is complicated by the scale of the environment, as well as the definition of a common set of SLAs that can be guaranteed to the customers by different providers. Another important challenge in this field is enabling inter-cloud migration, as it is complicated by the absence of a shared storage and the necessity to transfer memory, status and storage of the VMs trough a high latency medium. Furthermore, the monitoring systems of different clouds must implement strategies to cooperate in order to enable the autonomic management of the resulting whole set of resources.
- *Enhance decentralized techniques for management.* Although the performance of these strategies are still limited

when compared to those of centralized architectures, we believe that the benefits in terms of system scalability that a decentralized approach can bring to the autonomic management of a multiple clouds environment, justify further research in this field – especially if we consider the vision of an Inter-cloud environment, where each user can leverage and offer resources to a common "cloud of clouds".

- *Combine different monitoring/recovery approaches.* An interesting research direction in the field of federated/hybrid cloud computing is the definition of novel strategies for monitoring in order to combine the current status of the infrastructure (physical and virtual resource utilization) with information about the application level computation taking place (e.g., execution logs, applicative monitors, etc.). These challenge can be faced leveraging both classic optimization algorithms and declarative approaches. Substantially simplifying the definition of the expected system behavior (instead of focusing of all the possible error/undesired situations), we believe that declarative approaches are particularly promising to model both the monitoring and recovery of multiple-cloud management systems.

BIBLIOGRAPHY

[1] A. N. Toosi, R. N. Calheiros, and R. Buyya, "Interconnected cloud computing environments: Challenges, taxonomy, and survey," *ACM Comput. Surv.*, vol. 47, pp. 7:1–7:47, May 2014.

[2] K. Pepple, *Deploying OpenStack creating open source clouds*. O'Reilly, 2013.

[3] S. Rajan and A. Jairath, "Cloud computing: The fifth generation of computing," in *International Conference on Communication Systems and Network Technologies (CSNT)* (IEEE, ed.), vol. 15 - n 4, pp. 665–667, 2011.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," tech. rep., Berkeley, 2009.

[5] OpenStack community, "Openstack: Opensource cloud computing software." `https://www.openstack.org/`. Web Page, last visited in Oct. 2015.

[6] Amazon.com, Inc., "Amazon elastic compute cloud (ec2)." `http://aws.amazon.com/ec2`. Web Page, last visited in Oct. 2015.

[7] Aptana, Inc., "Aptana cloud." `http://aptana.com/cloud`. Web Page, last visited in Oct. 2015.

[8] Manjrasoft Pty Ltd. , "Aneka cloud." `http://www.manjrasoft.com/manjrasoft_company.html`, 2015. Web Page, last visited in Dec. 2015.

[9] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *The 34th ACM International Symposium on Computer Architecture*, pp. 13–23, ACM New York, 2007.

[10] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda, W. Emmerich, and F. Galan, "The

reservoir model and architecture for open federated cloud computing," *IBM Journal of Research and Development*, vol. 53, pp. 4:1–4:11, July 2009.

[11] D. Petcu, C. Crăciun, M. Neagul, S. Panica, B. Di Martino, S. Venticinque, M. Rak, and R. Aversa, "Architecturing a sky computing platform," in *Towards a Service-Based Internet. ServiceWave 2010 Workshops* (M. Cezon and Y. Wolfsthal, eds.), vol. 6569 of *Lecture Notes in Computer Science*, pp. 1–13, Springer Berlin Heidelberg, 2011.

[12] Jung, "Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures," in *International Conference on Distributed Computing Systems* (IEEE, ed.), pp. 62–73, June 2010.

[13] H. C. Lim, S. Babu, and J. S. Chase, "Automated control in cloud computing challeges and opportunities," in *ACDC '09, Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pp. 13–18, ACM New York, 2009.

[14] E. Kalyvianaki, "Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters," in *ICAC '09 Proceedings of the 6th international conference on Autonomic computing* (ACM, ed.), pp. 117–126, 2009.

[15] R. Jansen, "Energy efficient virtual machine allocation in the cloud," in *Green Computing Conference and Workshops (IGCC), 2011 International*, pp. 1–8, IEEE, July 2011.

[16] A. J. Younge, "Efficient resource management for cloud computing environments," in *Green Computing Conference, 2010 International*, pp. 357–364, IEEE, August 2010.

[17] A. Beloglazov and R. Buyya, "Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers," *Concurrency and Computation: Practice and Experience*, vol. 24, pp. 1397–1420, September 2012.

[18] J. Levine and F. Ducatelle, "Ant colony optimisation and local search for bin packing and cutting stock problems," *Journal of the Operational Research Society*, pp. 1–16, 2003.

[19] S. Zaman and D. Grosu, "Combinatorial auction-based allocation of virtual machine instances in clouds," in *2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)* (IEEE, ed.), pp. 127–134, December 2010.

[20] A. Marinos and G. Briscoe, "Community cloud computing," in *First International Conference, CloudCom 2009. Proceedings*, pp. 472–484, Springer Berlin Heidelberg, 2009.

[21] C. Giovanoli and S. G. Grivas, "Community clouds a centralized approach," in *CLOUD COMPUTING 2013, The Fourth International Conference on Cloud Computing, GRIDs, and Virtualization* (IARIA, ed.), pp. 43–48, 2013.

[22] K. Chard, S. Caton, O. Rana, and K. Bubendorfer, "Social cloud: Cloud computing in social networks," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, July 2010.

[23] R. Giordanelli, C. Mastroianni, and M. Meo, "Bio-inspired p2p systems: The case of multidimensional overlay," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 7, p. Article No. 35, December 2012.

[24] S. Balasubramaniam, K. Barrett, W. Donnelly, and S. V. D. Meer, "Bio-inspired policy based management (biopbm) for autonomic communications systems," in *7th IEEE International workshop on Policies for Distributed Systems and Networks* (IEEE, ed.), pp. 3–12, June 2006.

[25] M. Marzolla, O. Babaoglu, and F. Panzieri, "Server consolidation in clouds through gossiping," *Technical Report UBLCS-2011-01*, 2011.

[26] A. Vichos, *Agent-based management of Virtual Machines for Cloud infrastructure*. PhD thesis, School of Informatics, University of Edinburgh, 2011.

[27] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *INFOCOM, 2010 Proceedings IEEE*, March 2010.

[28] M. Sindelar, R. K. Sitaraman, and P. Shenoy, "Sharing-aware algorithms for virtual machine colocation," in

Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures, pp. 367–378, 2011.

[29] R. Lent, "Evaluating the performance and power consumption of systems with virtual machines," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pp. 778 – 783, 2011.

[30] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. Bhattacharya, "Virtual machine power metering and provisioning," in *ACM Symposium on Cloud Computing (SOCC)*, June 2010.

[31] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus, "Using control theory to achieve service level objectives in performance management. in proc. of im, 2002.," in *In Proc. of IM*, 2002.

[32] G. Soundararajan, C. Amza, and A. Goel, "Database replication policies for dynamic content applications," in *Proc. of EuroSys*, 2006.

[33] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic provisioning of multi-tier internet applications," in *Proc. of ICAC*, 2005.

[34] Joyent, Inc., "Joyent triton elastic container service." `http://www.joyent.com`. Web Page, last visited in Oct. 2015.

[35] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem, "Adaptive control of virtualized resources in utility computing environments," in *Proc. of EuroSys*, 2007.

[36] A. S. Foundation, "Apache tomcat." `http://tomcat.apache.org`. Web Page, last visited in Oct. 2015.

[37] D. Loreti, "Infrastruttura di supporto ad un'architettura cloud," Master's thesis, University of Bologna, 2009.

[38] C. Hyser, B. Mckee, R. Gardner, and B. J. Watson, "Autonomic virtual machine placement in the data center,"," tech. rep., HP Laboratories, December 2007.

[39] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future Generation Computer Systems*, vol. 28, pp. 755–768, May 2012.

[40] M. Yue, *A simple proof of the inequality FFD (L)<11/9 OPT (L)+ 1,for all l for the FFD bin-packing algorithm,.* Acta Mathematicae Applicatae Sinica, 1991.

[41] A. Verma, G. Dasgupta, T. Nayak, P. De, and R. Kothari, "Server workload analysis for power minimization using consolidation," *Proceedings of the 2009 USENIX Annual Technical Conference*, pp. 28–28, 2009.

[42] H. Abdi, *Multiple correlation coefficient*, pp. 648–651. Sage, Thousand Oaks, CA, 2007.

[43] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dinamic networks," *ACM Transaction on Computer Systems*, vol. 23, pp. 219–252, August 2005.

[44] M. Tighe, G. Keller, M. Bauer, and H. Lutfiyya, "A distributed approach to dynamic vm management," in *Network and Service Management (CNSM), 2013 9th International Conference on*, pp. 166–170, October 2013.

[45] D. Loreti and A. Ciampolini, "A decentralized approach for virtual infrastructure management in cloud datacenters," *International Journal On Advances in Intelligent Systems*, vol. 7, no. 3-4, pp. 507–518, 2014.

[46] D. Loreti and A. Ciampolini, "Policy for distributed self-organizing infrastructure management in cloud datacenters," in *Proceedings of The Tenth International Conference on Autonomic and Autonomous Systems*, IARIA, 2014.

[47] D. Loreti and A. Ciampolini, "Green-dam: a power-aware self-organizing approach for cloud infrastructure management," tech. rep., University of Bologna, 2013.

[48] D. Loreti and A. Ciampolini, "A distributed self-balancing policy for virtual machine management in cloud datacenters," in *High Performance Computing Simulation (HPCS), 2014 International Conference on*, pp. 391–398, July 2014.

[49] D. Loreti and A. Ciampolini, "A decentralized approach for virtual infrastructure management in cloud datacenters," *International Journal On Advances in Intelligent Systems*, vol. 7, pp. 507–518, December 2014.

[50] S. Martello and P. Toth, *Knapsack Problems*. Wiley, 1990.

[51] D. Loreti and A. Ciampolini, "Policy for distributed self-organizing infrastructure management in cloud data-centers," in *ICAS 2014, The Tenth International Conference on Autonomic and Autonomous Systems*, pp. 37–43, April 2014.

[52] D. Loreti, "Distributed autonomic manager simulator - damsim." `https://bitbucket.org/dloreti/cooperating.cloud.man`, 2013. Web Page, last visited in Oct. 2015.

[53] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, and C. Kim, "Vl2: a scalable and flexible data center network," in *SIGCOMM '09 Proceedings of the ACM SIGCOMM 2009 conference on Data communication* (ACM, ed.), pp. 51–62, October 2009.

[54] R. N. Calheiros, A. N. Toosi, C. Vecchiola, and R. Buyya, "A coordinator for scaling elastic applications across multiple clouds," *Future Generation Computer Systems*, vol. 28, no. 8, pp. 1350 – 1362, 2012. Including Special sections SS: Trusting Software Behavior and SS: Economics of Computing Services.

[55] T. Aoyama and H. Sakai, "Inter-cloud computing," *Business and Information Systems Engineering*, vol. 3, no. 3, pp. 173–177, 2011.

[56] R. Buyya, R. Ranjan, and R. Calheiros, "Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services," in *Algorithms and Architectures for Parallel Processing* (C.-H. Hsu, L. Yang, J. Park, and S.-S. Yeo, eds.), vol. 6081 of *Lecture Notes in Computer Science*, pp. 13–31, Springer Berlin Heidelberg, 2010.

[57] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile Networks and Applications*, vol. Volume 19, no. 2, pp. 171–209, 2014.

[58] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[59] Apache Software Foundation, "Hadoop mapreduce." `http://hadoop.apache.org/`. Web Page, last visited in Dec. 2013.

[60] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond, and M. Morrow, "Blueprint for the intercloud - protocols and formats for cloud computing interoperability," in *Internet and Web Applications and Services, 2009. ICIW '09. Fourth International Conference on*, pp. 328–336, May 2009.

[61] T. Kurze, M. Klemsy, D. Bermbachy, A. Lenkz, S. Taiy, and M. Kunze., "Cloud federation," in *Proceedings of the 2nd International Conference on Cloud Computing, GRIDs, and Virtualization*, pp. 32–38, 2011.

[62] D. Chen and G. Doumeingts, "European initiatives to develop interoperability of enterprise applicationsâ€"basic concepts, framework and roadmap," *Annual Reviews in Control*, vol. 27, no. 2, pp. 153 – 162, 2003.

[63] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, pp. 50–58, Apr. 2010.

[64] Google Inc., "App engine - google cloud platform." `https://developers.google.com/appengine/`. Web Page, last visited in Dec. 2015.

[65] A. J. Ferrer, F. Hernandez, J. Tordsson, E. Elmroth, A. Ali-Eldin, C. Zsigri, R. Sirvent, J. Guitart, R. M. Badia, K. Djemame, W. Ziegler, T. Dimitrakos, S. K. Nair, G. Kousiouris, K. Konstanteli, T. Varvarigou, B. Hudzia, A. Kipp, S. Wesner, M. Corrales, N. ForgÃ³, T. Sharif, and C. Sheridan, "Optimis: A holistic approach to cloud service provisioning," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 66 – 77, 2012.

[66] D. Villegas, N. Bobroff, I. Rodero, J. Delgado, Y. Liu, A. Devarakonda, L. Fong, S. M. Sadjadi, and M. Parashar, "Cloud federation in a layered service model," *Journal of Computer and System Sciences*, vol. 78, no. 5, pp. 1330 – 1344, 2012. {JCSS} Special Issue: Cloud Computing 2011.

[67] Apache Software Foundation, "Apachespark." `http://spark.apache.org`, 2015. Web Page, last visited in Dec. 2015.

[68] Jaliya Ekanayake. Supported by SALSA Team at Indiana University, "Twister, iterative mapreduce." `http:`

//www.iterativemapreduce.org, 2015. Web Page, last visited in Dec. 2015.

[69] Google Inc., "Google mapreduce for appengine.." https://cloud.google.com/appengine/docs/python/dataprocessing/, 2015. Web Page, last visited in Dec. 2015.

[70] H. Zhang, G. Jiang, K. Yoshihira, H. Chen, and A. Saxena, "Intelligent workload factoring for a hybrid cloud computing model," in *Services - I, 2009 World Conference on*, pp. 701–708, July 2009.

[71] C.-H. Suen, M. Kirchberg, and B. S. Lee, "Efficient migration of virtual machines between public and private cloud," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pp. 549–553, Nov 2011.

[72] E. Collins, "Intersection of the cloud and big data," *Cloud Computing, IEEE*, vol. 1, pp. 84–85, May 2014.

[73] W.-T. Tsai, P. Zhong, J. Elston, X. Bai, and Y. Chen, "Service replication strategies with mapreduce in clouds," in *Autonomous Decentralized Systems (ISADS), 2011 10th International Symposium on*, pp. 381–388, March 2011.

[74] F. Tian and K. Chen, "Towards optimal resource provisioning for running mapreduce programs in public clouds," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pp. 155–162, July 2011.

[75] T. Gunarathne, T.-L. Wu, J. Qiu, and G. Fox, "Mapreduce in the clouds for science," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pp. 565–572, Nov 2010.

[76] X. Zhang, L. Yang, C. Liu, and J. Chen, "A scalable two-phase top-down specialization approach for data anonymization using mapreduce on cloud," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, pp. 363–373, Feb 2014.

[77] B. Nicolae, P. Riteau, and K. Keahey, "Bursting the cloud data bubble: Towards transparent storage elasticity in iaas clouds," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 135–144, May 2014.

[78] B. Nicolae, P. Riteau, and K. Keahey, "Transparent throughput elasticity for iaas cloud storage using guest-side block-level caching," in *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pp. 186–195, Dec 2014.

[79] K. Chen, J. Powers, S. Guo, and F. Tian, "Cresp: Towards optimal resource provisioning for mapreduce computing in public clouds," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, pp. 1403–1412, June 2014.

[80] B. Palanisamy, A. Singh, and L. Liu, "Cost-effective resource provisioning for mapreduce in a cloud," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, pp. 1265–1279, May 2015.

[81] N. B. Rizvandi, J. Taheri, R. Moraveji, and A. Y. Zomaya, "A study on using uncertain time series matching algorithms for mapreduce applications," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1699–1718, 2013.

[82] A. Verma, L. Cherkasova, and R. H. Campbell, *Resource Provisioning Framework for MapReduce Jobs with Performance Goals*, vol. 7049 of *Lecture Notes in Computer Science*, pp. 165–186. Springer Berlin Heidelberg, 2011.

[83] M. Cardosa, C. Wang, A. Nangia, A. Chandra, and J. Weissman, "Exploring mapreduce efficiency with highly-distributed data," in *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, (New York, NY, USA), pp. 27–34, ACM, 2011.

[84] T. Bicer, D. Chiu, and G. Agrawal, "A framework for data-intensive computing with cloud bursting," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pp. 169–177, Sept 2011.

[85] K. Nagin, D. Hadas, Z. Dubitzky, A. Glikson, I. Loy, B. Rochwerger, and L. Schour, "Inter-cloud mobility of virtual machines," in *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, (New York, NY, USA), pp. 3:1–3:12, ACM, 2011.

[86] T. Guo, U. Sharma, P. Shenoy, T. Wood, and S. Sahu, "Cost-aware cloud bursting for enterprise applications,"

ACM Trans. Internet Technol., vol. 13, pp. 10:1–10:24, May 2014.

[87] H. Zhang, G. Jiang, K. Yoshihira, and H. Chen, "Proactive workload management in hybrid cloud computing," *Network and Service Management, IEEE Transactions on*, vol. 11, pp. 90–100, March 2014.

[88] M. Mattess, R. Calheiros, and R. Buyya, "Scaling mapreduce applications across hybrid clouds to meet soft deadlines," in *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pp. 629–636, March 2013.

[89] S. Kailasam, P. Dhawalia, S. Balaji, G. Iyer, and J. Dharanipragada, "Extending mapreduce across clouds with bstream," *Cloud Computing, IEEE Transactions on*, vol. 2, pp. 362–376, July 2014.

[90] B. Sharma, T. Wood, and C. Das, "Hybridmr: A hierarchical mapreduce scheduler for hybrid data centers," in *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pp. 102–111, July 2013.

[91] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads," *Proc. VLDB Endow.*, vol. 2, pp. 922–933, August 2009.

[92] K. Shirahata, H. Sato, and S. Matsuoka, "Hybrid map task scheduling for gpu-based heterogeneous clusters," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pp. 733–740, Nov 2010.

[93] M. M. Rafique, A. R. Butt, and D. S. Nikolopoulos, "A capabilities-aware framework for using computational accelerators in data-intensive computing," *J. Parallel Distrib. Comput.*, vol. 71, pp. 185–197, Feb. 2011.

[94] M. Rafique, B. Rose, A. Butt, and D. Nikolopoulos, "Cellmr: A framework for supporting mapreduce on asymmetric cell-based clusters," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–12, May 2009.

[95] R. Van den Bossche, K. Vanmechelen, and J. Broeckhove, "Cost-optimal scheduling in hybrid iaas clouds for deadline constrained workloads," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pp. 228–235, July 2010.

[96] S. Imai, T. Chestna, and C. A. Varela, "Accurate resource prediction for hybrid iaas clouds using workload-tailored elastic compute units," in *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, UCC '13, (Washington, DC, USA), pp. 171–178, IEEE Computer Society, 2013.

[97] D. Loreti and A. Ciampolini, "A hybrid cloud infrastructure for big data applications," in *Proceedings of the 17th International Conferences on High Performance Computing and Communications*, IEEE, 2015.

[98] D. Loreti and A. Ciampolini, "Shyam: a system for autonomic management of virtual clusters in hybrid clouds," in *1st Workshop on Federated Cloud Networking (FedCloudNet)*, Springer, 2015.

[99] D. Loreti and A. Ciampolini, "Mapreduce over the hybrid cloud: a novel infrastructure management policy," in *Proceedings of the 8th IEEE/ACM International Conference on Utility and Cloud Computing*, 2015.

[100] F. J. Clemente-Castellò, B. Nicolae, K. Katrinis, M. M. Rafique, R. Mayo, J. C. Fernàndez, and D. Loreti, "Enabling big data analytics in the hybrid cloud using iterative mapreduce," in *Proceedings of the 8th IEEE/ACM International Conference on Utility and Cloud Computing*, 2015.

[101] Eucalyptus Systems, Inc., "Eucalyptus." `https://www.eucalyptus.com/`. Web Page, last visited in Dec. 2015.

[102] Apache Software Foundation, "Cloudstack." `https://cloudstack.apache.org/`, 2015. Web Page, last visited in Dec. 2015.

[103] OpenNebula community, "Opennebula." `https://www.opennebula.org/`. Web Page, last visited in Dec. 2015.

[104] Open Grid Forum community, "Open grid forum." `https://www.ogf.org/`. Web Page, last visited in Dec. 2015.

[105] Open Grid Forum community, "Open cloud computing interface." https://www.occi-wg.org/. Web Page, last visited in Dec. 2015.

[106] Rackspace, Inc., "Rackspace public cloud." https://developer.rackspace.com. Web Page, last visited in Dec. 2015.

[107] Hewlett Packard Enterprise, "Hpe helion public cloud." https://hpcloud.com. Web Page, last visited in Dec. 2015.

[108] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Sixth symposium on operating system design and implementation*, (San Francisco, CA), December 2004.

[109] OpenStack community, "Sahara: module for data processing." https://wiki.openstack.org/wiki/Sahara. Web Page, last visited in Oct. 2015.

[110] OpenStack community, "Ceilometer: monitoring module." https://wiki.openstack.org/wiki/Ceilometer, 2015. Web Page, last visited in Oct. 2015.

[111] PUMA, "Benchmarks and dataset downloads." https://engineering.purdue.edu/~puma/datasets.htm. Web Page, last visited in Oct. 2015.

[112] Apache Software Foundation, "Apache hadoop balancer." https://hadoop.apache.org/docs/r1.0.4/commands_manual.html. Web Page, last visited in Oct. 2015.

[113] D. Loreti and A. Ciampolini, "A hybrid cloud infrastructure fo big data applications," in *Proceedings of IEEE International Conferences on High Performance Computing and Communications*, 2015.

[114] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–10, May 2010.

[115] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: Automatic resource inference and allocation for mapreduce environments," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, (New York, NY, USA), pp. 235–244, ACM, 2011.

[116] H.-H. Bock, "Clustering methods: A history of k-means algorithms," in *Selected Contributions in Data Analysis and Classification*, Springer Berlin Heidelberg, 2007.

[117] W. Zhao, H. Ma, and Q. He, "Parallel k-means clustering based on mapreduce," in *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, (Berlin, Heidelberg), pp. 674–679, Springer-Verlag, 2009.

[118] W. V. D. Aalst, A. Adriansyah, A. K. A. de Medeiros, and F. Arcieri, "Process mining manifesto," in *Business Process Management Workshops*, Springer Berlin Heidelberg, 2012.

[119] Amazon Inc., "Amazon cloud watch: Amazon cloud monitor system.." `https://aws.amazon.com/it/cloudwatch/`, 2015. Web Page, last visited in Oct. 2015.

[120] W. van der Aalst, A. ter Hofstede, and M. Weske, "Business process management: A survey," in *Business Process Management* (W. van der Aalst and M. Weske, eds.), vol. 2678 of *Lecture Notes in Computer Science*, pp. 1–12, Springer Berlin Heidelberg, 2003.

[121] M. Shanahan, "The event calculus explained," in *Artificial Intelligence Today* (M. Wooldridge and M. Veloso, eds.), vol. 1600 of *Lecture Notes in Computer Science*, pp. 409–430, Springer Berlin Heidelberg, 1999.

[122] M. Montali, F. M. Maggi, F. Chesani, P. Mello, and W. M. P. van der Aalst, "Monitoring business constraints with the event calculus," *ACM TIST*, vol. 5, no. 1, p. 17, 2013.

[123] M. Pesic and W. M. P. van der Aalst, "A Declarative Approach for Flexible Business Processes Management," in *Business Process Management Workshops*, LNCS, SPRINGER, 2006.

[124] D. Loreti, A. Ciampolini, F. Chesani, and P. Mello, "Process mining monitoring for map reduce applications in the cloud," in *Submitted to the 6th International Conference on Cloud Computing and Services Science*, 2016.

[125] G. Spanoudakis and K. Mahbub, "Non-intrusive monitoring of service-based systems," *International Journal of Cooperative Information Systems*, vol. 15, no. 03, pp. 325–358, 2006.

[126] A. Farrel, M. Sergot, M. Sallè, and C. Bartolini, "Using the event calculus for tracking the normative state of contracts," *International Journal of Cooperative Information Systems*, vol. 14, no. 02n03, pp. 99–129, 2005.

[127] D. Giannakopoulou and K. Havelund, "Automata-based verification of temporal properties on running programs," in *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*, pp. 412–416, Nov 2001.

[128] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for ltl and tltl," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, pp. 14:1–14:64, Sept. 2011.

[129] M. Montali, F. Chesani, P. Mello, and F. M. Maggi, "Towards data-aware constraints in declare," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, pp. 1391–1396, 2013.

[130] R. A. Kowalski and M. J. Sergot, "A Logic-Based Calculus of Events," *New Generation Computing*, 1986.

[131] S. Bragaglia, F. Chesani, P. Mello, M. Montali, and P. Torroni, "Reactive event calculus for monitoring global computing applications," in *Logic Programs, Norms and Action*, Springer, 2012.

ACKNOWLEDGEMENTS

I would like to thank Prof. Anna Ciampolini for helping me with competence and patience during every step of my Ph.D. studies. Her support and understanding went far beyond the role of a supervisor. Sincere thanks goes to Prof. Federico Chesani, Prof. Antonio Corradi and Prof. Paola Mello for their precious advices and support during these years. I also want to thank Dr. Pól Mac Aonghusa and Dr. Konstantinos Katrinis for giving me the chance to work with them at the IBM Dublin Research Lab; and the reviewers of this thesis, Prof. Rajkumar Buyya and Dr. Mustafa Rafique for their many suggestions that helped to improve the quality of this work.

Thanks to all the LIA guys – Raffaele, Federico, Ambra, Allegra, Thomas, Alessio, Luca, Andrea, Marco and Roberta – for respectfully sharing office, work and ultimately – given the number of hours we spend together – life. A special acknowledgment goes to all the ex-LIA for the memorable lunch discussions: Carlo, Giuseppe, Fede, Andrea, and Stefano. I still laugh by myself when I think back to some jokes. Your cynicism, kindness, bitterness and friendship made my first years of Ph.D. studies unmatchable.

I also want to thank my "preo" friends spread around the world: Gigi, Prandi, Bussi and Wero, who shared with me many absolutely silly yet incredibly important moments. I cannot forget to be grateful to the friends who are with me since unmemorable times – Piera and Garde – for always been a save shore to me, for the wonderful years: those gone by and those still to come. I don't know how I would do without you. Also thanks to the friends that arrived in the last few years and, nevertheless, contributed to make the journey preciously bright – the Fabulous Girls: Briga, Mary, Bea, Simo, Chiara, Elena, Marika and Tina.

Finally, the most important acknowledgements are for my family, present and future. My parents and my sister for always being how a family should be: understanding, supportive, messy and present whenever I need them. Your pride for each goal I achieve is pure strength to me. Alberto, my life and my future, for supporting me when the choice of Ph.D. studies was still a dangerous and confused dream: "it sounds reasonable but not yet reasoned, let's focus". Thank you for encouraging me when I was about to give up, for turning every obstacle into a joke and, of course, for your priceless love.

fin.