UNIVERSITÉ PARIS-SUD

ECOLE DOCTORALE D'INFORMATIQUE DE PARIS-SUD
INRIA SACLAY ÎLE-DE-FRANCE/LABORATORIE DE RECHERCHE EN INFORMATIQUE

DISCIPLINE : INFORMATIQUE

En Cotutelle Internationale avec
UNIVERSITÉ DE BRASÍLIA, BRÉSIL

THÈSE DE DOCTORAT

Soutenue le 2 décembre 2014 par

Alessandro FERREIRA LEITE

A User-Centered and Autonomic Multi-Cloud Architecture for High Performance Computing Applications

To my family and my parents
To my niece and my nephews

Acknowledgements

First, I would like to thank my supervisors for their support and advices. Next, I would like to thank my thesis's reviewers Christophe Cérin and Jean-Louis Pazat, and the members of my thesis committee: Christine Froidevaux, Célia Ghedini Ralha, and Christine Morin. I appreciate all yours comments and questions.

Additionally, I would like to thank the professors Vander Alves, Genaína Rodrigues, Li Weigang, and Jacques Blanc for their comments, suggestions, and discussions.

Next, I am very thankful to Katia Evrat and Valérie Berthou for kindly helping me in various administrative works. In addition, I am grateful to the authors that sent me their papers when I requested them.

After, I would like to thank all my team members in INRIA, especially Michael Kruse and Taj Muhammad Khan. I also thank my friends Cícero Roberto, Antônio Junior, Éric Silva, André Ribeiro, Francinaldo Araujo, Emerson Macedo, and the former Logus's team. Moreover, I would like to thank the people I collaborated with during the last years.

I further acknowledge the financial assistance of CAPES/CNPq through the program science without borders (grant 237561/2012-3) during the year of 2013, and Campus France/INRIA (project Pascale) during the period of February to July and November to December 2014.

Last but not least, I would like to thank my parents and my family for their unconditional support and patience.

Abstract

Cloud computing has been seen as an option to execute high performance computing (HPC) applications. While traditional HPC platforms such as grid and supercomputers offer a stable environment in terms of failures, performance, and number of resources, cloud computing offers on-demand resources generally with unpredictable performance at low financial cost. Furthermore, in cloud environment, failures are part of its normal operation. To overcome the limits of a single cloud, clouds can be combined, forming a cloud federation often with minimal additional costs for the users. A cloud federation can help both cloud providers and cloud users to achieve their goals such as to reduce the execution time, to achieve minimum cost, to increase availability, to reduce power consumption, among others. Hence, cloud federation can be an elegant solution to avoid over provisioning, thus reducing the operational costs in an average load situation, and removing resources that would otherwise remain idle and wasting power consumption, for instance. However, cloud federation increases the range of resources available for the users. As a result, cloud or system administration skills may be demanded from the users, as well as a considerable time to learn about the available options. In this context, some questions arise such as: (a) which cloud resource is appropriate for a given application? (b) how can the users execute their HPC applications with acceptable performance and financial costs, without needing to re-engineer the applications to fit clouds' constraints? (c) how can non-cloud specialists maximize the features of the clouds, without being tied to a cloud provider? and (d) how can the cloud providers use the federation to reduce power consumption of the clouds, while still being able to give service-level agreement (SLA) guarantees to the users? Motivated by these questions, this thesis presents a SLA-aware application consolidation solution for cloud federation. Using a multi-agent system (MAS) to negotiate virtual machine (VM) migrations between the clouds, simulation results show that our approach could reduce up to 46% of the power consumption, while trying to meet performance requirements. Using the federation, we developed and evaluated an approach to execute a huge bioinformatics application at zero-cost. Moreover, we could decrease the execution time in 22.55% over the best single cloud execution. In addition, this thesis presents a cloud architecture called Excalibur to auto-scale cloud-unaware application. Executing a genomics workflow, Excalibur could seamlessly scale the applications up to 11 virtual machines, reducing the execution time by 63% and the cost by 84% when compared to a user's configuration. Finally, this thesis presents a software product line engineering (SPLE) method to handle the commonality and variability of infrastructure-as-a-service (IaaS) clouds, and an autonomic multi-cloud architecture that uses this method to configure and to deal with failures autonomously. The SPLE method uses extended feature model (EFM) with attributes to describe the resources and to select them based on the users' objectives. Experiments realized with two different cloud providers show that using the proposed method, the users could execute their application on a federated cloud environment, without needing to know the variability and constraints of the clouds.

Keywords: Autonomic computing; High performance computing (HPC); Cloud federation; Software Product Line Engineering, Variability Management of Cloud Infrastructure

Résumé

Le cloud computing a été considéré comme une option pour exécuter des applications de calcul haute performance (HPC). Bien que les plateformes traditionnelles de calcul haute performance telles que les grilles et les supercalculateurs offrent un environnement stable du point de vue des défaillances, des performances, et de la taille des ressources, le cloud computing offre des ressources à la demande, généralement avec des performances imprévisibles mais à des coûts financiers abordables. En outre, dans un environnement de cloud, les défaillances sont perçues comme étant ordinaires. Pour surmonter les limites d'un cloud individuel, plusieurs clouds peuvent être combinés pour former une fédération de clouds, souvent avec des coûts supplémentaires légers pour les utilisateurs. Une fédération de clouds peut aider autant les fournisseurs que les utilisateurs à atteindre leurs objectifs tels la réduction du temps d'exécution, la minimisation des coûts, l'augmentation de la disponibilité, la réduction de la consommation d'énergie, pour ne citer que ceux-là. Ainsi, la fédération de clouds peut être une solution élégante pour éviter le sur-approvisionnement, réduisant ainsi les coûts d'exploitation en situation de charge moyenne, et en supprimant des ressources qui, autrement, resteraient inutilisées et gaspilleraient ainsi de énergie. Cependant, la fédération de clouds élargit la gamme des ressources disponibles. En conséquence, pour les utilisateurs, des compétences en cloud computing ou en administration système sont nécessaires, ainsi qu'un temps d'apprentissage considérable pour maîtrises les options disponibles. Dans ce contexte, certaines questions se posent : (a) Quelle ressource du cloud est appropriée pour une application donnée ? (b) Comment les utilisateurs peuvent-ils exécuter leurs applications HPC avec un rendement acceptable et des coûts financiers abordables, sans avoir à reconfigurer les applications pour répondre aux normes et contraintes du cloud ? (c) Comment les non-spécialistes du cloud peuvent-ils maximiser l'usage des caractéristiques du cloud, sans être liés au fournisseur du cloud ? et (d) Comment les fournisseurs de cloud peuvent-ils exploiter la fédération pour réduire la consommation électrique, tout en étant en mesure de fournir un service garantissant les normes de qualité préétablies ? À partir de ces questions, la présente thèse propose une solution de consolidation d'applications pour la fédération de clouds qui garantit le respect des normes de qualité de service. On utilise un système multi-agents (SMA) pour négocier la migration des machines virtuelles entre les clouds. Les résultats de simulations montrent que notre approche pourrait réduire jusqu'à 46% la consommation totale d'énergie, tout en respectant les exigences de performance. En nous basant sur la fédération de clouds, nous avons développé et évalué une approche pour exécuter une énorme application de bioinformatique à coût zéro. En outre, nous avons pu réduire le temps d'exécution de 22,55% par rapport à la meilleure exécution dans un cloud individuel. Cette thèse présente aussi une architecture de cloud baptisée « *Excalibur* » qui permet l'adaptation automatique des applications standards pour le cloud. Dans l'exécution d'une chaîne de traitements de la génomique, Excalibur a pu parfaitement mettre à l'échelle les applications sur jusqu'à 11 machines virtuelles, ce qui a réduit le temps d'exécution de 63% et le coût de 84% par rapport à la configuration de l'utilisateur. Enfin, cette thèse présente un processus d'ingénierie des lignes de produits (PLE) pour gérer la variabilité de l'infrastructure à la demande du cloud, et une architecture multi-cloud autonome qui utilise ce processus pour configurer et faire face aux défaillances de manière indépendante. Le processus PLE utilise le modèle étendu de fonction (EFM) avec des attributs pour décrire les ressources et les sélectionner en fonction des objectifs de l'utilisateur. Les expériences réalisées avec deux fournisseurs de cloud différents montrent qu'en utilisant le modèle proposé, les utilisateurs peuvent exécuter leurs applications dans un environnement de clouds fédérés, sans avoir besoin de connaître les variabilités et contraintes du cloud.

Mots-clés: Calcul Autonomique; Calcul Haute Performance (HPC); Cloud Federation; Ligne de Produits Logiciels; Modèles de Variabilité

Resumo

A computação em nuvem tem sido considerada como uma opção para executar aplicações de alto desempenho. Entretanto, enquanto as plataformas de alto desempenho tradicionais como *grid* e supercomputadores oferecem um ambiente estável quanto à falha, desempenho e número de recursos, a computação em nuvem oferece recursos sob demanda, geralmente com desempenho imprevisível à baixo custo financeiro. Além disso, em ambiente de nuvem, as falhas fazem parte da sua normal operação. No entanto, as nuvens podem ser combinadas, criando uma federação, para superar os limites de uma nuvem muitas vezes com um baixo custo para os usuários. A federação de nuvens pode ajudar tanto os provedores quanto os usuários das nuvens a atingirem diferentes objetivos tais como: reduzir o tempo de execução de uma aplicação, reduzir o custo financeiro, aumentar a disponibilidade do ambiente, reduzir o consumo de energia, entre outros. Por isso, a federação de nuvens pode ser uma solução elegante para evitar o sub-provisionamento de recursos ajudando os provedores a reduzirem os custos operacionais e a reduzir o número de recursos ativos, que outrora ficariam ociosos consumindo energia, por exemplo. No entanto, a federação de nuvens aumenta as opções de recursos disponíveis para os usuários, requerendo, em muito dos casos, conhecimento em administração de sistemas ou em computação em nuvem, bem como um tempo considerável para aprender sobre as opções disponíveis. Neste contexto, surgem algumas questões, tais como: (a) qual dentre os recursos disponíveis é apropriado para uma determinada aplicação? (b) como os usuários podem executar suas aplicações na nuvem e obter um desempenho e um custo financeiro aceitável, sem ter que modificá-las para atender as restrições do ambiente de nuvem? (c) como os usuários não especialistas em nuvem podem maximizar o uso da nuvem, sem ficar dependente de um provedor? (d) como os provedores podem utilizar a federação para reduzir o consumo de energia dos datacenters e ao mesmo tempo atender os acordos de níveis de serviços? A partir destas questões, este trabalho apresenta uma solução para consolidação de aplicações em nuvem federalizadas considerando os acordos de serviços. Nossa solução utiliza um sistema multi-agente para negociar a migração das máquinas virtuais entres as nuvens. Simulações mostram que nossa abordagem pode reduzir em até 46% o consumo de energia e atender os requisitos de qualidade. Nós também desenvolvemos e avaliamos uma solução para executar uma aplicação de bioinformática em nuvens federalizadas, a custo zero. Nesse caso, utilizando a federação, conseguimos diminuir o tempo de execução da aplicação em 22,55%, considerando o seu tempo de execução na melhor nuvem. Além disso, este trabalho apresenta uma arquitetura chamada *Excalibur*, que possibilita escalar a execução de aplicações comuns em nuvem. Excalibur conseguiu escalar automaticamente a execução de um conjunto de aplicações de bioinformática em até 11 máquinas virtuais, reduzindo o tempo de execução em 63% e o custo financeiro em 84% quando comparado com uma configuração definida pelos usuários. Por fim, este trabalho apresenta um método baseado em linha de produto de software para lidar com as variabilidades dos serviços oferecidos por nuvens de infraestrutura (IaaS), e um sistema que utiliza deste processo para configurar o ambiente e para lidar com falhas de forma automática. O nosso método utiliza modelo de *feature* estendido com atributos para descrever os recursos e para selecioná-los com base nos objetivos dos usuários. Experimentos realizados com dois provedores diferentes mostraram que utilizando o nosso processo, os usuários podem executar as suas aplicações em um ambiente de nuvem federalizada, sem conhecer as variabilidades e limitações das nuvens.

Palavras-chave: Computação Autonômica; Computação de Alto Desempenho; Federação de Nuvens; Linha de Produto de Software; Modelo de Features

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

AE	Autonomic Element
AM	Autonomic Manager
API	Application Programming Interface
AT	Allocation Trust
AWS	Amazon Web Services
BPEL	Business Process Execution Language
CA	Cloud Availability
CDMI	Cloud Data Management Interface
CE	Cost-Effectiveness
CERA	Carbon Emission Regulator Agency
CIMI	Cloud Infrastructure Management Interface
CK	Configuration Knowledge
Clafer	Class feature reference
CloVR	Cloud Virtual Service
CLSP	Cloud Service Provider
CLU	Cloud User
CORBA	Common Object Request Broker Architecture
CP	Constraint Programming
CPE	Compute Power Efficiency
CSP	Constraint Satisfaction Problem
DAG	Directed Acyclic Graph
DAOP	Dynamic Aspect-Oriented Programming
DCD	Data Center Density
DCeP	Data Center Energy Productivity
DCiE	Data Center Infrastructure Efficiency
DCOM	Distributed Computing Object Model
DCPE	Data Center Performance Efficiency
DFS	Distributed File System
DHT	Distributed Hash Table

DMTF	Distributed Management Task Force
DPM	Dynamic Power Management
DSL	Domain Specific Language
DSTM	Distributed Software Transaction Memory
DTM	Distributed Transaction Memory
DVFS	Dynamic Voltage and Frequency Scaling
EC2	Elastic Compute Cloud
ECA	Event-Condition-Action
ECU	Elastic Computing Unit
EFM	Extended Feature Model
EPP	Electric Power Provider
FAP	Federated Application Provisioning
FM	Feature Model
FODA	Feature-Oriented Domain Analysis
GAE	Google App Engine
GCE	Google Compute Engine
GCEUs	Google Compute Engine Units
GCUP	Billions of Cell Updates per Second
GGF	Global Grid Forum
GHG	Greenhouse Gas
GPI	Green Performance Indicator
GUI	Graphical User Interface
HPC	High Performance Computing
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure-as-a-Service
IC	Instance Capability
IE	Instance Efficiency
IM	Infrastructure Manager
IOPS	Input/Output Operations per Second
IPT	Instance Performance Trust
ISP	Instance Sustainable Performance
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
MAS	Multi-Agent System
MDE	Model-Driven Engineering
ME	Managed Element
MIMD	Multiple Instruction Multiple Data

MOOP	Multi-Objective Optimization Problem
MPI	Message Passing Interface
MPP	Massively Parallel Processor
MQS	Message Queue System
MTC	Many Task Computing
MTRA	Mean Time to Resource Acquisition
NAS	Network-Attached Storage
NC	Network Capacity
NIST	National Institute of Standards and Technology
OCCI	Open Cloud Computing Interface
OD	Outage Duration
OGF	Open Grid Forum
OGSA	Open Grid Service Architecture
OGSI	Open Grid Service Infrastructure
OS	Operating System
OVF	Open Virtualization Format
P2P	Peer-to-peer
PaaS	Platform-as-a-Service
PC	Personal Computer
PLE	Product Line Engineering
PLR	Packet Lost Ratio
PUE	Power Usage Effectiveness
PVM	Parallel Virtual Machine
QoS	Quality of Service
RE	Resource Efficiency
REST	Representational State Transfer
RMI	Java Remote Method Invocation
RPC	Remote Procedure Call
RRT	Resource Release Time
SaaS	Software-as-a-Service
SC	Storage Capacity
SCA	Service Component Architecture
SCORCH	Smart Cloud Optimization for Resource Configuration Handling
SDA	Service Discovery Agent
SDAR	Storage Data Access Ratio
SGE	Sun Grid Engine
SIMD	Single Instruction Multiple Data

SLA	Service-Level Agreement
SNIA	Storage Networking Industry Association
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SOOP	Single-Objective Optimization Problem
SPEC	Standard Performance Evaluation Corporation
SPL	Software Product Line
SPLE	Software Product Line Engineering
SS	Self-Scheduling
SSD	Solid-State Disk
SSO	Single Sign-On
SW	Smith-Waterman
TGI	The Green Index
TM	Transactional Memory
TVA	Typical Virtual Appliance
VA	Virtual Appliance
VI	Virtual Infrastructure
VM	Virtual Machine
VMI	Virtual Machine Image
VMM	Virtual Machine Monitor
W3C	World Wide Web Consortium
WfMS	Workflow Management System
WSDL	Web Service Definition Language
WSLA	Web Service Level Agreement
WSRF	Web Service Resource Framework
WWW	World-Wide Web
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
YAML	YAML Ain't Markup Language

Chapter 1

Introduction

Contents

1.1 Motivation

Cloud computing is a recent paradigm for provisioning of computing infrastructure, platform and/or software. It provides computing resources through a virtualized infrastructure, letting applications, computing power, data storage and network resources to be provisioned, and remotely managed over private networks or over the Internet [154, 307]. Hence, cloud computing enhances collaboration, agility, scalability, and availability to end users and enterprises.

Furthermore, the clouds offer different features, enabling resource sharing (e.g., infrastructure, platform and/or software) among cloud providers and cloud users in a pay-as-you-go model. These features have been used for many objectives such as (a) to decrease the cost of ownership, (b) to increase the capacity of dedicated infrastructures when they run out of resources, (c) to reduce power consumption and/or carbon footprint, and (d) to respond effectively for changes in the demand. However, the access to these features depends on the levels of abstraction (i.e., cloud layer). The levels of abstraction are usually defined as being: infrastructure-as-a-service (IaaS), platform-as-a-service (PaaS), and software-as-a-service (SaaS). While IaaS offers low-level access to the infrastructure, including virtual machines (VMs), storages, and network, PaaS adds a layer above the IaaS infrastructure, offering high-level primitives to help the users on developing native cloud applications. In addition, it also provides services for application deployment, monitoring, and scaling. Finally, SaaS provides the applications, and manages the whole computing environment.

In this context, the users interested in the cloud face the problem of choosing low-level services to execute ordinary applications, thus, being responsible for managing the computing resources (i.e., VMs) or choosing high-level services needing to develop native cloud applications, in order to delegate the management of the computing environment to cloud providers. These two options exist, because the clouds often target Web applications, whereas users' applications are usually batch-oriented performing parameter sweeps. Therefore, deploying and executing an application in the cloud is still a complex task [179, 392]. For instance, to execute an application using the cloud infrastructure, the users must first select a virtual machine (VM), configure all necessary applications, transfer all data, and finally, execute their applications. Learning this process can require days or even weeks, if the users are unfamiliar with system administration, without guarantees of meeting their objectives. Hence, this can represent a barrier to use the cloud infrastructure.

Moreover, with the number of cloud providers growing, the users have the challenge of selecting an appropriate cloud and/or resource (e.g.,VM) to execute their applications. This represent a difficult task because there is a wide range of resources offered by the clouds. Furthermore, these resources are usually suited for different purposes, and they often have multiple constraints. Since clouds can fail, or may have scalability limits, a cloud federation scenario should be considered, increasing the work and the difficulties in using clouds' infrastructures. Some questions that arise in this context are: (a) which resources (VMs) are suitable for executing a given application; (b) how to avoid under and over-provisioning taking into account both resources' characteristics and users' objectives (e.g., performance at lower cost), without being tied to a cloud provider; and (c) how a non-cloud specialist may maximize the usage of the cloud infrastructure without re-engineering their applications to the cloud environment.

Although some efforts have been made to reduce the cloud's complexity, most of them target software developers and are not straightforward for inexperienced users [179]. Besides that, cloud services usually run on large-scale data centers and demand a huge amount of electricity. Nowadays, the electricity cost can be seen as one of the major concerns of data centers, since it is sometimes nonlinear with the capacity of the data centers, and it is also associated with a high amount of carbon emission (CO_2). The projections considering the data center energy-efficiency [142, 201, 202] show that the total amount of electricity consumed by data centers in the next years will be extremely high, and it is like to overtake the airlines industry in terms of carbon emissions.

Additionally, depending on the efficiency of the data center infrastructure, the number of watts that it requires can be from three to thirty times higher than the number of watts needed for computations [330]. And it has a high impact on the total operation costs [31], which can be over 60% of the peak load. Nevertheless, energy-saving schemes that result in too much degradation of the system performance or in violations of service-level agreement (SLA) parameters would eventually cause the users to move to another cloud provider. Thus, there is a need to reach a balance between the energy savings and the costs incurred by these savings in the execution of the applications.

In this context, we advocate the usage of cloud federation to seamlessly distribute the services workload across different clouds, according to some objectives (e.g., to reduce energy consumption) without incurring in too much degradation of performance requirements defined between cloud providers and cloud users. Moreover, we advocate a declarative approach where the users describe their applications and objectives, and submit it to a system that automatically: (a) provisions the resources; (b) sets up the whole computing environment; (c) tries to meet the users' requirements such as performance at reduce cost; and (d) handles failures in a federated cloud scenario.

1.2 Objectives

Following a goal-oriented strategy, this thesis aims to investigate the usage of federated clouds considering different viewpoints. It considers the point of views of the: (a) cloud providers, (b) experienced software developers, (c) ordinary users, and (d) multiple users' profiles (i.e., system administrators, unskilled and skilled cloud users, and software developers). To achieve the main goal, this thesis considers the following four sub-goals.

1. **reducing power consumption**: the cloud services usually execute in big data centers that normally contain a large number of computing nodes. Thus, the cost of running these services may have a major impact on the total operational cost [31] due to the amount of energy demanded by such services. In this scenario, we aim to investigate the use of a cloud federation environment to help cloud providers on reducing power consumption of the services, without having a great impact in quality of service (QoS) requirements.

2. **execution of a huge application at reduced-cost**: most of the clouds provide some resources at low financial costs. These resources normally have limited computing capacity and small amount of RAM memory. Moreover, they are heterogeneous with regard to the cloud layer (i.e., PaaS and IaaS), requiring different strategies to use and to connect them. Thus, we want to investigate if a federated execution using exclusively the cheapest resources of each cloud can achieve an acceptable execution time.

3. **reducing the execution time of cloud-unaware applications**: some of the users applications were designed to execute in a single and dedicated resource with almost predictable performance. Hence, these applications do not fit the cloud model, where resource failures and performance variation are part of its normal operation. However, most of these applications have parts that can be executed in parallel. In other words, these applications comprise parts of independent tasks. In this scenario, we aim to investigate the execution of cloud-unaware applications taking into account the financial cost and trying to reduce their execution time without users' intervention.

4. **automate the tasks of selection and configuration of clouds resources for different kinds of users**: nowadays, the users interested in the clouds face two major problems. One is knowing what are the available resources including their constraints and characteristics. Another is the required skill to select, to configure, and to use these resources taking into account different objectives such as performance and cost. In this context, one of our goal is to investigate how to help the users on

dealing with these problems, requiring minimal users' intervention to configure a single or a federated cloud environment.

1.3 Thesis Statement

In 1992, Smarr and Catlett [326] introduced the concept of metacomputing. Metacomputing refers to the use of distributed computing resources connected by the networks. Thus creating a virtual supercomputer – the metacomputer. In this case, they advocated that the users must be unaware of the metacomputer or even any computer, since the metacomputer has the capacity to obtain whatever computing resources are necessary.

Based on this vision, our thesis statement is that:

Cloud computing is an interesting model for implementing a metacomputer due to its characteristics such as on-demand, pay-per-usage, and elasticity. Moreover, the cloud model focuses on delivering computing services rather than computing devices; i.e., in the cloud, the users are normally unaware of the computing infrastructure. Altogether, the cloud model can increase resource federation and reduce the efforts to democratize the access to high performance computing (HPC) environments at reduced cost.

Besides that, we ask the following research questions in this thesis:

can the cloud federation be used to reduce power consumption of data centers, without incurring into several performance penalties for the users?

can software developers use the clouds to speed up the execution of a native-cloud application at reduced-cost, without being locked to any cloud provider?

can inexperienced users utilize a cloud environment to execute cloud-unaware applications, without having to deal with low-level technical details, having both an acceptable execution time and a financial cost, and without having to change their applications to meet cloud's constraints?

is there a method to support automatic resource selection and configuration on cloud federation, and that offers a level of abstraction suitable for different users' profiles?

1.4 Contributions

To tackle the introduced sub-goals this thesis makes the following five contributions:

1. **Power-Aware Server Consolidation for Federated Clouds**: we propose and evaluate a power and SLA-aware application consolidation solution for cloud federations [215]. To achieve this goal, we designed a multi-agent system (MAS) for server consolidation, taking into account service-level agreement (SLA), power consumption, and carbon footprint. Different for similar solutions available in the literature, in our solution, when a cloud is overloaded its data center needs to negotiate with other

data centers before migrating the workload (i.e., VM). Simulation results show that our approach can reduce up to 46% of the power consumption, while trying to meet performance requirements. Furthermore, we show that federated clouds can provide an adequate solution to deal with power consumption in clouds. In this case, cloud providers can use the computing infrastructure of other clouds according to their objectives. This work was published in [215].

2. **Biological Sequence Comparison at Zero-Cost on a Vertical Public Cloud Federation**: we propose and evaluate an approach to execute a huge bioinformatics application on a vertical cloud federation [214]. This approach has two main components: (i) an architecture that can transparently connect and manage multiple clouds, thus creating a multi-cloud environment and (ii) an implementation of a MapReduce version of the bioinformatics application in this architecture. The architecture and the application were implemented and executed in five public clouds (Amazon EC2, Google App Engine, Heroku, OpenShift, and PiCloud), using only their free-quota. In our tests, we executed an application that did up to 12 million biological comparisons. Experimental results show that (a) our federated approach could reduce the execution time in 22.55% over the best stand-alone cloud execution; (b) we could reduce the execution time from 5 hours and 44 minutes (*SSEARCH* sequential tool) to 13 minutes (our Amazon EC2 execution); and (c) federation can enable the execution of huge applications in clouds at no expense (i.e., using only the free-quota). With this work, it became clear that our architecture could federate real clouds and it could execute a real application. Even though the architecture proposed was very effective, it was application-specific (i.e., Mapreduce application). Moreover, it became clear for us that configuration tasks are complex in a real cloud environment, and they often require advanced computing skills. So, we decided to investigate generic architectures and models that did not impose complex configuration tasks for the users. This work was published in [214].

3. **Excalibur: A User-Centered Cloud Architecture for Executing Parallel Applications**: we propose and evaluate a cloud architecture, called Excalibur. This architecture has three main objectives [217]: (a) provide a platform for high performance computing applications in the cloud for users without cloud skills; (b) dynamically scale the applications without user intervention; and (c) meet the users requirements such as high performance at reduced cost. Excalibur comprises three main components: (i) an architecture that sets up the cloud environment; (ii) an auto-scaling mechanism that tries to reduce the execution time of cloud-unaware applications. In this case, the auto-scaling solution focuses on applications that were developed to be executed sequentially, but that have parts that can be executed in parallel; (iii) a domain specific language (DSL) that allows the users to describe the dependencies between the applications based on the structure of their data (i.e., input and output). We executed a complex genomics cloud-unaware application in our architecture, which was deployed on Amazon EC2. The experiments showed that the proposed architecture could dynamically scale this application up to 11 instances, reducing the execution time by 63% and the cost by 84% when compared to the execution in a configuration specified by the users. In this case, the execution time was reduced from 8 hours and 41 minutes to 3 hours and 4 minutes; and the cost was

reduced from 78 USD to 14 USD. With this work, the advantages of auto-scaling in clouds became clear to us. Furthermore, we showed that it was possible to execute a complex cloud-unaware application in the cloud. This work was published in [217].

4. **Resource Selection Using Automated Feature-Based Configuration Management in Federated Clouds**: we propose and evaluate a model to handle the variabilities of IaaS clouds. The model uses extended feature model (EFM) with attributes to describe the resources and their characteristics and to select appropriate virtual machine based on users' objectives. We implemented the model in a solver (i.e., Choco [178]) considering the configurations of two different clouds (Amazon EC2 and Google Compute Engine (GCE)). Experimental results showed that using the proposed model, the users can get an optimal configuration with regard to their objectives without needing to know the constraints and variabilities of each cloud. Moreover, our model enabled application deployment and reconfiguration at runtime in a federated cloud scenario without requiring the usage of virtual machine image (VMI).

5. **Dohko: An Autonomic and Goal-Oriented System for Federated Clouds**: we propose and evaluate an autonomic and goal-oriented system for federated clouds. Our system implements the autonomic properties: self-configuration, self-healing, and context-awareness. Using a declarative strategy, in our system, the users specify their applications and requirements (e.g., number of CPU cores, maximal financial cost per hour, among others), and the system automatically (a) selects the resources (i.e., VMs) that meet the constraints using the model proposed in contribution 4; (b) configures and installs the applications in the clouds; (c) handles resource failures; and (d) executes the applications. We executed a genomics application (i.e., SSEARCH, September 2014) to compare up to 24 biological sequences with the UniProtKB/Swiss-Prot (September 2014) in two different cloud providers (i.e., Amazon EC2 and GCE) and considering different scenarios (e.g., standalone (i.e., single cloud) and multiple clouds). Experimental results show that our system could transparently connect different clouds and configure the whole execution environment, requiring minimal users intervention. Moreover, by employing a hierarchical management organization (i.e., a hierarchical P2P overlay), our system was able to handle failures and to organize the nodes in a way that reduced inter-cloud communication.

1.5 Publications

1. Alessandro Ferreira Leite, Claude Tadonki, Christine Eisenbeis, Tainá Raiol, Maria Emilia M. T. Walter, and Alba Cristina Magalhães Alves de Melo. *Excalibur: An autonomic cloud architecture for executing parallel applications.* In 4th International Workshop on Cloud Data and Platforms, pages 2:1–2:6, Amsterdam, Netherlands, 2014

2. Alessandro Ferreira Leite and Alba Cristina Magalhães Alves de Melo. *Executing a biological sequence comparison application on a federated cloud environment.* In

19th International Conference on High Performance Computing (HiPC), pages 1-9, Bangalore, India, 2012

3. Alessandro Ferreira Leite and Alba Cristina Magalhães Alves de Melo. Energy-aware multi-agent server consolidation in federated clouds. In Mazin Yousif and Lutz Schubert, editors, Cloud Computing, volume 112 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 72–81. 2013

4. Alessandro Ferreira Leite, Hammurabi Chagas Mendes, Li Weigang, Alba Cristina Magalhães Alves de Melo, and Azzedine Boukerche. *An architecture for P2P bag-of-tasks execution with multiple task allocation policies in desktop grids.* Cluster Computing, 15(4), pages 351-361, 2012

1.6 Thesis Outline

This thesis is organized in two parts: background and contributions.

In the first part, we present the concepts and recent developments in the domain of large-scale distributed systems. In this case, it comprises the following chapters:

chapter 2: we provide an historical perspective of concepts, mechanisms and tools that are landmarks in the evolution of large-scale distributed systems. Then, we present the most representative types of these systems: clusters, grids, P2P systems, and clouds.

chapter 3: we discuss the practical aspects related to cloud computing, such as virtualization, service-level agreement (SLA), MapReduce, and cloud computing architectures.

chapter 4: we describe autonomic computing. First, we present the definition of autonomic systems, followed by the autonomic properties. Then, we present the concepts related to the architecture of autonomic systems. Finally, some autonomic systems for large-scale distributed systems are presented and compared.

chapter 5: we present the concept of energy-aware computing, providing information about green data centers followed by a discussion about green performance indicators.

In the second part, we present the contributions of this thesis, and it is organized as follows:

chapter 6: we present the first contribution of this thesis: a server consolidation approach to reduce power consumption in cloud federations. First, we present the proposed multi-agent system (MAS) server consolidation strategy for federated clouds. Then, the experimental results are discussed followed by the related work in this area. Finally, we present final considerations.

chapter 7: in this chapter, we describe the seconds contribution of this thesis: our approach to execute the Smith-Waterman (SW) algorithm in a cloud federation with zero-cost. First, we provide a brief introduction for biological sequence comparison followed by a description of the Smith-Waterman algorithm. Next, we present the proposed architecture and the experimental results realized in a public cloud federation scenario. Finally, we discuss some of the related works that have executed the SW algorithm in different platforms followed by final considerations.

chapter 8: in this chapter, we describe the third contribution of this thesis: a cloud architecture to help the users on reducing the execution time of cloud-unaware applications. First, we present our cloud architecture. After, experimental results are discussed followed by a discussion of similar cloud architecture available in the literature. Finally, we present final considerations.

chapter 9: this chapter presents the fourth contribution of this thesis: a model to handle the variabilities of IaaS clouds. First, it presents the motivation and challenges addressed by our model followed by an overview of multi-objective optimization problem (MOOP) and feature modeling. Next, the proposed model is presented. After, it describes the experimental results followed by the related works. Finally, it presents final considerations.

chapter 10: in this chapter, we present the fifth contribution of this thesis: an autonomic cloud architecture. First, we present the proposed architecture and its main components, followed by a description of its autonomic properties. Then, experimental results are discussed. Next, a comparative view of some important features of cloud architectures is presented. Finally, we present final considerations.

chapter 11: in this chapter, we summarize the overall manuscript, and we present the limitations and future directions.

Part I

Background

Chapter 2

Large-Scale Distributed Systems

Contents

Over the years, we have observed a considerable increase in the demand for powerful computing infrastructures. This demand has been satisfied by aggregating resources connected through a network, forming large-scale distributed systems. These systems normally appear to users as a single system or computer [88, 341]. One example is the Internet, where the users use different services to communicate and to share information, without needing to know about its computing infrastructure.

Large-scale distributed systems can be defined as systems that coordinate a big number of geographically distributed and mostly heterogeneous resources to deliver scalable services without having a centralized control. Scalability is an important characteristic of large-scale distributed systems since it guarantees that even if the number of users, nodes or the system workload increases, these systems can still deliver their services without noticeable effect on performance or on administrative complexity [192]. Examples of such systems are the SETI@home [370], the LHC Computing Grid [33], among others.

In this context, this chapter presents the evolution of computing systems from the 1960s until today. Then, the most common large-scale systems are discussed: clusters, grids, P2P systems, and clouds.

2.1 Evolution

Nowadays, we observe an astonishing level of complexity, interoperability, reliability, and scalability of large-scale distributed systems. This is due to concepts, models, and techniques developed in the last 50 years in many research domains, including computer architecture, networking, and parallel/distributed computing. In this section, we present a general landscape of events and the main landmarks that contributed to our current development state. It must be noted that this is not intended to be an exhaustive list. In order to give a historical perspective, this section is organized in subsections that describe the main developments in each decade.

2.1.1 The 1960s

The 1960s was a period of innovative ideas that could not become reality because of several technological barriers. It was also the decade of the first developments in the areas of supercomputing and networking.

In the 1960s, some researchers conceived the idea of the computer as a utility. They envisioned a world where the computer was not an expensive machine restricted to some organizations, but a public utility as the telephone system. This idea was exposed by John McCarthy in 1961 at a talk at the MIT Centennial [132]. In 1962, J. C. L. Licklider proposed the Galactic Network concept [226], where a set of globally interconnected computers could be used by anyone to quickly access data and programs. Even though these ideas became popular in the 1960s, many technological barriers were encountered that prevented their implementation at that time.

Still in the 1960s, the CDC 6000 series was designed by Seymour Cray as the first supercomputer, composed of multiple functional units and one or two CPUs. This was a multiple instruction multiple data (MIMD) machine, according to Flynn's categorization [118]. In the late 1960s, the CDC 8600 had 4 CPUs, with a shared memory design.

In 1962, the first wide-area network experiment was made, connecting a computer in Massachusetts to a computer in California, using telephone lines. In 1969, the ARPANET was created, aiming to connect computing nodes in several Universities in the US, making use of the recently developed packet switching theory.

In 1965, IBM announced the System 370/67, introducing the usage of a software layer called virtual machine monitor (VMM) that enabled to share the same computer among several operating systems and computing environments [138, 146].

The concepts involved in the client/server model were first proposed in 1969, where the server was named server-host and the client was named using-host. In this early model, both client and server were physical machines.

2.1.2 The 1970s

The 1970s can be seen as the decade where some ideas proposed in the 1960s started to be implemented. One of the main landmarks of this decade is the Internet.

In the early 1970s, Seymour Cray left CDC to create his own company, using a different approach to build supercomputers. The idea was to exploit the single instruction multiple data (SIMD) categorization [118], named vector processing at that time. It replicated the execution units, instead of the whole CPU. The Cray-1 Machine, released in 1976, could operate by blocks of 64 operands (SIMD capability), attaining 250 MFlops (millions of floating-point operations per second).

Still in the 1970s, computers started to be connected by networks, using the ARPANET. In 1972, a large demonstration of the ARPANET took place, using electronic mails as the main application. In addition, the concept of open-architecture network was proposed, where individual networks might be separately designed and then connected by a standard protocol. A packet radio program that used the open-architecture concept was called Internetting at DARPA. In 1973, the first Ethernet network was installed at the Xerox Palo Alto Research Center. Telnet (remote login) and ftp (file transfer) protocols were proposed in 1972 and 1973, respectively. In 1974, Cerf and Kahn published their paper presenting the TCP protocol [72]; and the term Internet was coined to describe a global TCP/IP network.

The combination of the time-sharing vision and the decentralization of computer infrastructures led to the concept of distributed systems in the late 1970s. A distributed system can be defined as a collection of autonomous computers connected by a network that appears to its users as a single system. In this system, computers communicate with each other using messages that are sent over the network, and this communication process is hidden from the users [88, 134, 341]. Transparency is a very important point in this definition. It means that users and applications should interact with a distributed system in the same way as they interact with a standalone system. Other important characteristics are scalability, availability and interoperability which, respectively, enable distributed systems to be relatively easy to expand or scale; to be continuously available, even though some parts are unavailable; and to establish communication among different hardware and software platforms.

2.1.3 The 1980s

The 1980s observed a rapid development in supercomputing, networking and distributed systems. The World-Wide Web (WWW) was proposed in this decade.

In the 1980s, Cray continued to release supercomputers based on vector processing and, in 1988, the first supercomputer to attain a sustained rate of 1 gigaflop was a Cray-YMP8, composed of 8 CPUs, capable of operating simultaneously on 64 operands each. Still in the 1980s, several companies focused on shared memory MIMD supercomputers. The company Thinking Machines built the CM-2 (Connection Machine) hypercube supercomputer, one of the first so-called massively parallel processors (MPPs), composed of 65,536 (1-bit check) processing elements with local memory. In 1989, the CM-2 with 65,536 processing elements attained 2.5 gigaflops.

In 1983, the TCP/IP protocol was adopted by the ARPANET, replacing the previous NCP protocol. In 1985, the TCP/IP protocol was used by several networks, including the USENET, BITNET and NSFNET. In 1989, Berners-Lee [39, 40] proposed a new protocol based on hypertext. This protocol would become the World-Wide Web (WWW) in 1991.

With the advent of personal computers (PCs) and workstations, the client-server model gained popularity in the late 1980s, being mainly used for file sharing. Still in the 1980s, important concepts in distributed systems such as event ordering, logical clocks, global states, Byzantine faults and leader election were proposed [88].

In the 1980s, it became clear that a distributed system must comprise different providers and that it should be independent from the communication technology. Initially, this was a difficult objective to achieve since the communication in a distributed system was mostly implemented using low-level socket primitives. Socket programming is complex and requires a deep understanding of the underlying network protocols. To bypass these difficulties, the remote procedure call (RPC) was proposed in 1983 [47], enabling functions that belong to the same program to be performed by remote computers, as if they were running locally. This represented a great advancement and RPC was the base for the CORBA and Web services.

2.1.4 The 1990s

In the 1990s, great technological advances and intelligent design choices made it possible to break the teraflop performance barrier. It was also the decade of the widespread adoption of the Internet. Metacomputing, grid computing and cloud computing were proposed in the 1990s. At this decade, the researchers also began to consider the use of virtual machines to overcome some limitations of the x86 architecture and operating systems [301].

The idea of cluster computing became very popular in the 1990s. Clusters consisted of commodity components connected in order to build a supercomputer. In 1993, the Beowulf project was able to connect 8 PCs (DX4 processors) with the 10Mbit Ethernet, using the Linux operating system. The number of PCs connected grew quickly and the 10Mbit Ethernet was replaced by Fast Ethernet. One of the reasons for the success of clusters was the development of programming environments such as parallel virtual machine (PVM) and message passing interface (MPI). In 1997, clusters were included among the fastest 500 machines at the Top 500 list (top500.org), with a sustained performance of 10 gigaflops.

Still in the 1990s, research on supercomputing continued and, in 1997, the ASCI Red machine, with 4,510 MIMD computing nodes was the first supercomputer to attain 1 teraflop of sustained performance.

In 1992, more than a million computers were connected to the Internet. In 1993, the first graphic browser, called Mosaic, was introduced [254]. The World Wide Web Consortium (W3C) was created in 1994 to promote and to develop standards for the Web.

Originated in 1991, the Common Object Request Broker Architecture (CORBA) was one of the first attempts to create a standard for distributed objects management in distributed systems [88]. However, due to some ambiguities in its specification, its adoption was slow and its implementation complex. The ambiguities were solved in the second version of Common Object Request Broker Architecture (CORBA) in 1998, however the advent of other technologies as the Java Remote Method Invocation (RMI) and the Extensible Markup Language (XML) led to the decline of CORBA.

The Distributed Computing Object Model (DCOM) came out in 1993. It was restricted to the Windows platform and, beyond that, such as CORBA, the usage of Distributed Computing Object Model (DCOM) over the Internet imposed some administrative challenges, as it required the opening of some doors in the firewall, since both CORBA and DCOM did not use the HTTP protocol.

In 1994, the Representational State Transfer (REST) was proposed as a model for communicating Web concepts while developing the HTTP specification. REST is an architecture style for Web applications that aims to minimize latency and network communication, and to maximize scalability and independence of applications [114]. In practice, REST provides a semantics interface based on the actions (e.g., GET, POST, PUT, DELETE) of the HTTP protocol rather than on arbitrary or application-specific interfaces to manipulate resources only exchanging the representations. Moreover, REST interactions are stateless, which decouple the meaning of a message from the state of a conversation. This architecture style has influenced many Web standards, and nowadays it is widely adopted by the enterprises.

The global degree of maturity achieved by the Internet in the 1990s made possible the appearance of several advanced distributed computing paradigms. In 1992, a paradigm called Metacomputing [326] was proposed. Its main idea was to create a supercomputer by connecting computers spread over the world in a transparent way via a network environment. Later, in 1995, the term grid computing was conceived to denote a new infrastructure of distributed computing that allows consumers to obtain resources on-demand in an advanced scope [125]. The grid paradigm can be seen as an evolution of metacomputing combined with utility computing, where not only processing power is shared but also other resources such as databases, specific equipments and softwares, among others. Inspired by the electrical power grid because of its pervasiveness, ease of use and reliability, the motivation of grid computing was initially driven by large-scale resource sharing and data-intensive scientific applications that require more resources than they provided by a single computer [125]. In 1995, the Information-Wide-Area-Year (I-WAY) project [124] was one of the first projects to demonstrate the power of distributed supercomputing. Furthermore, the Globus Project (toolkit.globus.org/alliance/news/prGAannounce.html) was established to develop standards for grid's middlewares [124].

Simultaneously with the growing research interest in grid systems, peer-to-peer (P2P) systems have evolved. A P2P system can be seen as a system where the nodes (peers) organize themselves into different topologies and operate without any central control [14]. One of the first successful P2P systems was the Napster file sharing system, proposed in 1999. Another type of P2P system, that harnesses computer power to solve complex problems, was also proposed in the 1990s. In 1999, the SETI@home project started, and it could connect millions of computers through the Internet [12].

In 1997, the term cloud computing was used by Chellappa in a talk about a new computing paradigm [74]. Cloud computing was indeed proposed in the late 1990s aiming to shift the location of the computing infrastructure, platform and/or software systems to the network in order to reduce costs associated with resource management (hardware and software) [154]. In cloud computing, data and programs are usually stored and executed in huge data centers, with hundreds or thousands of machines.

In 1998, the VMware company was founded, introducing a system that virtualizes the x86 architecture providing full isolation of a guest operating system [54]. Virtualization would become a fundamental concept in cloud computing.

In 1999, the salesforce.com system was one of the first systems to provide a software-as-a-service (SaaS) cloud solution to manage sales activities. Also, in 1999, an extension for the HTTP protocol was proposed, and it became the HTTP/1.1 [115]. Such extension provided a model for the WWW, defining how it should work; and it became the foundation for the Web services architecture.

2.1.5 2000-2014

The first years of the 21st century observed extraordinary advancements in supercomputing and cloud computing.

From 2000 to 2014, research on supercomputing continued obtaining astonishing results. Supercomputing infrastructures are nowadays normally composed of clusters, where the nodes contain general-purpose processors combined with accelerators, and high performance networks. In 2008, the IBM RoadRunner supercomputer was the first to achieve 1 petaflop of sustained performance, with 6,912 Opteron processors combined with 12,960 PowerXCell accelerators. In 2011, the K Computer attained 10 petaflops, with 705,024 Sparc64 cores. In the latest supercomputer Top500 list, released in June 2014, the fastest supercomputer is the Tianhe-2, attaining 33.8 petaflops with 3,120,000 cores, in a hybrid design containing general-purpose Intel Xeon processors and Intel Xeon Phi accelerators.

The interoperability issues presented by CORBA and DCOM were mainly solved through the specification of Web services in 2000. The term Web service was defined by the W3C in 2004, describing its interoperability and naming the Web Service Definition Language (WSDL) format and the SOAP (Simple Object Access Protocol) protocol. Web services are technology-independent and they use HTTP as the communication layer [51]. In this case, services are described using WSDL, enabling the interoperability between distinct programming languages. Nowadays, Web services are the predominant model in distributed computing.

From 2000 to now, cloud computing received a lot of attention from the industry and academia since it aims to reduce the costs of managing hardware/software infrastructures. Clouds are similar to grids since they also employ distributed resources to achieve the computing objectives. However, in grid computing, the resources, which can be provided as utility, are allocated to applications for a period of time, paying for the whole period. Instead, cloud computing normally uses on-demand resource provisioning, eliminating over-provisioning, since the users pay for what they use [388].

The term autonomic computing was proposed in 2001 [162], referring to self-management systems. These systems could adapt themselves for changes in the computing environment/infrastructure.

MapReduce [98] is a programming model proposed in 2004 as a solution to be used by Google for large-scale data processing, and lately available for the general public through the open source Apache Hadoop (hadoop.apache.org), in 2005. Hadoop provides a parallel programming model, a distributed file system, a collection of tools for managing large distributed systems, and coordination services for distributed applications. With Apache Hadoop, the developers often do not need to be aware of data distribution and failures as they are seamless delivered by Hadoop.

In 2006, Amazon created its Elastic Compute Cloud (EC2) as a commercial cloud system, using the pay-per-use model.

Clouds are having success owing to the fact that the underlying infrastructure is completely transparent to the users. Beyond that, clouds exhibit good scalability, allowing users to run complex applications in the cloud infrastructure. In 2014, the Amazon Web Services (AWS) is the largest cloud hosting company in the world. It provides services in many geographic regions, including Australia, Brazil, Ireland, Japan, Singapore, and the US. AWS can also be used as a supercomputer. In the supercomputer Top500 list, released in June 2014, it appears in the 76th position, attaining 593.5 Teraflops with 26,496 cores.

2.1.6 Timeline

Figure 2.1 presents the main landmarks of each decade. As can be seen, the advances in computing in five decades are astonishing. For instance, in the beginning of the 1970s, the supercomputers could perform 250 millions of floating point operations per second (MFlops). In 2014, the fastest supercomputer performs 33.8 quadrillions of operations per second (PFlops).

These advances are also owing to the improvements in networking that started as a way to decrease the cost to build and to operate the computers as well as to increase data sharing and collaboration.

It is also important to notice that some concepts were proposed in a visionary way. One example is the concept of utility computing, proposed in the 1960s but incorporated to grids and clouds only in the 1990s.

Nowadays, due to the utility model (i.e., cloud computing), the users can provision a cluster with reasonable performance paying comparatively few dollars per cores per hour. For instance, a cluster with $30,000$ cores distributed across different Amazon data centers costs \$1279/hour (0.043 USD per cores by hour) [91]. In this model, the developers with innovative ideas for new Internet services no longer require large capital outlays in hardware to deploy their services or the human expense to operate it. Moreover, companies with large batch-oriented tasks can get results as quickly as their programs can scale, since using 1,000 servers for one hour costs no more than using one server for 1,000 hours. This elasticity of resources, without paying a premium for large scale, is unprecedented in the history of computer science [20].

The success of cloud computing is also due to the advances in virtualization technologies, which hides the physical infrastructure from the users and often leads to increase resource utilization, and in programming models as MapReduce [98] and its public available implementation Apache Hadoop.

In the context of cloud computing, there are still many challenges to be addressed. First, cloud providers should agree and support cloud standards to enable full portability and interoperability of data and applications among the clouds. Second, based on standardization, the clouds should be combined to increase performance and/or to decrease operational costs. Third, similar to the electrical power grid, that often providers electricity with a well known variation, the clouds should deliver services with near constant performance, defined according to the resource type. Nowadays, to achieve this, the users have to distribute their applications across multiple clouds [269, 376] and to optimize different parameters manually. Finally, cloud tools should be created to allow the users without any cloud skill to maximize the cost/benefit ratio of the cloud.

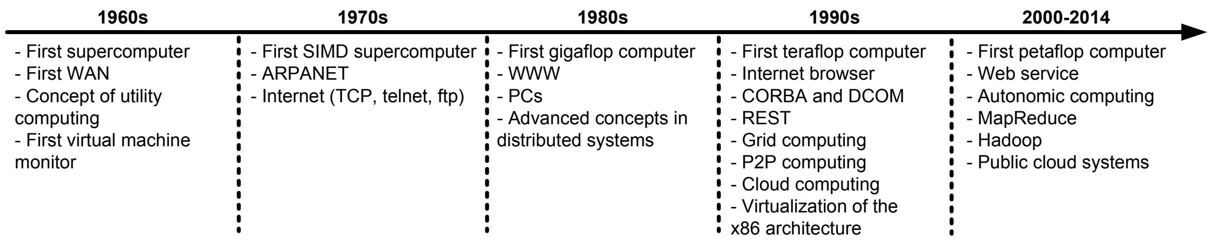

1960s
- First supercomputer
- First WAN
- Concept of utility computing
- First virtual machine monitor
1970s
- First SIMD supercomputer
- ARPANET
- Internet (TCP, telnet, ftp)
1980s
- First gigaflop computer
- WWW
- PCs
- Advanced concepts in distributed systems
1990s
- First teraflop computer
- Internet browser
- CORBA and DCOM
- REST
- Grid computing
- P2P computing
- Cloud computing
- Virtualization of the x86 architecture
2000-2014
- First petaflop computer
- Web service
- Autonomic computing
- MapReduce
- Hadoop
- Public cloud systems

Figure 2.1: Computing landmarks in five decades

2.2 Cluster Computing

A cluster is set of dedicated computing nodes connected by a dedicated and high-speed local-area network. Cluster computing systems are often built to deliver high-performance capabilities with a good price/performance ratio, running compute intensive applications [341].

In general, clusters use a master/slave architecture, where the master node is responsible for providing an interface to the users, for distributing the tasks to the slaves, and for managing the execution of the tasks. Slave nodes, on the other hand, are responsible for executing the assigned tasks. Furthermore, the nodes in a cluster are often homogeneous with the same operating system and they belong to a single organization.

Figure 2.2 shows the architecture of the Tompouce cluster, which is a high-performance medium-scale cluster maintained by INRIA Saclay (www.inria.fr/en/centre/saclay). It has a master node and twenty compute nodes connected by two InfiniBand and one gigabit Ethernet networks. In Tompouce, each node has two six-core Intel Xeon processors, thereby 240 cores.

Clusters can be categorized in three classes [55]: high-availability (HA), load-balancing, and compute clusters.

A high-availability (HA) cluster aims to improve the availability of the services provided by the cluster and to avoid single-points of failure. In this case, it operates employing

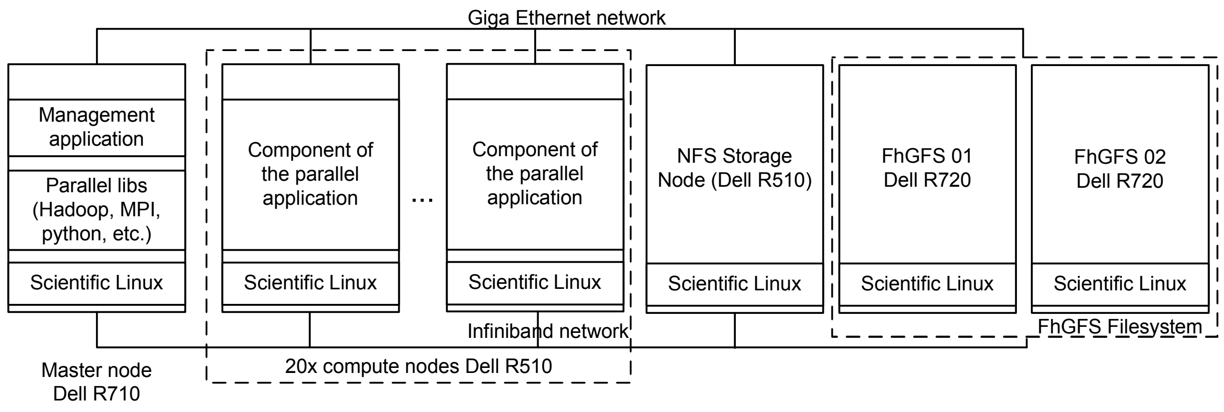
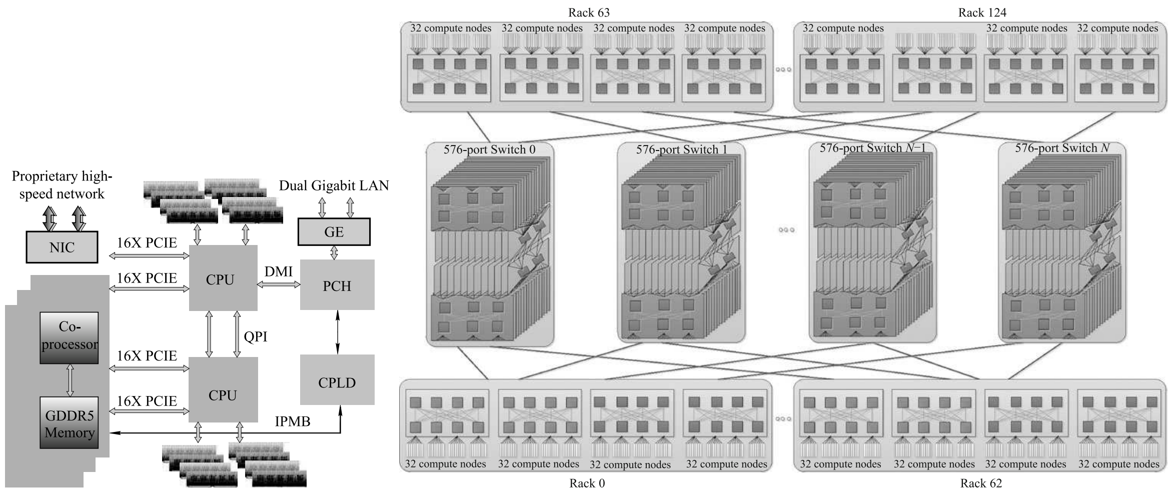


Figure 2.2: Tompouce: an example of a medium-scale cluster

redundant nodes, which are used when a system component fails. In load-balancing clusters, multiple computers are connected to share the computational workload to improve the overall performance of the system. Compute clusters, on the other hand, are designed to provide supercomputing capabilities, often executing message passing interface (MPI) applications.

Nowadays, clusters are being built with an astonishing number of cores, and they represent up to 84% of the fastest machines listed in the Top500 list, released in June 2014. In this list, the number one (Tianhe-2, or TH-2) is a cluster of 16,000 nodes, each one comprising two Intel Ivy Bridge processors and three Xeon Phi coprocessor boards (Figure 2.3(a)), counting 3,120,000 cores [225]. These nodes are connected by a proprietary interconnection called the TH Express-2 interconnect network. The TH Express-2 is an optoelectronics hybrid transport technology organized as a fat tree network with 13,576 ports at the top level (Figure 2.3(b)), which allows it to achieve high throughput and low latency.



(a) Node architecture (b) Tianhe-2 organization

Figure 2.3: The Tianhe-2 compute node architecture and its network topology [225]

2.3 Grid Computing

Just as electrical power grids can acquire power from multiple power generators and deliver the power needed by the consumers, the key emphasis of grid computing is to enable resource sharing, where resources usually belong to different organizations, forming a pool of shared resources that can be delivered in a transparent way to users. The goal of grid computing is to enable resource aggregation in a dynamic environment abstracting the complexity of computing resources. For this, grid computing relies on the usage of standard network protocols and middlewares to mediate the access to heterogeneous resources.

Due to the overlap between the grid's characteristics and other distributed architectures such as clusters and supercomputers, Ian Foster and colleagues [126] defined a three point checklist for determining whether a system is a grid or not. For them, a grid is a system that: (i) coordinates resources that are not subject to centralized control; (ii) uses standard, open, general-purpose protocols and interfaces; and (iii) delivers non-trivial quality of service.

The first criterion emphasizes that a grid should integrate computing resources under different control domains. This requires the ability to negotiate resource sharing agreements among the virtual organizations through direct access to the resources either with collaborative resource sharing or through negotiated resource brokering strategies. The second criterion states the need of standard protocols to make the grid operations feasible. Finally, a grid environment should support different quality of service requirements to meet the users needs. Figure 2.4 shows this grid computing model.

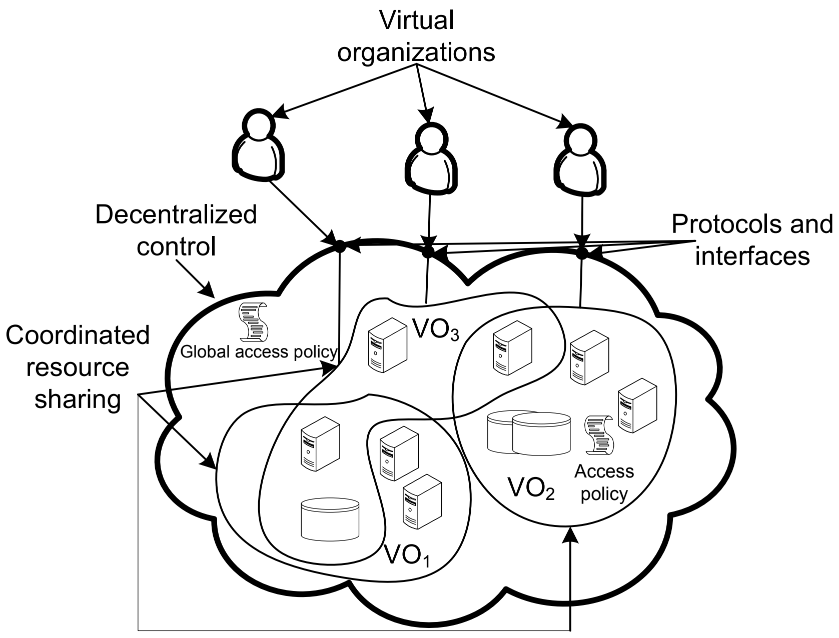


Figure 2.4: Foster's grid computing model [126]

In practice, grid systems employ a hierarchical administration with rules governing resources' availability. This availability can significantly vary over time, and the applications request resources by specifying a set of attributes such as the number/type of CPU and the amount of memory [349].

In grid computing, virtual organizations are either physically distributed institutions or logically distributed users perceived as a single unit that share a common objective. In practice, each virtual organization manages its own resources, including allocation policies, authorization, and access control.

Since the grid aggregates resources managed by virtual organizations, it makes possible the solution of new types of problems, which in a single organization would take considerable time to accomplish. In other words, a grid focuses on distributed resource integration across multiple administrative domains, abstracting the resource heterogeneity and giving to users a powerful computational environment.

2.3.1 Architecture

Many different architectural models have been proposed for grid. The first model that became a standard was known as the hourglass model, depicted in figure 2.5. This model has five different layers. The fabric layer providers access to computing resources mediated by grid protocols. The connectivity layer defines the core protocols for authentication and inter-node communication. The collective layer is responsible for interacting with different services such as brokering services, directory services, authorization services, and even MPI-based programming systems. The application layer comprises the users applications developed using the grid components. This layer supports the users to execute their applications in the grid.

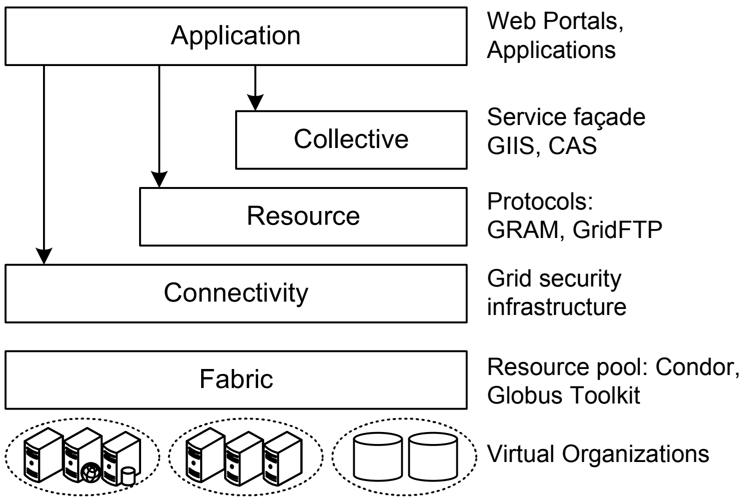


Figure 2.5: Hourglass grid architecture [126]

In 2002, the Open Grid Service Architecture (OGSA) was proposed by the Global Grid Forum (GGF) as a Web service-based standard for grid systems, creating the concept of grid services. The first specification of OGSA was called OGSA-OGSI (Open Grid Service Infrastructure) and was implemented by the Globus Toolkit 3 [308]. Globus quickly became the standard for grid middleware.

The OGSA-OGSI architecture had some problems because Web services are supposed to be stateless and grid services are stateful. To achieve the convergence between Web services and grid services, the OGSA-WSRF (Web Service Resource Framework) was proposed and it was implemented by the Globus Toolkit 4 [122]. In 2014, the most recent version of globus is the Globus Toolkit 5, which has added mainly incremental updates to Globus Toolkit 4. The figures 2.6 and 2.7 show the difference between Globus Toolkit 3 (OGSA-OGSI) and Globus Toolkit 4 (OGSA-WSRF).

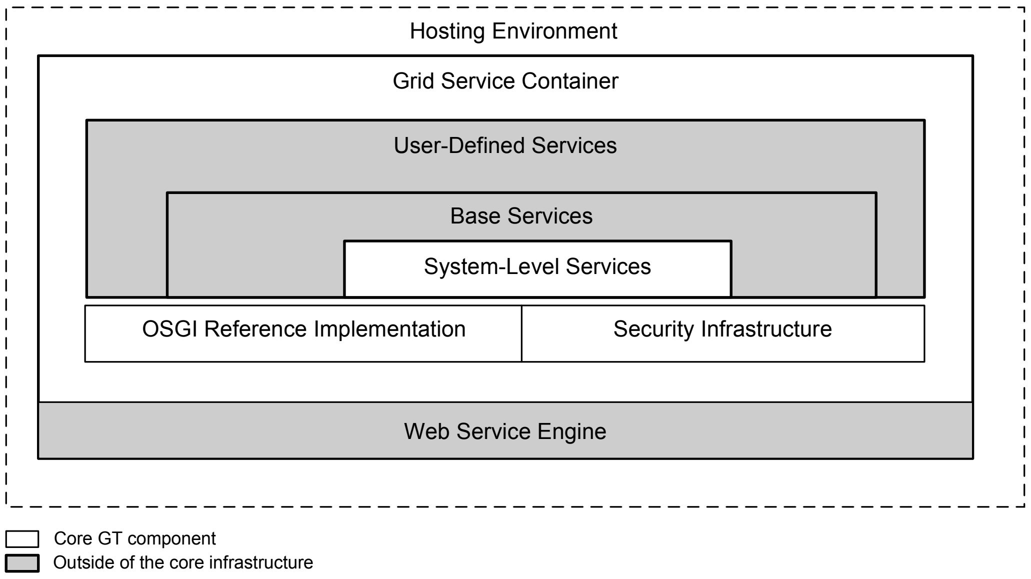


Figure 2.6: Globus Toolkit 3 architecture [308]

2.4 Peer-to-peer

Peer-to-peer (P2P) systems and grids are distributed computing systems driven by the need of resource sharing, but targeting different communities. While the main target of grid computing is the scientific community, the target of P2P systems is the general user. Grids are usually used for complex scientific applications, which are time critical and require some quality of service (QoS) guarantees. On the other hand, P2P applications are normally content-sharing, IP communication, and cycle stealing available at the edges of the Internet and without QoS guarantees [14].

Peer-to-peer systems are defined as distributed systems where the nodes (peers) autonomously organize the system topology and respond to external usage in a decentralized fashion to share resources without any central control [14].

Although there are differences between grid and P2P systems, their advantages can be combined to increase scalability or to decrease maintenance costs. For instance, grid resource discovery can be implemented with P2P structures due to their scalability and self-management properties.

In P2P systems, there is no notion of clients or servers since all nodes are equal, and thus can be both client and server. In another words, in a P2P system, the role of the peers is symmetric. This symmetric role property helps, in many cases, to reduce communication overhead or to avoid bottlenecks.

Although in the strictest definition P2P systems are totally distributed, sometimes nodes with specialized functions can be used for specific tasks or communication management (e.g.,

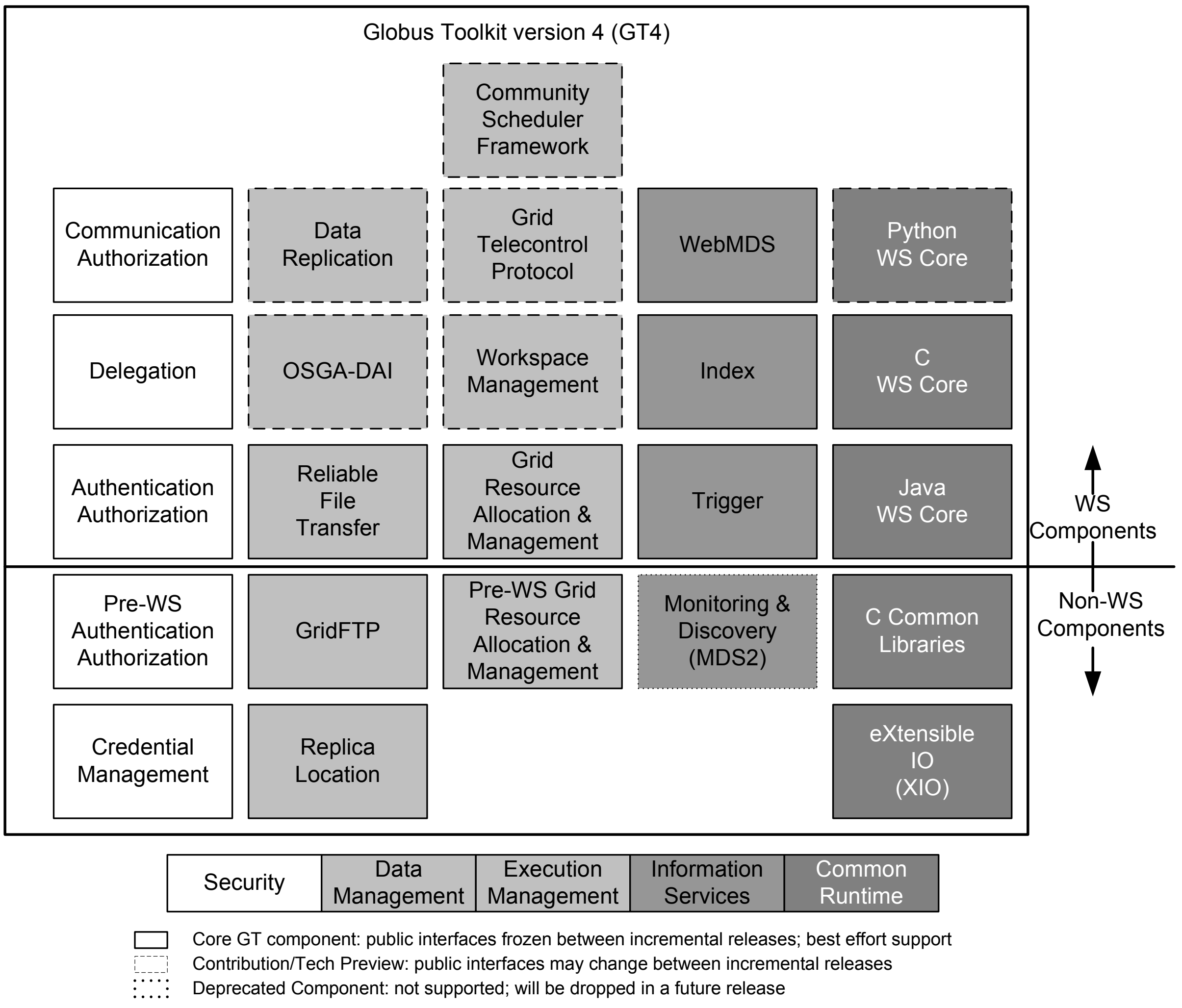

Globus Toolkit version 4 (GT4)
Community Scheduler Framework
Communication Authorization
Data Replication
Grid Telecontrol Protocol
WebMDS
Python WS Core
Delegation
OSGA-DAI
Workspace Management
Index
C WS Core
Authentication Authorization
Reliable File Transfer
Grid Resource Allocation & Management
Trigger
Java WS Core
WS Components
Non-WS Components
Pre-WS Authentication Authorization
GridFTP
Pre-WS Grid Resource Allocation & Management
Monitoring & Discovery (MDS2)
C Common Libraries
Credential Management
Replica Location
eXtensible IO (XIO)
Security
Data Management
Execution Management
Information Services
Common Runtime
Core GT component: public interfaces frozen between incremental releases; best effort support
Contribution/Tech Preview: public interfaces may change between incremental releases
Deprecated Component: not supported; will be dropped in a future release

Figure 2.7: Globus Toolkit 4 architecture [122]

system bootstrapping or to obtain a global encryption key) since the system does not rely on one or more global centralized nodes for its basic operation. However, it must be noted that a system that uses a node to maintain a global index and depends on it to operate (e.g., searching through the index) cannot be defined as a P2P system.

P2P systems have been used for different purposes and different kinds of applications, as shown in figure 2.8. In P2P systems, every node potentially contributes to the system. In this case, when the number of nodes increases, the capacity of the system also increases, because of additional resources brought by the new nodes. Even though there may exist differences in the resources provided by each node, such nodes have the same functional capacity and responsibility in the system. In addition, P2P systems eliminate the single point of failure by employing a decentralized architecture [14]. Furthermore, P2P systems require minimum management, since they are often self-organized systems and they do not depend on dedicated servers to manage the system. However, unlike centralized systems, in which the decision point to access the resources is concentrated in a single node, P2P systems provide access to resources located across the network. This requires efficient algorithms to distribute and to locate data in the presence of a highly transient population of nodes.

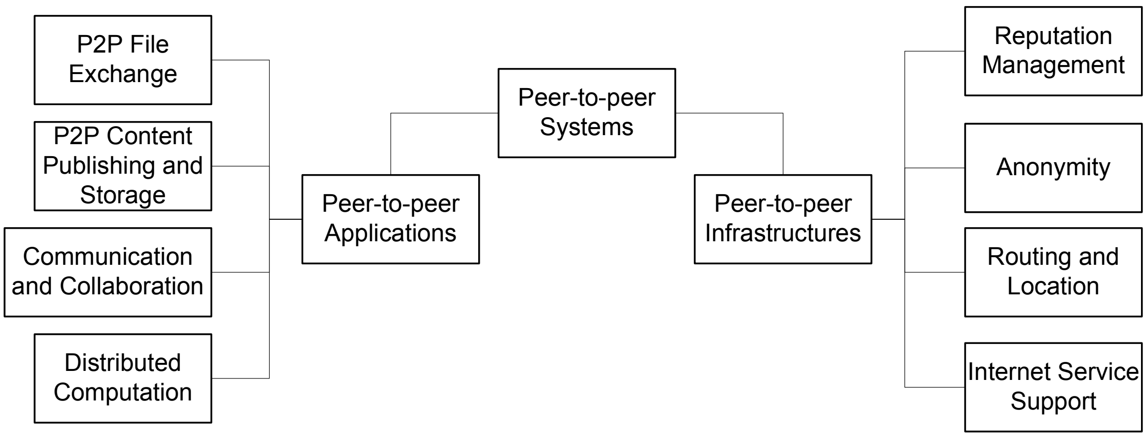


Figure 2.8: Different use of P2P systems [14]

2.4.1 Architecture

A P2P architecture relies on a network of connected nodes built on top of an underlying network, known as overlay network [347] (Figure 2.9). The underlying network refers to the physical network used by the nodes to route their communication packets. The overlay network is responsible for providing P2P services (e.g., data storage) for the applications built on top of it, and it is independent of the physical network. It also ensures that any node can access any object by routing requests through the nodes, exploiting knowledge at each of them to locate an object. According to Touch [347], the tasks of an overlay network are: (i) to route requests to objects; (ii) to insert and to remove objects; (iii) to place the nodes in the network; and (iv) to maintain the network. In this context, the design of an overlay network is crucial for the operation of the system as it may affect its scalability, performance, fault-tolerance, self-management, and security. A P2P application uses the overlay network to provide services for the users such as content distribution, instant messaging, among others.

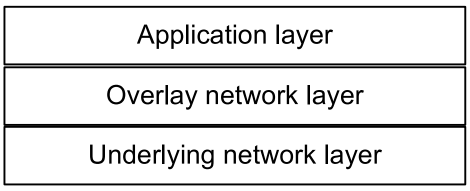


Figure 2.9: Generic P2P architecture [14]

A P2P overlay can be classified as: unstructured, structured, hybrid, and hierarchical. These types of P2P overlays are discussed in sections 2.4.2 to 2.4.5.

2.4.2 Unstructured P2P Network

Unstructured P2P networks form arbitrary topologies. Each node joins the network following some basic local rules to establish connectivity with other nodes. The network can use flooding as the mechanism to send queries across the overlay with a limited scope or more sophisticated strategies such as random walks [136, 234], gossiping, and routing indices [350]. In an unstructured P2P network, when a node receives a query, it matches it locally against its own content and sends a list of all content matching the query to the requesting node. If there is no match, the query is sent to some neighbor using, for instance, flooding. Flooding techniques are effective for locating highly replicated objects and they are resilient to highly-transient node populations. However, they are not scalable as the load of each node and the size of the system increases linearly with the number of queries. In general, an unstructured P2P overlay has the following advantages: (i) it supports complex queries since all searches are performed locally; (ii) it is self-organized and resilient to either node failures or nodes frequently joining and leaving the network [175, 235], which is known as churn; (iii) it is easy to built since it does not impose any constraint about the topology [77]; and (iv) it has lower maintenance overhead, especially in the presence of a high churn rate.

On the other hand, the disadvantages of unstructured P2P overlays are: (i) they cannot guarantee on locating an object, even though the object exists in the network. This is due to the strategy used for searching an object. In other words, as the size of the network is unknown, the overlay limits the number of nodes that a searching message can be forwarded to; and (ii) they are not scalable due to high message overhead [14].

Examples of unstructured P2P overlay are: Publius [361], FreeHaven [101], Gnutella (rfc-gnutella.sf.net), FreeNet (freenetproject.org), and BitTorrent (bittorrent.com).

2.4.3 Structured P2P Network

Even though unstructured P2P overlays are simple and easy to deploy, they suffer from low query processing efficiency. Unlike unstructured P2P overlays, structured P2P overlays impose constraints on the topology and on data placement to enable efficient resource discovery. In these systems, when a node joins the system, it follows some strict procedures to set up its position in the system according to the topology used.

The constraints imposed by structured P2P networks allow an efficient and accurate resource discovery. In particular, these systems can guarantee that if a resource exists in the system, it will be found with at most $O(log\ n)$ messages where n is the number of nodes in the system [14]. This resolves the issues of resource discovery and scalability of the unstructured P2P systems. Nevertheless, structured P2P systems require considerable efforts to maintain them in a consistent state, which makes structured P2P overlays vulnerable in a churn scenario. In addition, as some structured P2P overlays are unaware of the underlying network due to random key assignment, one logical hop can correspond to multiple physical hops.

In general, structured P2P systems can be classified in three categories based on the distributed data structure used: distributed hash table (DHT) based system, skip list based system, and tree based system [360].

A distributed hash table based system uses a DHT to organize the nodes and to index the data. In this case, each resource is identified by a key obtained by a uniform hashing function [184] such as SHA-1. The key space is divided among the nodes, where each node is responsible for one partition of the key space and for storing the data or a pointer to them locally. As a result, DHT-based systems support efficient exact match queries and uniform load distribution among the nodes, which ensures good performance. Nevertheless, the disadvantages of DHT-based overlays are: (i) the lookup latency in the DHT-based overlays can affect the performance of the applications running on them; (ii) they do not provide anonymity since they map resources directly to nodes and store this information for routing purpose; and (iii) they cannot support range queries since the uniform distribution destroys the data ordering. DHT-based systems can be based on several structures such as rings as in Chord [332], a multi-dimensional grid such as in CAN [291], a Plaxton mesh as in Tapestry [391] and Pastry [302], or a XOR-based metric such as in Kademlia [241].

Skip list based systems employ the Skip list [285] structure to place the nodes and to partition the data. In these systems, each level has many lists and a node participates in a list at each level. As result, these systems preserve the order of the data, supporting both exact match queries and range queries. Moreover, unlike DHT-based systems, skip-list-based systems can inflate or deflate without needing to know the size of the system to determine the range key values of the nodes. Example of skip list based systems are Skip Graph [21], SkipNet [150], and SkipIndex [384].

Tree based systems use different tree types to index the data to support range queries efficiently. Examples of the tree types employed by these systems are binary prefix tree as in P-Grid [2], multi-way trees as in [223], balanced trees as in BATON [171], B^+trees as in P-ring [89, 90], and R-trees as in [172, 221, 252].

In section 2.4.3, we describe Chord [332], one of the DHT-based P2P overlays most used in the literature and also used in this doctoral thesis.

Chord

Chord [332] organizes the nodes in a one-dimensional circle according to their identifier. It uses a consistent hash function [184] to assign an $m - bit$ identifier to each node and data. The identifier space is a circle of numbers from 0 to $2^m - 1$, where m is the number of bits in the identifier space. A node identifier ($N.id$) is chosen by hashing the node's IP address and port, while a key ($K.id$) is chosen by hashing the data. In particular, the value of m must be big enough to make the probability of key collision negligible.

In Chord, a key k is assigned to the first node whose identifier is equal to or follows k in the identifier space, which is denoted the successor of k ($successor(k)$). Each node maintains two links called predecessor ($N.pred$) and successor ($N.succ$), where $N.pred$ is equal to $N.id - 1$ and similarly, the $N.succ$ is equal to $N.id + 1$, and a finger table to accelerate resource discovery queries. A finger table consists of a data structure with a maximum of m entries. Each entry i in the finger table of a node N represents the first node whose identifier succeeds or is equal to $N + 2^i$, where $0 \leq i \leq m - 1$. A finger table entry includes both the node identifier and its IP address and port. As a node can join and leave the system any time, Chord uses a stabilization protocol [332] that runs periodically to update the nodes' links and the entries in the finger table.

In Chord, when a node N wants to find a key k, it first searches in its finger table for a node n' such that its identifier ($n'.id$) is between n and k. If such node exists, node n asks n' to re-start the search. Otherwise, n asks its immediate successor to find k.

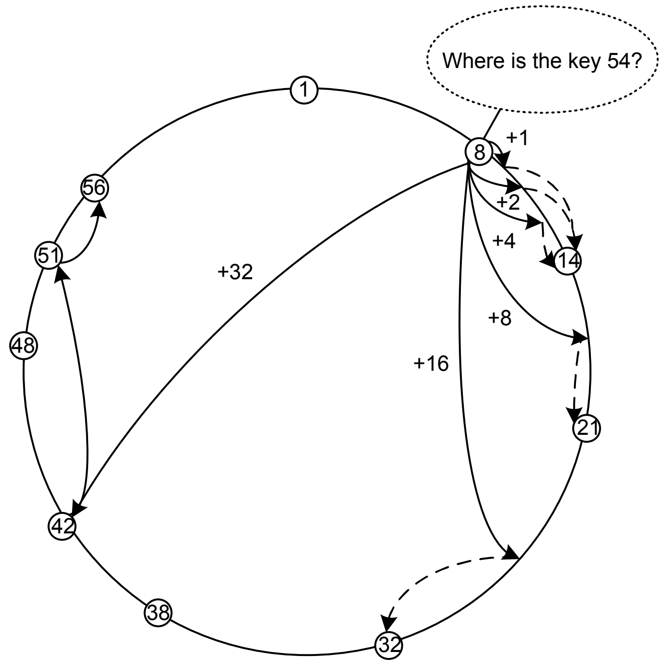
For example, suppose a P2P network with 10 nodes built using Chord to store five keys ($k_1 = 10$, $k_2 = 24$, $k_3 = 30$, $k_4 = 38$, $k_5 = 54$), m equals to 6, and a query lookup for key $k = 54$, starting at node $N8$ as shown in figure 2.10. In this scenario, k_1 will be located at node 14, as N_{14} is the first node whose identifier follows the identifier 10. In the same way, the keys k_2 and k_3 would be located at node 32, key 38 at node N_{38}, and key 54 at node 56. To lookup the the $k = 54$, first, node N_8 checks if the key is in either its identifier space or in its successor. After, it searches its finger table to find the farthest node that precedes the key ($k = 54$) in the identifier space. Since node N_{42} is this node, N_8 asks it to resolve the query. As a result, node N_{42} forwards the query to node N_{51}. Then, node N_{51} discovers that its successor, node N_{56}, succeeds the key, and hence, it continues to forward the the query to node N_{56}. Finally, since node N_{56} holds the key, it returns the result to node N_8.

Using this mechanism, Chord can find a key requiring only $O(log\ (n))$ messages, even in the presence of node failures [332]. Moreover, in Chord, nodes can dynamically join or leave the network. In these cases, the ring structure and finger tables must be updated. Aggressive updates must be avoided since they add a considerable overhead to the system. For this reason, Chord uses a stabilization scheme that first fixes the predecessor and successor information (Chord ring) and asynchronously adjusts the finger tables [332].

2.4.4 Hybrid P2P Network

Due to the scalability problem of unstructured P2P overlays and the maintenance cost of structured ones on a high churn scenario, hybrid P2P overlays [231, 323] can be used to combine the advantages of structured and unstructured P2P overlays and to minimize their disadvantages. Hybrid P2P overlays often use flooding techniques for searching highly replicated resources and a DHT to locate resources which are rarely accessed.

A major problem of hybrid P2P overlays is how to distinguish rarely accessed resources from popular ones in a decentralized and self-organized environment. One solution can be to identify rare resources based on the number of queries in which they appear, and



N_8's Finger table

N8 +1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 + 16	N32
N8 + 32	N42

Figure 2.10: Example of the lookup for the key $k = 54$ starting at the node N_8 and $m = 6$ [332]

using a DHT to cache them. Another solution is gossiping historical summary statistics of replicated resources [231].

Many approaches have been proposed to build hybrid P2P overlays. For instance, Castro and colleagues [65] use search and data placement strategies of unstructured overlays on a structured overlay to implement complex queries with a smaller number of messages and with a higher rate of success response queries.

2.4.5 Hierarchical P2P Network

Hierarchical P2P overlays introduce layers to reduce the network load. To keep the P2P advantages such as self-organization and decentralization, hierarchical P2P overlays distinguish the nodes as super-peers and leaf-nodes. In this case, peers with a large amount of physical resources such as CPU and network bandwidth are elected as super-peers and they are responsible for defining a set of services for peers with fewer resources, i.e., the leaf-nodes or ordinary nodes [336, 395]. Ordinary peers are connected to a super-peer that acts as a proxy for them. Super-peers contact other super-peers on behalf of their peers through flooding [336], gossiping [208, 228], or through a DHT [8, 64, 161, 250]. For example, Hui and colleagues use a DHT to organize the nodes in small clusters managed by a head node (i.e., super-peer) to minimize the maintenance overhead of the structured overlay. In this case, each cluster has one head node and k inner nodes. A head node is responsible for maintaining a link for both its inner nodes (i.e., leaf-nodes) and for other head nodes. Inner nodes of different clusters communicate through their head nodes. In other words, when an inner node wants to communicate with a node outside its own cluster, it sends a message for its super-peer that first identifies the super-peer of such node, then it forwards the message for it. Other approaches [148, 270, 377] use a ring to organize the super-peers and a tree to place the ordinary peers, and some [323] are interested in mutual anonymity and content ownership.

Super-peers can be dynamically elected according to the network load, since only the super-peers are involved in advertising, discovering, and resource selection [26]. Many hierarchical overlays have been proposed such as TOPLUS [131], and Chordella [161].

2.4.6 Comparative View of P2P Structures

Table 2.1 presents a comparative view of the P2P structures discussed in section 2.4. In the first column we show the P2P structure. The second column shows the scalability of the structure considering the number of messages to find a resource. In the third column, the robustness of the P2P structure is presented, considering a network under highly churn scenario. Finally, the last two columns present if the given structure requires maintenance procedures to work properly, and if it supports range queries.

Table 2.1: Comparative view of the P2P overlay networks

Network structure	**Scalability (hops)**	**Robustness**	**Maintenance operations**	**Range queries**
Unstructured	$O(n)$	High	No	No
Structured	$O(log\ n)$	Low	Yes	Yes
Hybrid	$O(log\ n)$	High	Yes	Yes
Hierarchical	$O(log\ n)$	High	Yes	Yes

Scalability is an import characteristic of P2P systems and unstructured P2P systems lack this property, due to the network overhead by flooding messages. Although some improvements have been made to reduce this overhead, such as random walks [136, 234], gossiping, and routing [350] indices, they increase the response time without guarantees of finding the resources. On the other hand, structured networks are more scalable, but they incur a high cost of maintenance under a churn scenario.

As P2P systems are expected to be formed of ephemeral nodes, this is a serious concern for structured P2P systems. In these systems, nodes must be notified periodically to update their view about the system. The result is an increase in the network traffic, if the interval between two notifications is small, or outdated information if the interval is too large. Although this is not a crucial problem for content-sharing P2P applications [14], it can be an issue if the structured P2P system is used as resource discovery in a grid environment, for instance.

These issues can be solved with hybrid P2P systems, since they often use unstructured techniques for searching replicated resources and a DHT to locate rare resources. Nevertheless, there is the cost of propagate the queries statistics to identify rare resources. Furthermore, some nodes may become overloaded due to the high number of messages to process or due to their limited computational capacity. The solution may be the use of hierarchical P2P networks since they decrease the number of messages by employing the concept of super-peers.

Moreover, with the exception of unstructured P2P systems, where each node answers the queries using only its own data, P2P systems can be used for resource discovery in large-scale distributed systems since they can process queries efficiently [349].

2.5 Cloud Computing

Many cloud computing definitions have been proposed over the last years. One reason for the existence of different perceptions about cloud computing is that cloud computing is not a new technology, but a new operational model that brings together a set of existing technologies in a different way [353, 386].

The National Institute of Standards and Technology (NIST) defines cloud computing as [243]:

> A model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service interaction.

To Foster and colleagues [121], cloud computing is:

> A large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on-demand to external customers over the Internet.

Finally, Buyya and colleagues [56] define cloud computing as:

> A type of parallel and distributed system consisting of a collection of interconnected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resource(s) based on service-level agreements established through negotiation between the service provider and consumers.

These three definitions focus on certain aspects of cloud computing such as resource sharing, virtualization and provision. In the NIST [243] definition, cloud computing is characterized by the idea of elastic capacity and the illusion of infinite computing resources, available through network access that can be easily provisioned with minimal management effort or negotiation with the service provider. In this definition, cloud computing is a specialization of the distributed computing paradigm that differs from traditional distributed computing because it can be encapsulated as an abstract entity that delivers different levels of services to customers outside the cloud and that is driven by economies of scale, dynamically configured with on-demand delivery.

Even though there are considerable differences among the cloud computing definitions, most of them state that a cloud computing system should have (i) pay-per-use capabilities, (ii) elastic capacity and the illusion of infinite resources, (iii) self-service, and (iv) abstract or virtualized resources.

In this thesis, we consider a definition that is mainly based on [56] and [121], where cloud computing is a type of distributed system that dynamically provisions virtualized

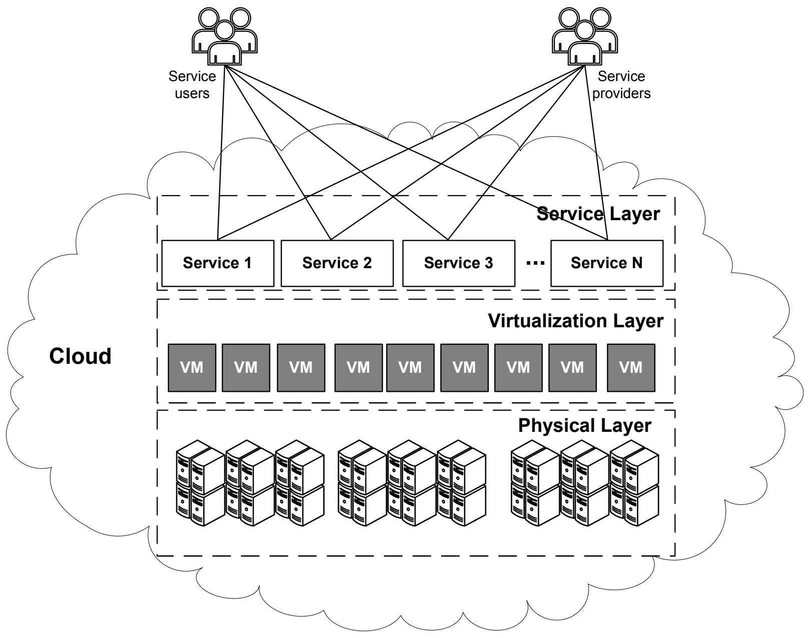


Figure 2.11: A cloud computing system

elastic and on-demand resources, respecting service-level agreements (SLA) defined between the service provider and the consumers.

In a cloud computing system, a cloud user or cloud customer is a person or organization that uses the cloud. It may also be another cloud and, in some cases, that cloud may be at the same time both a cloud user and a cloud provider. A cloud provider, on the other hand, is either an organization or distributed organizations perceived as a single unit by the consumers, that provide cloud services. In other words, in a cloud computing environment, the role of the cloud provider is demonstrated in two aspects: the *infrastructure providers*, who are responsible to manage cloud platforms and to lease resources according to a usage-based pricing model, and the *service providers*, who rent resources from infrastructure providers to serve the end users [386]. A cloud client is a machine or application that accesses a cloud over a network connection, perhaps on behalf of a cloud consumer [24]. Figure 2.11 illustrates a typical cloud computing system.

The cloud computing model overlaps with many existing technologies such as grid computing, P2P, and Web applications in general as depicted in figure 2.12.

2.5.1 Characteristics

According to Mell and Grance [243], the cloud model is composed of five essential characteristics: on-demand self-service, broad network access, resource pooling, elasticity, and measured service.

on-demand self-service: the customers can provision resources without requiring human negotiation with the providers.

broad network access: cloud services are available over the Internet and can be accessed through standard network protocols.

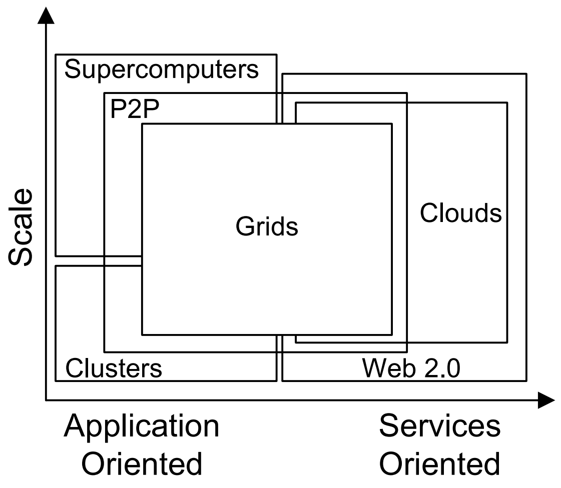


Figure 2.12: A vision of grid, P2P, and cloud computing characteristics overlaps. Adapted from [121]

resource pooling: cloud resources are dynamically provisioned by the providers to serve multiple cloud users using a multi-tenant[1] model according to user's demands, where the cloud users should only be aware of some high-level location information about the cloud's infrastructure such as country, state or data center as they might have some location restrictions.

elasticity: resources can be dynamically provisioned or released on-demand and in any quantity.

measured service: the provider monitors and controls the resources used by the cloud users; provides appropriate report levels according to the type of the service; and charges the cloud users based on a pay-as-you-go model.

Cloud computing provides several features that make it attractive, such as [20, 386]:

no need for up-front investment: resources are available for the users in a pay-as-you-go model. In this case, a service provider does not need to invest in the infrastructure to gain the benefits from cloud computing. It just demands resources from the cloud according to its own need and pays for the usage.

low operating cost: resources can be easily allocated and de-allocated on-demand. In other words, the service provider provisions the resources according to the peak load and releases them when the service demand is low. This allows companies to start small and increase the number of resources only when there is an increase in their needs.

high scalability: infrastructure providers maintain a pool of resources from typically large data centers and make their resources accessible to users, eliminating the need

[1]A tenant is a user that shares his/her computing infrastructure such as application instance or hardware with other users (i.e., tenants), but they have their data and control flow completely isolated [43]. This constitutes a multi-tenant environment.

to plan far ahead for provisioning, and achieving high scalability in a relatively easy way.

easy access: services hosted in the cloud are generally web-based and easily accessible through a variety of devices with Internet connections.

reduced business risk: in this case, a service provider shifts its business risk to the infrastructure provider, which often has better expertise and is better equipped for managing these risks.

To Foster and colleagues [121], the factors that contributed to the growing interests in cloud computing are:

1. the rapid decrease in hardware cost, the increase in computing power and storage capacity, combined with the advent of multi-core architectures and modern supercomputers consisting of hundreds of thousands of cores.
2. the exponentially growing data size in scientific instrumentation/simulation and Internet publishing and archiving.
3. the wide-spread adoption of services computing and Web 2.0 applications.

Cloud computing is also gaining popularity because it helps companies to reduce costs and carbon footprint. Cloud data centers generally employ virtualization techniques to provide computing resources as utilities and virtual machine (VM) technologies for server consolidation inside big data centers, containing a large number of computing nodes. To realize the potential of cloud computing, cloud providers have to ensure flexibility in their services to meet different usage pattern requirements, allowing on-demand access to resources, with no need for the user to provision or maintain resources. At the same time, cloud providers aim to maximize the resources utilization and to reduce energy consumption, for instance.

2.5.2 Drawbacks

There are still many problems concerning cloud computing such as [46, 144, 167, 338, 364, 365]:

lack of appropriate cloud simulation and testing tools: before moving his/her infrastructure to the cloud, the service provider should be able to simulate it and reason about the advantages/disadvantages of this movement. In a similar way, the end user should also be able to simulate the execution of his/her application in a controlled cloud environment. So far, there are few simulation tools for clouds [58, 198, 329, 372] and they are not prepared to simulate real-world scenarios.

reduced performance stability: since clouds use virtualized resources, which are allocated to the application in a totally transparent way, big performance variations can occur [167]. This is particularly problematic when running scientific applications that take days or even weeks to execute.

data lock-in and standardization: the lack of standardized APIs can restrict data migration between different cloud providers and often prevents data sharing or restricts the user to save data in a different format. This can easily lock a user to a particular cloud provider. In some situations, the data are physically stored in distributed locations and only the cloud provider has full control over the data. For instance, it is often the case where only the cloud provider has access to logs and the ability to remove data physically. One way to deal with the data lock-in issue can be using independent data representation and adopting standard data import and export functionality [275]. Google has attempted to address this issue through its *data liberation front* (dataliberation.org) whose goal is to allow data movement into/outside the Google infrastructure [117]. Other ways can be [156] (i) using APIs that have different implementations; (ii) choosing an application model that can run on multiple clouds (e.g., MapReduce); (iii) manually decoupling the cloud-specific code of the application designed for each cloud provider from the application logic layer; (iv) creating widespread standards and APIs; and (v) using vendor-independent cloud abstraction layer. Cloud standardization is difficult since there are many barriers to adopt standard APIs and protocols [275, 276] such as (i) cloud providers are often creating facilities to avoid losing their users to other cloud providers; (ii) cloud providers offer differentiated services and they want to have unique services to attract more users; (iii) cloud providers usually do not easily agree on standards; (iv) standards take years to be developed and to be adopted globally; (v) there are various standards being developed simultaneously and agreement on which one to adopt may be difficult, if not impossible, to attain; and (vi) the cloud computing model requires multiple standards, rather than one overarching set of standards.

reduced number of effective energy-efficient solutions: since the cloud applications are executed in big data centers, power usage becomes a major concern. Even though the industry and academy are starting to investigate energy-efficient proposals for cloud computing environments, an effective and integrated solution to this problem is yet to be conceived.

reduced security: security is a difficult issue [257, 338] for many reasons. First, a cloud environment may have heterogeneous hardware and software resources. Second, the cloud works on employing a shared model, where services are spread across multiple providers. This can lead to ownership and compliance concerns. Finally, the delivery and deployment model of cloud computing may result in security breaks such as incorrect data isolation level. To some extent, virtualization achieves resource homogeneity, but as cloud services are shared by different users, data and application might break down, compromising its confidentiality and/or integrity. Security breaks can occur intercepting inter-host or multi-domain communication messages, VM migrations, preparing malicious VM images, failure in multi-tenancy implementations, malicious service implementation, among others [185, 338]. Also, a cloud service can involve different providers. For instance, a user can subscribe to one cloud provider, that is subscribed to another cloud provider, which utilizes the infrastructure of a third cloud provider. In this scenario, the guarantee of trust and security properties can be affected since each cloud provider has normally distinct security policies in a manner that the user does not have guarantees that his/her data are protected among cloud services interactions. In addition to that, there is a lack of transparency about security assurances [185].

data integrity: several researches reported internal and external data threats for data integrity in the cloud that resulted in data loss [364]. Also, some providers discard rarely accessed data [364], decreasing their availability.

2.6 Summary

In this chapter, we discussed many important aspects of several types of large-scale distributed systems. First, we briefly presented the main landmarks of the last fifty years that led to the current state of these systems. Then, cluster, grid, P2P, and cloud computing systems were briefly discussed, considering their characteristics and architectures.

While grid applications are often built to provide services for moderated-size communities assuming a stable environment to be able to deliver non-trivial qualities of services, P2P systems are designed to offer limited services to a huge number of participants in an unstable, unpredictable, and often untrusted environment. These distinctions are associated to the goals of each environment.

Grid computing was driven by the need for more compute power [126], aggregating powerful resources distributed across different virtual organizations, normally universities and research institutes, to enable collaboration in a specific domain.

A P2P system, on the other hand, was driven by the need to decrease costs with the use of commodity computing resources autonomously organized, employing ad-hoc communications to increase collaboration among the peers [14]. This not only decreases the cost, but also allows information to be disseminated effectively in a large-scale scenario [360].

A consequence of these communities characteristics is that early grid computing did not address scalability and self-management as priorities [126, 360]. Furthermore, the participants in a grid environment are trustful, usually not large, and they have incentives to collaborate following well-defined policies. This enhances the resource's ability to deliver the expected quality of service but with a higher cost to manage the resources.

P2P computing became popular offering mass-culture services (e.g., file-sharing) with anonymity guarantees, and executing highly parallel applications (e.g., SETI@home [370]) that scale in presence of thousands of nodes with intermittent participations and highly variable behaviors [123]. In contrast to grids, the number of participants in a P2P system is very large (e.g., hundred of thousands [12]) and the participants do not need to be trustful.

In the last decade, we observed a convergence between grid computing and P2P systems [123], aiming to provide the best characteristics of each paradigm. More specifically, many grid environments began to employ P2P decentralized techniques to deal with large-scale resource sharing. In the same way, in addition to the traditional file sharing applications, P2P systems started to execute a wide range of scientific applications that require sophisticated resource management techniques, usually present in grid systems.

Cloud computing is similar to grid computing since it also aims to provide distributed resources to achieve some computing objectives at reduced costs. However, whereas the resources in grid are provided through immediate or advance reservation, cloud employs an on-demand resource provision. This removes the need of reservation in order to meet the users' demands [20, 388] and reduces the costs. Also, the cloud infrastructure is available to everyone at different scales [139, 181, 335] and not only to members of some communities as in the grid. Moreover, grid computing is a one-model solution with tools as the Globus Toolkit [122] deployed to enable resource aggregation [325]. On the other hand, a cloud-based solution is based on multiple models (i.e., IaaS, PaaS, and SaaS), which leads to a flexible environment for the customers. In the cloud, the customers do not need to wait in a queue to execute their applications, but they can use the clouds' APIs to allocate the resources on-demand in nearly real-time.

We claim that, in the next years, a distributed computing environment will be built combining the advantages of each distributed architecture discussed in this chapter, available for the users as a utility computer. It will require adequate tools to abstract the computer according to the users goals and the users shall be unaware of it such as it happens in the Internet today, approximating to the utility computing concept envisioned by McCarthy [132].

Chapter 3

A Detailed View of Cloud Computing

Contents

This chapter presents practical aspects that must be considered when developing a cloud computing solution. First, we discuss virtualization techniques in section 3.1.1, since the majority of cloud systems employ virtualization techniques for resource management and workload isolation. Then, in section 3.1.2, we briefly discuss service-level agreement (SLA), and the guarantees that are usually provided by the clouds. Cloud applications are usually programmed using the MapReduce programming model, and we present the concepts of MapReduce in section 3.1.3. The cloud organization is presented in section 3.2 followed by some available cloud standards and metrics (Section 3.3). Cloud computing systems are discussed in section 3.4 followed by a review of IaaS cloud architectures (Section 3.5). Finally, section 3.6 summarizes this chapter.

3.1 Technologies Related to Cloud Computing

3.1.1 Virtualization

In this section, we discuss virtualization, which is a technology widely adopted to implement clouds since it helps to increase resource utilization in an effective way. First, we present a definition of virtualization. Second, we describe the virtualization techniques. Then, the concept of virtual machine live migration and two approaches used to implement it are discussed. Finally, we present the concept of workload and server consolidation, which are by products of virtualization.

3.1.1.1 Definition

The term virtualization refers to the abstraction of computing resources (e.g., CPU, disk, network) from the applications aiming to improve sharing and utilization of computer systems [138]. The use of virtualization exists since 1960s, when it was first implemented by IBM to provide concurrent, interactive access to the mainframe 360/67 (Sections 2.1 and 2.1.4).

A virtual machine (VM) is an environment provided by a virtualization software called virtual machine monitor (VMM), or hypervisor. In this case, the virtualization layer is placed between the bare hardware and the guest operating systems and gives the OSes a virtualized view of the hardware as shown in figure 3.1. The platform used by the hypervisor is named host machine, and the module that uses the virtual machine is named guest machine. An important function of the hypervisor is to provide the connection between virtual machines and the host machine. It also abstracts the resources of the host machine, which will be used by the operating system through the virtual machine, and it provides the isolation among virtual machines placed in the same host machine, guaranteeing the independence of each other. Furthermore, the hypervisor handles changes in the processor where the application is running on without affecting the user's OS or application [258, 325].

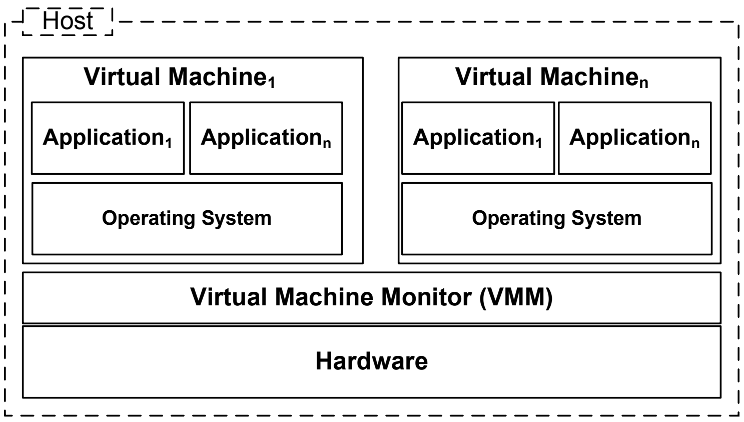


Figure 3.1: Example of virtualization [138]

3.1.1.2 Techniques

Virtualizing the entire hardware faces some challenges. First, most of the CPU architectures were not designed to be virtualizable. Second, virtualization requires to place a layer between the hardware and the operating system to create and to manage the virtual machines. In other words, the guest operating system must often run in an unprivileged level. However, most of the x86 operating systems are designed to run directly on the bare hardware (privileged level), having full control of the machine. In this scenario, the VMM must be able to intercept privileged instructions performed by the virtual machine and to give them the appropriate treatment [283]. Nowadays, there are three techniques to perform this interception namely *full virtualization*, *paravirtualization*, and *hardware-assisted virtualization*.

Full virtualization provides a complete hardware emulation allowing the guest operating system to have full control of the hardware. In this case, the guest operating system runs in privileged mode using the hypervisor to intercept and to translate privileged instructions on-the-fly [283, 301]. The main advantage of this technique is that the guest OS runs completely unaware of the virtual machine environment and does not need to be modified. The main disadvantage is the reduced performance of the VMM, due to frequent context switching activity. As example, we can cite KVM [197], Microsoft Hyper-V, and VMware ESX Server [100, 362].

Paravirtualization introduces a virtualization layer on top of the bare hardware. Unlike full virtualization, in this technique the guest operating system is aware of the virtual machine. In this case, the kernel of the guest OS is modified to delegate the execution of non-virtualized instructions to the VMM and this simplifies the design of the hypervisor. This technique can lead to better performance than full virtualization since the hypervisor and the operating system are aware of each other and can cooperate to achieve several tasks. Also, the operating system is provided with an interface to access directly some devices like disks or network interfaces [28], improving the performance. As an example, we can cite Xen [28].

The *hardware-assisted* virtualization is an alternative approach which aims to decrease the performance gap between paravirtualization and full virtualization. It uses special hardware instructions to automatically direct privileged and non-virtualized instructions to the VMM, instead of emulating them. This allows the guest operating system to run without changes. However, it has the overhead of context switches between the VMMs, and it requires hardware support, which imposes a higher cost to server consolidation [5].

3.1.1.3 Live Migration

Live migration is the seamlessly movement of running virtual machines from one physical machine to another with negligible downtime (i.e., in order of milliseconds) [80]. For instance, with live migration, a streaming media server can be migrated without requiring the client to reconnect. Furthermore, moving an entire virtual machine allows the administrators to optimize the placement of system workload, to perform maintenance tasks, and to re-allocate resources on-the-fly without knowing details about the virtual machine, which significantly improves the system manageability [80].

In most cases, VM's live migration is done using a network-attached storage (NAS) in preference to using local disks in each individual nodes, in order to avoid virtual machine (VM) disk migration. It must also be noted that live migration without shared storage is only supported by few hypervisors.

VM live migration can be categorized into two approaches: pre-copy [80] and post-copy [157].

In the pre-copy approach [80] the VM's memory pages are iteratively copied to the target node while the VM continues to execute at the source node. If, during the copy process, a transmitted page is modified, it is re-sent to the target node in another copy iteration. After the end of the memory state transfer, the VM is suspended at the source node and its CPU state and any remaining inconsistent memory pages are transferred to the target node, where the VM is resumed. Finally, with the acknowledge of the target node, the device drivers are reconfigured in the new node, the IP address of the migrated VM is advertised, and the source node releases its allocation of the virtual machine. This approach aims to minimize both VM downtime and application's performance degradation when the VM is executing a read-intensive workload.

On the other hand, the post-copy approach aims to minimize the network overhead due to possible duplicate memory copies, transferring each memory page at most once. The post-copy [157] live migration approach works as follows. The VM is first suspended on the source node and its CPU state is transferred to the target node. Then, the VM is resumed at the target node, that actively pushes the VM's memory pages from the source from the target node, when they are accessed. Note that this approach creates a residual dependency between the source and target nodes.

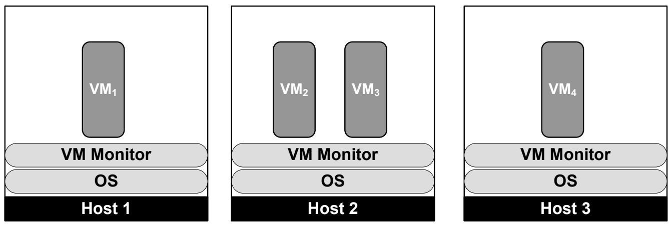
The usage of each VM live migration approach depends on the VM workload type and the performance goals of migration. For VMs with read-intensive workloads, the pre-copy approach would be the best approach whereas the post-copy approach is best suited for cases where a small number of pages is accessed in the target node.

3.1.1.4 Workload and Server Consolidation

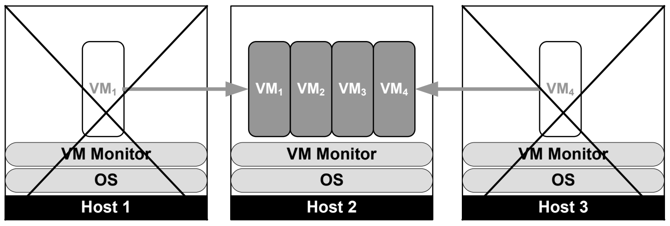
Consolidation is a technique that reallocates virtual machines to achieve some objectives. For instance, it can be used to reduce the amount of physical machines to run the virtual machines or to improve performance of a virtual machine, migrating it for a more powerful physical machine [239, 358].

Virtualization and live migration make possible to consolidate heterogeneous workloads onto a single physical platform, reducing the total cost of ownership and leading to better utilization. This practice is also employed to overcome potential software and hardware incompatibilities in case of upgrades, allowing systems to run legacy and new operating systems concurrently [351]. Recently, consolidation has been used to reduce the number of underutilized servers.

Figure 3.2 illustrates the consolidation strategy. First, a power-inefficient allocation is shown (Figure 3.2(a)). In this case, there are three active quad-core hosts, two of them with 25% of their capacity utilized and one with 50% of its capacity utilized. With consolidation, as shown in figure 3.2(b), all virtual machines are allocated in one host and the other hosts can be turned off, reducing the power consumption of the whole system.



(a) Before consolidation



(b) After consolidation

Figure 3.2: Example of workload consolidation using virtual machines

Server consolidation uses workload consolidation in order to reduce the number of active servers. It is the process of gathering several virtual machines into a single physical server. It is often used by data centers to increase resource utilization and to reduce electricity costs [358].

The consolidation process can be performed in a single step using the peak load demands, known as static consolidation, or in a dynamic manner, re-evaluating periodically the workload demand in each virtual machine (i.e., dynamic consolidation).

In static consolidation, once allocated, a virtual machine stays in the same physical server during its whole lifetime. In this case, live migration is not used. The utilization of the peak load demand ensures that the virtual machine does not overload. However, in a dynamic environment with different load patterns, the virtual machine state can be idle most of the time, resulting in an inefficient power allocation.

Dynamic consolidation usually yields better results since the allocation of virtual machines occurs according to the current workload demands. Dynamic consolidation may require migrating virtual machines between physical servers in order to [113]: (i) pull out physical servers from an overload state when the total number of virtual machines mapped to a physical server becomes higher than its capacity; (ii) or turn off a physical server when it is idle or when the virtual machines mapped to it can be moved to another physical server.

Consolidation influences utilization of resources in a non-trivial manner. Clearly, energy usage does not linearly add when workloads are combined. For example, in an Intel i7 machine (4 real cores and 4 cores emulated) an application using 100% of one core, with the other cores in the idle state, consumes 128W whereas the same application using 100% of eight cores consumes 170W [36]. Moreover, resource utilization and performance can also change in a non-trivial manner. Performance degradation occurs with consolidation because of internal conflicts among consolidated applications, such as cache conflicts, conflicts at functional units of the CPU, disk scheduling conflicts, and disk write buffer conflicts [359].

In a cloud computing environment, server consolidation presents some additional difficulties such as: (i) the cloud computing environment must provide reliable QoS, normally defined in terms of service-level agreement (SLA), which describe characteristics such as minimal throughput and maximal response time delivered by the deployed systems; (ii) there can be dynamic changes of the incoming requests for the services; (iii) the resource usage patterns are usually unpredictable; and (iv) the users have distinct preferences.

3.1.2 Service-Level Agreement

In cloud computing, the relation between consumers and service providers is usually based on service-level agreement (SLA) [66]. A service-level agreement (SLA) defines the level of services, priorities, guarantees, warrants, and obligations of both service providers and consumers. It may also specifies the penalties in case of violation of the SLA [382]. Moreover, an SLA includes some service parameters referred to as quality of service (QoS), such as availability, throughput, reliability, security, and performance indicators (e.g., response time, I/O bandwidth) [367, 382].

Since different organizations have distinct definitions for QoS parameters, it is not possible to fulfill all consumer expectations from the service provider perspective and a balance needs to be made via a negotiation process, committed in the end by the consumer and the service provider. After the agreement has been established, it is recommended to continuously monitor the attributes to ensure adherence to the contracted SLA parameters or, in some cases, renegotiate them. This requires dedicated resources or an assistance of a third-party to measure the SLA's parameters appropriately. In the literature, there are many frameworks [52, 199, 316] to help consumers and providers to measure their SLA's parameters.

In practice, an SLA is a document composed of several sections, which usually includes [45]:

objective: describes why the agreement was created.

parts: describes the parts involved in the agreement and their roles (consumer, provider).

validity: defines the period of the time for which the agreement is valid.

scope: defines the services and resources involved in the agreement.

service level objectives: determines the service levels agreed between the parts, which normally include indicators such as availability, response time, CPU time, disk usage, among others.

penalties: defines the actions that should be taken if the service level objectives are violated.

Figure 3.3 presents an example of an SLA document written in the Web service level agreement (WSLA) language [190]. In this case, it describes an agreement named *StockquoteServiceLevelAgreement12345* specifying that the service provider must keep the service response time below 5 seconds.

Nowadays, most of the cloud providers define their SLA in function of availability guarantees. In such case, they define the minimum percentage of time that their services will be available during a certain period of time and the penalties as discount for the users.

```
<wsla:SLA
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:wsla="http://www.ibm.com/wsla"
  name="StockquoteServiceLevelAgreement12345">

  <Parties>
    ...
  </Parties>

  <ServiceDefinition>
       ...
  </ServiceDefinition>

  <Obligations>
    <ServiceLevelObjective name="g1">
      <Obligated>provider</Obligated>
      <Validity>
        <Start>2001-11-30T14:00:00.000-05:00</Start>
        <End>2001-12-31T14:00:00.000-05:00</End>
      </Validity>
      <Expression>
        <Predicate xsi:type="wsla:Less">
          <SLAParameter>AverageResponseTime<SLAParameter>
          <Value>5</Value>
          <TimeUnit>second</TimeUnit>
        </Predicate>
      </Expression>
      <EvaluationEvent>NewValue</EvaluationEvent>
    </ServiceLevelObjective>
  </Obligations>
</wsla:SLA>
```

Figure 3.3: Example of an SLA structure. Adapted from [190]

3.1.3 MapReduce

Designing and writing applications to execute in large-scale distributed systems is often time consuming and a complex task. For instance, the developers are responsible for specifying how the data are partitioned among the nodes; for handling failures and performance variations during the applications' execution; and finally, for synchronizing data and processes.

Many programming frameworks and models have been created to address these issues. MapReduce [98] is one of them, and it has been adopted by industry and academia due to its characteristics such as automatic fault-tolerance and parallelization of the applications adapted to the MapReduce model.

This section describes the MapReduce model, its characteristics and limitations.

3.1.3.1 Definition

MapReduce was proposed as a parallel programming model targeted to data-intensive applications running on clusters of commodity machines [98]. In this model, a unit of work is divided into two major phases called *map* and *reduce*. The *map* phase computes a set of intermediate key/value pairs from the input data, as defined by the user, groups all intermediate values associated with a key and passes it to the *reduce* phase. The *reduce* phase accepts the key and its intermediate values, producing zero or one output value. The intermediate values are supplied to the *reduce* function via an iterator, allowing to handle lists of values that are too large to fit in memory [98].

Listing 3.1 shows an example of a MapReduce code for counting the occurrences of each word in a text.

```
map(keyin, text)
  foreach word w in document
     emit-intermediate(w,1)

reduce(keyout, values)
    foreach v in values
       result += v
    emit(keyout,result)
```

Listing 3.1: Counting the number of occurrence of each word in a text using MapReduce [98]

Conceptually, the MapReduce execution model can be expressed as [98]:

$$\begin{aligned} map &: (K_1, V_1) \rightarrow [(K_2, V_2)] \\ reduce &: (K_2, V_2) \rightarrow [V_2] \end{aligned}$$

where K and V are respectively the key and value pairs. In the map operation, input keys (K_1) and values (V_1) belong to different domains than the output keys (K_2) and values (V_2).

3.1.3.2 Characteristics

In MapReduce the processing engine and the underlying storage system are designed to scale up or down independently. The storage system often uses a distributed file system to split and to distribute the data over the machines. In this case, each partition is used as an input for the mapper. Therefore, if the input data are split into M partitions, MapReduce will create M mappers to process the data. In practice, the processing engine is structured as a master/slave system, where the master is responsible to assign the tasks (map and reduce) for the workers and to coordinate the execution flow. A worker parses the key/value pairs out of the input data and executes the user-defined *map* function.

Figure 3.4 shows the overall flow of a MapReduce execution. First, the system splits the input data in M partitions and then starts many copies of the user-program on a set of machines. Second, one execution is elected the master, that uses a scheduler to assign the tasks for idle workers based on data locality and network state. It also controls failed tasks by rescheduling them to another worker. Third, the worker executes the *map* function and stores the intermediate key/value pairs in memory. Forth, periodically, the worker flushes the intermediate key/value pairs to its local file system and passes back the location to the master, which is responsible for forwarding these locations to a reduce worker. Fifth, a reducer worker reads all data and sorts them by the intermediate keys. The sorting is necessary since many different keys may map to the same reduce task. Finally, the reduce worker iterates over the intermediate values executing the user-defined *reduce* function and stores the values into the final output file.

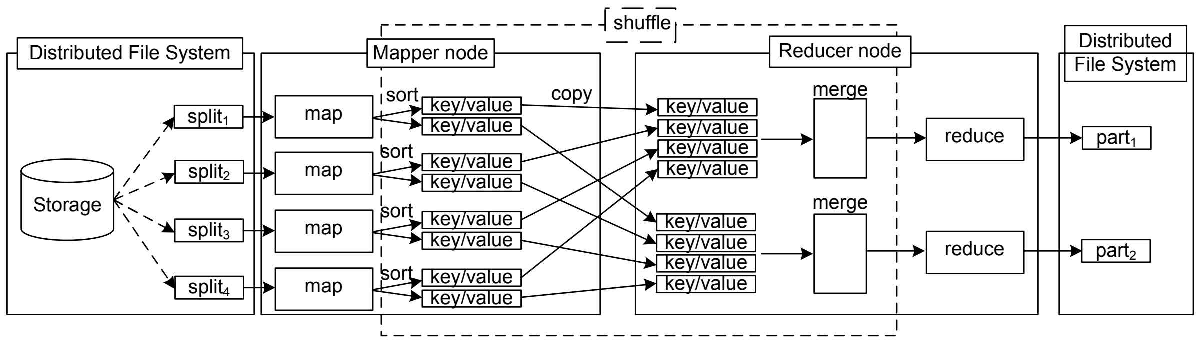


Figure 3.4: MapReduce execution flow [98]

MapReduce tolerates failures of the *map* and *reduce* phases. When a *map* worker fails, even for a completed map, the system re-executes the failed map and notifies all reduce workers' that are executing. This is necessary, because map workers store the result locally. Also reduce tasks that have not already read the data from the failed worker need to be signalized to read from the new worker. On the other hand, completed reduce tasks do not need to be re-executed when a node failure occurs, since their output is stored into a global file system.

MapReduce is in general guided by the following features [98]:

flexibility: the programmers just need to write the map and the reduce functions to process the data, without needing to know how to parallelize a MapReduce job.

scalability: many existing applications face the challenge of scaling when the amount of data increases. When elastic scalability is desired, it requires dynamic behavior to scale up or down as the computation requirements change.

efficiency: MapReduce minimizes data movement, scheduling the tasks to process as close as possible to the data location.

fault-tolerance: thanks to the distributed file system, which keeps replicas of a partition, and the use of stateless functions to operate over the input values, tasks or machines can fail without requiring effort from the programmer. The failures are compensated by re-scheduling the tasks to another machine that can handle the load.

Some limitations of MapReduce are [219]: (a) lack of support for multiple datasets and (b) lack of support for iterative data analysis, requiring data to be loaded at each iteration and demanding an extra MapReduce task to detect termination.

3.2 Cloud Organization

3.2.1 Architecture and Service Model

Several cloud architectures have been proposed in the literature. Generally, the architecture of a cloud computing system can be divided into four layers: hardware, infrastructure, platform and application layers, as shown in figure 3.5.

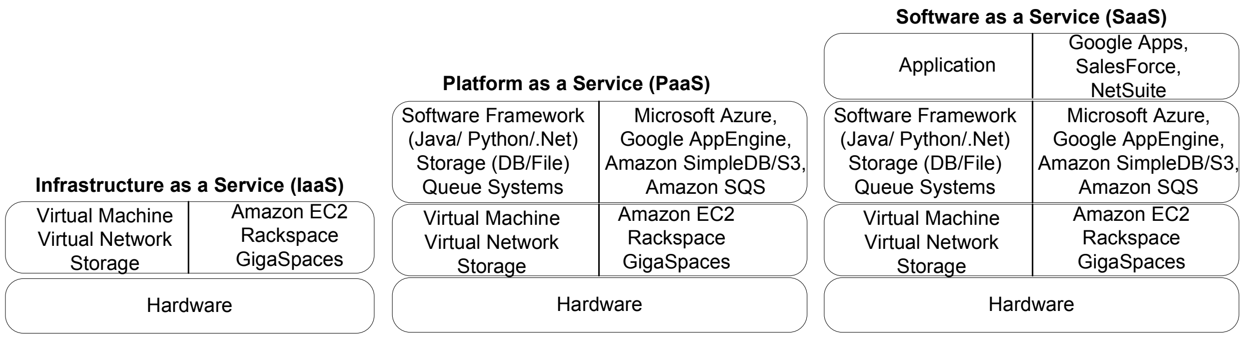

Software as a Service (SaaS)
Application
Google Apps, SalesForce, NetSuite
Platform as a Service (PaaS)
Software Framework (Java/ Python/.Net) Storage (DB/File) Queue Systems
Microsoft Azure, Google AppEngine, Amazon SimpleDB/S3, Amazon SQS
Infrastructure as a Service (IaaS)
Virtual Machine Virtual Network Storage
Amazon EC2 Rackspace GigaSpaces
Hardware

Figure 3.5: Cloud computing architecture [243]

The hardware layer contains the physical resources of the cloud, such as CPUs, disks and networks. It is usually confined in data centers which contain thousands of servers and storage systems interconnected by switches.

The infrastructure layer contains resources that have been abstracted typically by using virtualization techniques (Section 3.1.1), creating a pool of computing resources to be exposed as integrated resources to the upper layer and end users [121]. This layer is an important component of cloud computing, since many features such as elastic resource assignment are made available in this layer [386].

The platform layer consists of application frameworks and a collection of specialized tools on top of the infrastructure layer to provide a development and/or deployment platform aiming to minimize the burden of deploying applications directly into virtual machines containers [121, 386].

The application layer contains the applications that run in the clouds. Different from traditional applications, cloud applications can leverage on automatic scaling to achieve better performance and availability in an on-demand usage.

Usually, a cloud service model is mapped to the cloud architecture. In such case, the cloud service model is divided into three classes, according to the abstraction level and the service model of the providers: infrastructure-as-a-service (IaaS), platform-as-a-service (PaaS), and software-as-a-service (SaaS) [243] (Figure 3.5).

The main difference between these cloud service models relies on the kind of control that the users may have over the cloud infrastructure (Figure 3.6).

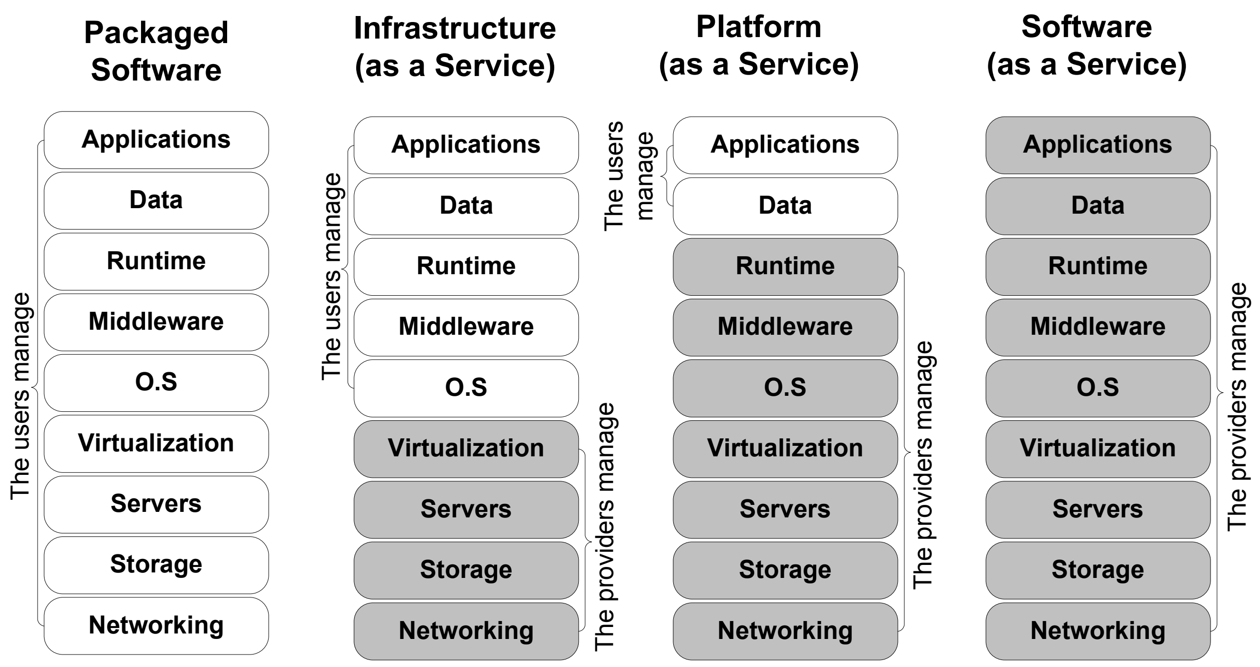


Figure 3.6: Cloud service model considering the customers' viewpoint. Adapted from [243]

In the traditional approach (i.e., non-cloud scenario), the users are responsible for managing the whole stack (e.g., hardware, software, and data center facilities), which gives them full control over the infrastructure.

In the infrastructure-as-a-service (IaaS) model, the users request processing power, storage, network and other computing resources such as the operating system and pay for what they use. The users pay for the use of resources, instead of having to setup them, and deploy their own software on physical machines, controlling and managing them. The amount of instances can be scaled dynamically to fill the users' need. Examples of IaaS providers are Amazon Elastic Compute Cloud (Amazon EC2)–(aws.amazon.com/ec2), Rackspace cloud (rackspace.com/cloud), GigaSpaces (gigaspaces.com), Microsoft Windows Azure (windowsazure.com), and Google Compute Engine (GCE)–(cloud.google.com/compute).

Platform-as-a-service (PaaS) are development platforms that allow the creation of applications with supported programming languages and tools hosted in the cloud and accessed through a browser. This model can slash development time, offering readily available tools and services. PaaS providers offer a higher-level software infrastructure, where the users can build and deploy particular classes of applications and services using the tools and programming languages supported by the PaaS provider. The users have no control over the underlying infrastructure, such as CPU, network, storage or operating system, as it is abstracted away below the platform [243]. Examples of PaaS services are Google App Engine (cloud.google.com/appengine), OpenShift (openshift.com), and Heroku (heroku.com).

In the software-as-a-service (SaaS) model, applications run on the cloud infrastructure and are accessible from various client devices. The users of these services do not control the underlying infrastructure and application platform, i.e., only limited user-configurations are available. The main architectural difference between the traditional software model and SaaS model is the number of tenants the applications support. From the user viewpoint, the SaaS model allows him/her to save money in servers and software licenses. Examples of SaaS are SalesForce (salesforce.com), NetSuite (netsuite.com) and Microsoft Office Web Apps (office.com).

3.2.2 Deployment Model

The cloud deployment model can be classified as [24]: private, public, community, and hybrid.

In a private cloud, the cloud infrastructure usage is restricted to a single organization. The infrastructure may be owned and managed by the organization that will use it or by a third party organization. A private cloud can be built using either in-house solution or third party solutions (e.g., VMware vCloud (vmware.com/products/vcloud-suite)) and open-source solutions (e.g., OpenStack (openstack.org)).

In public clouds, the cloud infrastructure is available for everyone on the Internet. It is often managed by public cloud providers, which allow the customers to use the resources if they pay for them.

In community clouds, the cloud infrastructure is restricted to some organizations that have common interests. Unlike the public cloud, in a community cloud the access is limited to the community members. For example, scientific institutions (e.g., FutureGrid (futuregrid.org), WestGrid (westgrid.ca), and Chameleon (chameleoncloud.org) that can create a community cloud to collaborate and to share resources.

Finally, in hybrid clouds, the cloud infrastructure is composed by two or more distinct infrastructures (e.g., private, community, or public cloud). In other words, each cloud can allocate resources owned by different clouds but they are managed by a unique cloud. For instance, a private cloud can support a hybrid cloud using, in some situations, resources from a public cloud [24, 328]. Hybrid clouds are often used for cloud bursting when the first-choice infrastructure cannot meet the users demand. Cloud bursting is the process of using another type of cloud (e.g., a public cloud) when workload spikes. It allows an application to run in a private infrastructure and to burst into a public cloud in a scenario with high demand of resources. A hybrid cloud can be also achieved by an application that runs across multiple clouds or use different cloud services at the same time. Figure 3.7 show an example of a hybrid cloud scenario. Hybrid clouds can be seen as a type of cloud federation, which will be discussed in section 3.2.3.

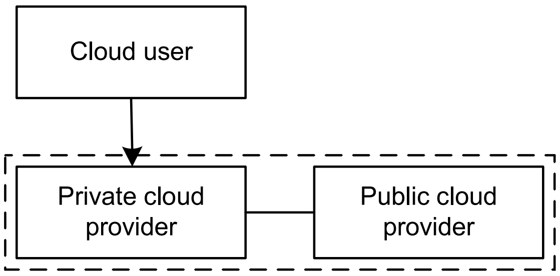


Figure 3.7: Hybrid cloud scenario

3.2.3 Cloud Federation

Although most of the cloud providers claim an infinite pool of resources, i.e., infinite scalability, in practice even the biggest provider may have scalability problems due to the increasing demand of computational resources by the users. In addition, some clouds may have outage problems owing to regional problems such as network partition or due to technical problems such as software bugs, which can make unavailable the service deployed on the cloud [10, 11, 107, 209, 292]. Furthermore, cloud services deployed on a single cloud location may present significant performance degradation (e.g., response time) as the number of users from different locations increases, requiring considerable data transfer. Finally, the cloud providers can limit the number of resources that can be acquired in a period of time.

These issues can be addressed by distributing the services either across different cloud providers or across distinct cloud locations employing distributed coordination as depicted in figure 3.8. This process is known as cloud federation, and it probably represents the next stage of the cloud computing paradigm [56]. A cloud federation aims to increase resource availability in a way similar to what we have in grid computing environments (Section 2.3). On the one hand, cloud federation allows the users to diversify their infrastructure portfolio in terms of both vendors and location. On the other hand, it enables the cloud providers to increase the capacity of their infrastructure on-the-fly, renting resources from other providers in order to meet unpredictable workloads without needing to maintain extra resources in a ready state. Moreover, it also helps them to meet service-level agreements (SLA) (Section 3.1.2).

3.2.3.1 Definition

A cloud federation can be defined as a cloud model that has the purpose of guaranteeing quality of service, such as performance and availability, allowing on-demand reassignment of resources and workload migration through a network of different cloud providers to offer non trivial QoS services for the users, based on standard interfaces, and without a centralized coordination [120].

This definition overlaps with the grid definition (Section 2.3), which also aims to aggregate resources and to increase resource availability. It also does not specify if the clouds collaborate voluntarily or not to create the federation [145]. In a federated environment, each cloud may have its own resource allocation policy which requires negotiation capability of the cloud providers to negotiate the most appropriate resources

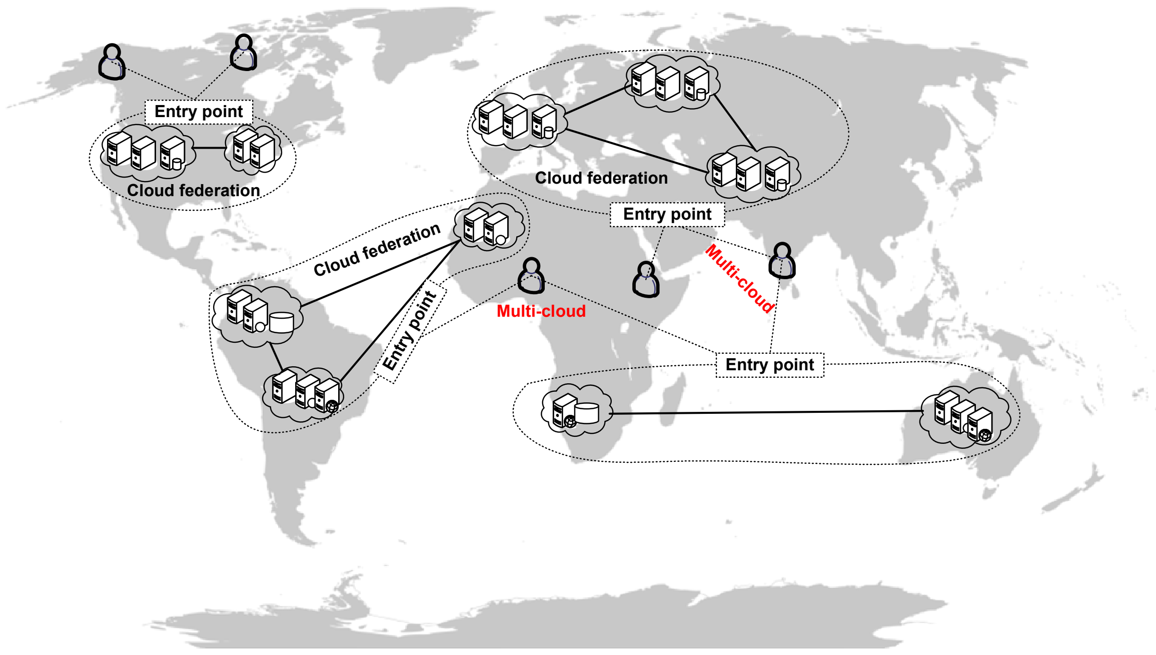


Figure 3.8: An example of the cloud federation approach [145]

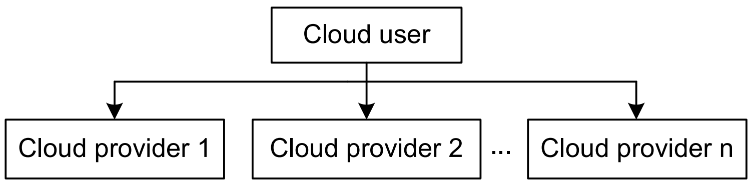
to achieve their goals. These goals are usually described as QoS metrics, such as minimum execution time, minimum price, availability, minimum power consumption and minimum network latency, among others. This requires coordination and orchestration of resources that belong to more than one cloud, which will be used, for instance, to meet the users' demand. Furthermore, the requirements to achieve a cloud federation are [68, 296]:

> *automated and efficient deployment*: cloud federation should support automated provisioning of complex services based on an SLA that specifies the cloud's infrastructure QoS requirements. This SLA should be reusable to provision multiple instances of the services for different users and also be flexible to support customizations according to the user's need. In practice, this requires automatic discovery mechanisms to look for resources in other clouds, automatic matchmaking process to select the cloud that better fits the requirements, and automatic authentication mechanisms to create a trusted context among the clouds.
>
> *dynamic resource allocation*: the cloud infrastructure should autonomously adjust the parameters of each virtual environment (e.g., resources) according to the system's workload aligned with the service-level agreement (SLA).
>
> *technology and provider independence*: the cloud federation should be independent of both technology and cloud provider. It should seamlessly execute services across distributed cloud providers considering the characteristics of each execution environment and also the services' requirements.

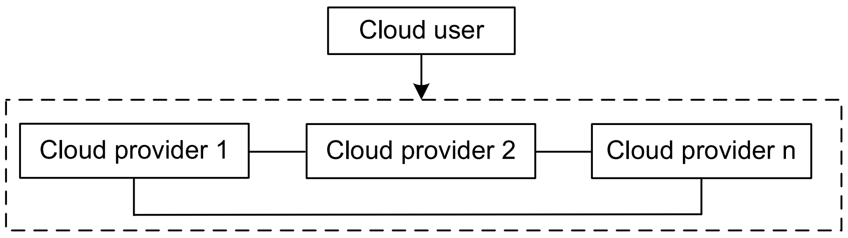
3.2.3.2 Classification

A cloud federation scenario can be classified according to the level of coupling or inter-operations among the clouds [253, 277]. In this context, a federation can be defined as *loosely*, *partially* or *tightly* coupled. In a loosely coupled federation, a cloud service has limited control over the remote resources, monitoring data is limited, and some features such as cross-site networks or migration are not supported. Partially coupled federation allows some level of control over remote resources, as well as monitoring data interchanging or advanced networking features [277]. Finally, a tightly coupled federation comprises clouds that belong to the same organization, with advanced control over remote resources, full access to monitoring data, and advanced networking features.

A cloud federation can be also achieved by combining multiple and independent clouds by a client or service [112], which is denoted as a *multi-cloud* environment (Figures 3.7 and 3.9(a)). In this scenario, the clients or services are responsible to coordinate the clouds and in most of the cases, the cloud providers are unaware of the federation. One of the main issues in a multi-cloud scenario is the portability of applications between clouds, whereas in the traditional federation the concern is about the interoperability between different clouds [189, 278].



(a) Multi-cloud scenario



(b) Federated cloud scenario

Figure 3.9: Difference between multi-clouds and federated clouds

A cloud federation can be categorized as horizontal federation, vertical federation, inter-cloud, cross-cloud, and sky computing as shown in figure 3.10.

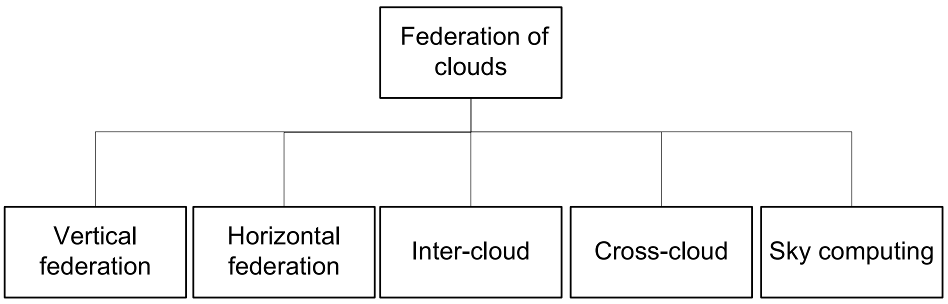


Figure 3.10: Categories of federation of clouds: vertical, horizontal, inter-cloud, cross-cloud, and sky computing [275]

A *vertical federation* occurs when two or more cloud providers join to offer services to different layers (e.g., IaaS, PaaS, SaaS) [357], whereas in a *horizontal federation*, these cloud providers join to offer services at the same cloud layer [297].

Inter-cloud is the federation of clouds that maintains some characteristics such as addressing, naming, identity, security policies, presence, messaging, multicast, time domain, and application messaging [41]. In such case, the clouds are unified based on standard protocols to allow the applications to scale across the clouds. The terms federated clouds and inter-cloud are usually used interchangeably in the literature. However, there are some important differences between these terms. First, inter-cloud is based on standards and open interfaces, while federated clouds use the interfaces provided by the cloud providers [106]. Second, inter-cloud goes beyond resources and services management from multiple clouds that comprises a federation or a multi-cloud scenario, as it includes cloud governance as well as a cloud broker [277]. Cloud governance can be seen as a set of decision making processes, criteria and policies involved in the planning, design, acquisition, deployment, operation and management of cloud computing resources [256]. In this case, federated clouds can be seen as a prerequisite to achieve inter-cloud.

Cross-cloud is characterized by the use of resources owned by an external cloud (i.e., a cloud that does not belong to the federation) in a trusted context [69].

Finally, *sky computing* occurs when multiple clouds are combined to provide new services which are not provided by each isolated cloud. Moreover, sky providers are consumers of other clouds [188]. In other words, sky providers are usually data center-less providers. For example, a PaaS provider can deploy its services in a computing infrastructure offered by an IaaS provider.

Considering the user viewpoint, a cloud federation can be also obtained through a third agent called cloud broker. A cloud broker manages the usage, the performance, and the relationships between the users and the cloud providers [275].

Figure 3.11 illustrates a horizontal cloud federated scenario. In this scenario, there are three clouds, where cloud 1 is renting resources from clouds 2 and 3 to increase the capacity of its infrastructure. In a federation, the resources do not need to move physically across the clouds, but only be logically considered as belonging to another cloud. In other words, the resources continue to be physically placed in the origin cloud, but they are logically considered as resources indeed hosted within the buyer cloud.

3.2.3.3 Challenges

Over the years, the rapid development of cloud computing has pushed to a large market of cloud services usually operated by different proprietary APIs. This has resulted in heterogeneous service descriptions, data representation, and message level naming conflicts making service interoperability (i.e., inter-operation management) and service portability a complex task [275].

Service interoperability and service portability between clouds are key requirements to create a cloud ecosystem and to take full advantage of cloud properties [275]. Service interoperability is the ability of cloud users to use their data and applications across multiple cloud providers with the same management interface, whereas service portability is the ability to move their applications from one cloud provider to another regardless of their choice as operating system, storage format or even APIs [49, 390].

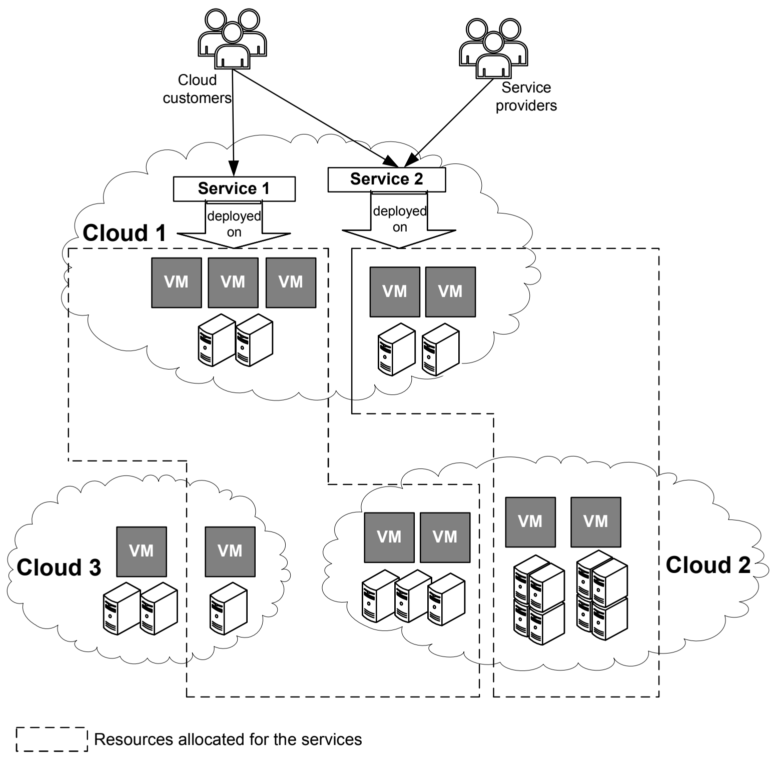


Figure 3.11: A horizontal cloud federated scenario with three clouds. Adapted from [151]

In this context, there are some issues concerning cloud federations such as:

discovery: since each cloud provider employs different languages and formats to describe its services and resources, automatic service/resource discovery is usually a challenge. Service and resource discovery in federated clouds can allow users to select the resources distributed across the clouds based on different objectives. Although there exist some efforts to create standards for resource description (Section 3.3.1), these standards are not supported by most of the cloud providers. Enforcing a standard syntax on resource description is normally difficult due to the dynamic nature of the clouds [140]. For instance, cloud providers are constantly creating new services or adding more attributes to describe them.

selection: appropriate resource selection requires reliable data. However, the lack of reliable data about cloud services performance criteria forces this task to be performed manually, based often only in the user's experiences. Therefore, resource allocation across multiple clouds may benefit from some features such as different geographical locations, lower latency, lower financial cost, and higher availability. In this scenario, automated resource selection is important to improve performance and to reduce financial costs. This can be performed based on either static information (e.g., resource description) or through dynamic QoS negotiation [276].

monitoring: resource monitoring can be used to collect data about the status of computing resources, storage, and network. These data are often required for load distribution or even disaster recovery [16]. In cloud federation environments, resource monitoring is usually a challenge due to the dynamic nature of the environment that can be comprised of heterogeneous infrastructures and resources may reside across different clouds. In such case, monitoring systems must collect and aggregate data provided by the monitored resources despite of the underlying infrastructure [16]. Therefore, most of the monitoring systems such as Ganglia [240], Nagios (nagios.org), and GridICE [13] cannot be used in a cloud federation scenario. Moreover, in a cloud federation environment, when virtual resources are migrated from one cloud to another one, monitoring data needs to be collected in the destination. In this case, the clouds need to support remote monitoring [81]. Furthermore, monitoring systems must be designed to work autonomously. In other words, such monitoring systems must reconfigure themselves when reconfiguration changes (e.g., services or VM migrations, or even resource failure) occur.

3.3 Cloud Standards and Metrics

3.3.1 Cloud Standards

Cloud interoperability requires different standards to work properly at any level of the cloud stack. Nowadays, there exist many organizations working on various standardization efforts for cloud computing. Some of them are the Open Grid Forum (OGF) — (gridforum.org), the Distributed Management Task Force (DMTF), and the Storage Networking Industry Association (SNIA).

This section highlights some standards for cloud computing and their adoption by cloud providers and libraries. Table 3.1 summarizes the standards discussed in this section.

Open Virtualization Format (OVF) is an open standard of the DMTF. This standard defines a descriptor for virtual appliance (VA) independent of both virtualization platform and vendor. In this context, a VA is a complete software stack to be run on virtual machines. In other words, it is the description of the software packages and the required hardware configuration [311]. OVF has been designed to enable the deployment and migration of virtual appliances across different virtualization platforms using a portable format. In this case, it defines management actions for the whole life cycle of a VA such as development, package, distribution, deploy, and retirement. An OVF package comprises an XML file with the description of the VA such as name, hardware constraints, storage, operating system, and network description. This standard is supported by many companies such as Amazon, RackSpace, Google, Microsoft, VMWare, Oracle, IBM, and OpenStack.

The Cloud Infrastructure Management Interface (CIMI) is also a standard proposed by the DMTF. It defines a protocol and a model to govern the interactions between an infrastructure-as-a-service (IaaS) provider and a cloud user. Its model comprises a description of the resources available in an IaaS such as virtual machines, storages, and networks to enable the portability between clouds that support this standard, whereas its protocols defines a RESTful API based on both XML and JavaScript Object Notation (JSON) renderings [94].

The Cloud Data Management Interface (CDMI) (cdmi.sniacloud.com) is a standard proposed by SNIA. This standard defines a RESTful interface for storage management, which is not bound to clouds. It aims to promote data portability among storage providers. In this case, it specifies functions that (i) allow users to discover the capabilities in a data service offering; (ii) manage containers (e.g., a directory within a file system) and data that are placed in them; and (iii) associate metadata for containers and objects. In addition, CDMI offers a set of protocols that enable the users to perform other storage operations such as monitoring, billing, and access management independent of provider. The main feature of CDMI is that it allows users to understand the capabilities of the underlying storage and data services through a homogeneous interface. Moreover, many storage companies such as NetApp and EMC[2] are compliant with this standard.

Open Cloud Computing Interface (OCCI) was proposed by OGF to create a remote management API for IaaS-based services. Later, it has evolved to support all the cloud service models (Section 3.2.1). OCCI aims to guarantee interoperability among multiple cloud providers. In such context, cloud providers should work together without data schema or format translation between their APIs. In other words, the OCCI API acts as a service front-end for controlling cloud resources [390]. OCCI is compatible with other cloud standards such as OVF and CDMI. Moreover, it also works as an integration point for standardization among various working groups and organizations [344]. Actually, the OCCI specification consists of three documents, namely OCCI core, OCCI Renderings, and OCCI Extensions. The OCCI core defines the instance type of the OCCI model. Basically, it provides an abstraction of cloud resources. The OCCI Rendering, on the other hand, defines how the OCCI core model is rendered over the HTTP protocol to lead to a RESTful API. Finally, the OCCI Extensions define some OCCI infrastructure extensions for IaaS domain.

Furthermore, many libraries implement the OCCI model such as Apache CloudStack (cloudstack.apache.org), OpenNebula [248], CompatibleOne [378], and jclouds (jclouds.apache.org).

Table 3.1: A summary of some cloud standards

Standard	Group	Focus	Key operations
OVF	DMTF	Standard description of virtual appliances	Develop, package, distribution, deploy, and retirement
CIMI	DMTF	Portability and resource management in the IaaS domain (RESTful API)	Resource management
CDMI	SNIA	Data portability	Common file system operations (i.e., read, write), backup, replication, and billing
OCCI	OGF	Service interoperability (RESTful API)	Deployment, monitoring, and auto-scaling

3.3.2 Cloud Metrics

The growing interest in cloud computing has demanded metrics to support the users on selecting a service considering different criteria. Cloud metrics is also required since the providers often focus on resource availability without performance guarantees [20, 170, 257], which makes difficult for users to select the appropriate cloud. This is mostly owing to the cloud architecture that is based on a co-location model designed to minimize the costs for providers [392]. In this scenario, cloud evaluation must rely on different characteristics such as bandwidth, throughput, latency, computing and storage capacities as well as on resource availability and efficiency. Different works [137, 203, 206, 224, 237, 318] have been proposed to evaluate a cloud service. Based on these works, this subsection presents some metrics to evaluate IaaS services. Table 3.2 shows the metrics discussed in this section, which are classified as *general* or *specific* metrics. *General metrics* are metrics that can be used to evaluate any cloud service and/or resource, whereas *specific metrics* are metrics bound to a specific cloud service and/or resource.

In table 3.2, the *instance sustainable performance (ISP)* metric measures the effective average performance of an instance (i.e.,VM) to execute a particular application [205]; and it is computed as the geometric mean of the Flop rate of an application. The *instance capability (IC)*, on the other hand, represents the sustained performance of the instance integrated over a time period. This metric can be also used to assess the price performance of the instance dividing it by the cost of the instance. In other words, this metric can help to determine the instance that delivers the best value out of all the available instances.

3.4 IaaS Cloud Computing Systems

This section describes the architecture of an IaaS cloud. First, its components are discussed. Then, the steps to use a service offered by this architecture are presented. Finally, some IaaS architectures are presented and compared.

3.4.1 Architecture

IaaS clouds comprise a set of physical resources offering virtualized resources for the users on-demand in nearly real-time. In this context, the users are responsible to request the virtual resources (e.g., virtual machines) and to manage them, whereas the providers are responsible (a) to select a physical resource to host the virtual machines; (b) to create/instantiate the virtual machines; (c) to monitor the hosts (i.e., physical machines) to guarantee their availability, and (d) to charge for their usage.

A generic IaaS cloud architecture (Figure 3.12) relies on a *virtualization* layer (Section 3.1.1).

Above the *virtualization* layer, there is a *virtual infrastructure (VI)* manager. The *VI* manager creates the virtual resources (e.g., virtual machines, storage) aggregating a pool of physical resources often regardless of the underlying *virtualization* layer. In addition, it can implement more advanced functions such as automatic load balancing, server consolidation, and dynamic infrastructure resizing and partitioning [328]. In general, VI managers can be grouped into two categories [328]: cloud toolkit and non-cloud toolkit. Cloud toolkits expose a remote interface to create, to monitor, and to control the virtualized resources, but they lack some management functions such as server consolidation. The non-cloud toolkits, on the other hand, provide advanced features such as automatic load balancing and server consolidation without exposing remote cloud-like interfaces. Examples of VI managers include Eucalyptus [263], Nimbus [186], OpenNebula [248], and OpenStack (openstack.org).

The services are built on top of the *virtual infrastructure (VI)* manager, and organized within the *service* layer. The services include virtual machines, storage, networking, and virtual machine images (VMI).

Table 3.2: Some metrics to evaluate an IaaS cloud

Metric name	**Formula**	**Scope**	**Type**	**Unit**
Mean time to resource acquisition (MTRA)	$\frac{\text{provisioning time}}{\text{boot time}}$	General	Cloud	s (seconds)
Outage duration (OD)	outage end time – outage start time	General	Cloud	m (minutes)
Cloud availability (CA)	$\frac{\text{uptime}}{\text{total time}}$	General	Cloud	%
Allocation trust (AT)	$\frac{\text{number of resources provisioned}}{\text{number of resources available}}$	General	Cloud	%
Cost-effectiveness (CE)	$\frac{\text{cost}}{\text{performance improvement}}$	General	Service	%
Resource release time (RRT)	$\frac{\text{time to stop}}{\text{time to remove}}$	General	Resource	s
Instance efficiency (IE)	$\frac{\text{CPU peak}}{\text{CPU power}}$	Specific	Instance	%
Instance performance trust (IPT)	$\frac{\text{acquired performance duration}}{\text{uptime}}$	Specific	Instance	%
Instance sustainable performance (ISP)	Geometric mean [205]	Specific	Instance	Flops/s
Instance capability (IC)	$\frac{ISP}{\int_{t_i}^{t_j} P(t)dt}$	Specific	Instance	MFlops/$
Instance ECU ratio	$\frac{\text{CPU power in GHz}}{\text{1.2 GHz}}$	Specific	CPU	ECU
Resource efficiency (RE)	$\frac{\text{expected performance}}{\int_{t0}^{t1} P(t)dt}$	Specific	CPU	MFlops/$
Storage capacity (SC)	$\sum_{i=1}^{n} \text{storage size(i)}$	Specific	Storage	GB
Storage data access ratio (SDAR)	$\frac{\text{number of operations}}{\text{measured time}}$	Specific	Storage	Bits/s
Network capacity (NC)	$\frac{\text{Amount of data transmitted}}{\text{measured time}}$	Specific	Network	MBits/s
Packet lost ratio (PLR)	$\frac{\text{number of packet lost}}{\text{number of packets}}$	Specific	Network	%

Computing services are provided as virtual machines (VM) instances. A virtual machine belongs to an instance type that determines its hardware capacity such as the number of CPU cores (i.e., vCPUs), the amount of RAM memory, the disks' constraints, and the network capacity. In addition, instance types are divided into different family types or classes defined according to their purpose usage such as memory-, CPU-, or I/O-optimized, among others. For instance, at the time of writing this thesis, Amazon EC2 offers seven family types (general purpose, standard, compute optimized, memory optimized, GPU, storage optimized, and HS1)[1] and Google Compute Engine offers four family types (small or shared-core, standard, high CPU, high memory)[2].

Storage is provided as regular disks for the virtual machines and it can be (Figure 3.13): instance disk, network disk, or object storage. Instance disks (Figure 3.13(a)) are ephemeral disks physically attached to the host computer of a virtual machine. Network disks or block storages (Figure 3.13(b)) are persistent disks that can be attached to a running virtual machine that is in the same data center (i.e., availability zone). Object storages (Figure 3.13(c)), on the other hand, are external storage services that can be mounted in a running virtual machine. Moreover, storages belong to a storage type that determines their performance and constraints. An example of a storage type is the solid-state disk (SSD). Finally, storage is often billed per size in a period of time, and some may have addition charges such as the number of transactions or provisioned input/output operations per second (IOPS).

Networking services provide network connectivity between the virtual resources and other networks (e.g., Internet); and their costs are usually defined in function of the amount of data transfered between the networks.

Virtual machine images (VMI) are provided as persistent disks that contain an operating system and a root file system required to start a virtual machine. They can be configured to run on a specific hypervisor (i.e., virtualization technique (Section 3.1.1.2)), and their price often depends on the operating system. Moreover, the providers allow the users to: (i) upload their own images to the cloud; (ii) create a new one based on one of their virtual machines (VMs), i.e., to take a snapshot of a VM; or (iii) export an image. In this case, the users are usually billed per transaction (i.e., import/export, snapshot) and per the amount of storage used to store a snapshot and/or a VMI.

The *client* layer comprises the applications and the APIs available to access the services. In most of the cases, the users have to create their own applications to access the services through the application programming interfaces (APIs) provided by the clouds.

3.4.2 Using an IaaS Cloud Service

The steps to use an IaaS cloud service are: (1) choose a region to host the resources; (2) select an instance type; (3) select a VMI; (4) select a disk type; (5) request the provider to create the VM; (6) configure and monitor the VM; (7) execute the application; and (8) transfer the data from the cloud to the users local machine.

1. **selecting a region**: a region is an independent cloud environment deployed in a geographic location in order to increase availability, i.e., to meet users' SLAs and

[1] aws.amazon.com/ec2/instance-types

[2] cloud.google.com/compute/docs/machine-types

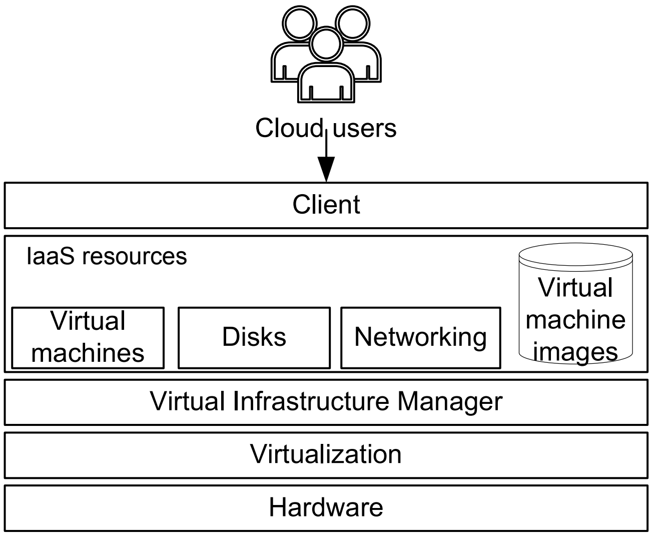


Figure 3.12: A generic IaaS architecture

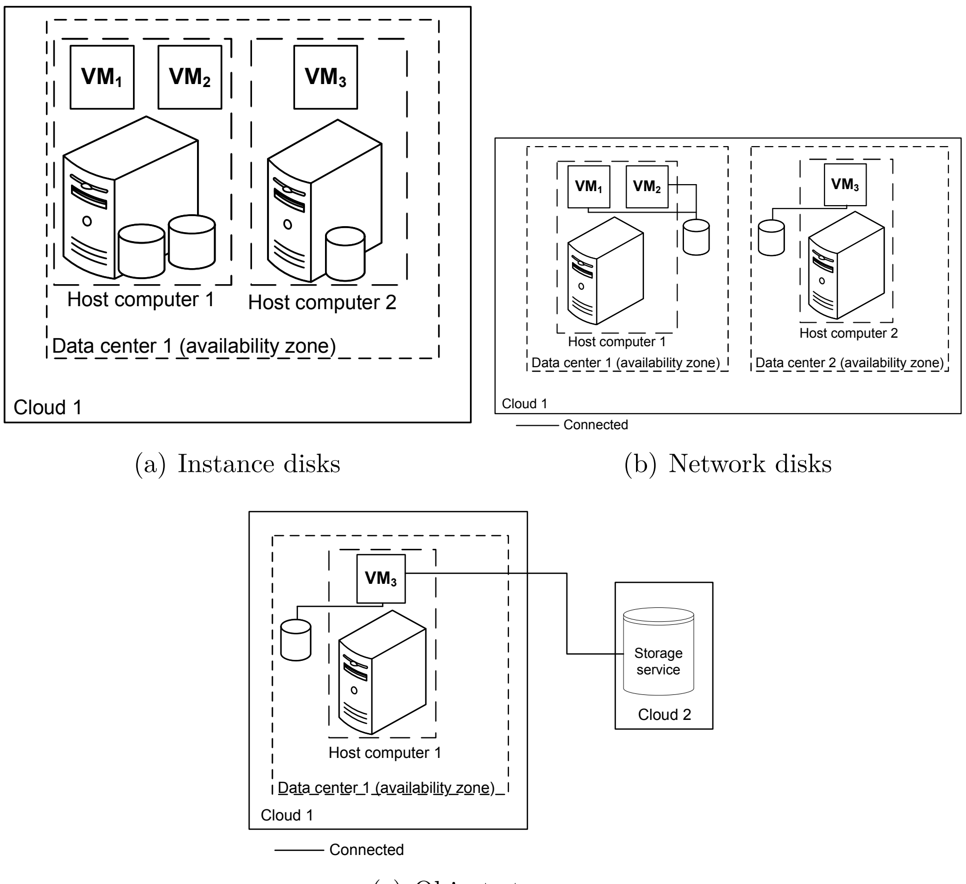


Figure 3.13: Storage types usually available in IaaS clouds

to decrease network latency. A region may have multiple availability zones, or for short, zone. Availability zones are distinct data centers engineered to be decoupled from failures in other zones; and to have high-network bandwidth and low-latency network connections to other zones in the same region. The selection of a region depends on the resources needed by the users and on the cost they want to pay for, i.e., the users' requirements. If a region has more than one data center (i.e., availability zone), the users may select one or they may delegate this task for the cloud provider.

2. **selecting an instance type**: in this step, the users must select an instance type based on the requirements of their computing environment, such as the minimum number of CPU cores and the minimum amount of memory; and on the characteristics of their applications (e.g., CPU- or memory-bound). Usually, there are several instance types that meet the users constraints, and the users usually select one in an ad-hoc way, which can lead to a high cost due to the mismatch between the instance purpose usage and the application behavior.

3. **selecting a virtual machine image (VMI)**: after choosing the instance type, a virtual machine image must be selected. This selection depends on the operating system and in some cases, on the instance type. An instance type can restrict the usage of the images due to the hypervisor or virtualization technique required by the instance.

4. **selecting a disk type**: as we said in the previous section, each disk type implies different performance, costs, and constraints. The selection of a disk type depends on the characteristics of the applications or on some I/O requirements. As in the selection of an instance type, this process is realized manually by the users.

5. **requesting a VM**: after choosing the region, the instance type, the VMI, and the disk type, the users request the provider to create the VM. Then, the provider creates the requested VM through its *virtual infrastructure (VI)* manager (Figure 3.12). Normally the steps to create a VM are: (a) select a physical machine to host the VM; (b) create the disks to the new VM; (c) copy the virtual machine image (VMI) to the select host; (d) request the hypervisor (*virtualization layer*) to start the virtual machine; (e) contextualize the VM [187]; i.e., assigns an IP address, imports the access keys, among other configurations; (f) test the state of the VM; and (g) start to charge for its usage.

6. **configuring the VM**: in this step, the users connect and install their applications and libraries in the virtual machine, as well as transfer all data from their local machine to the cloud.

7. **executing the applications**: the users execute their applications and wait them to finish. During this execution, the users must monitor both the virtual machine and the application execution.

8. **releasing the VM**: in this step, the users transfer the output of the applications to their local machine and terminate the virtual machine.

If the users want to deploy their application in more than one VM, they should restart from the second step.

3.5 Cloud Computing Architectures

Over the years, many cloud computing systems have been proposed to manage and to create an IaaS cloud. This section presents some of these systems that aim to manage a cloud. They are organized as centralized or decentralized systems.

3.5.1 Centralized Systems

In this section, we present ten centralized cloud architectures that are in the same domain of this doctoral thesis. The architectures are considered centralized as the decisions to access the resources rely on a single node. Other centralized cloud architectures can be found in [18, 128, 147, 180, 238, 280].

3.5.1.1 Claudia

Claudia [299] is a software layer that provides homogeneous interface for cloud service management (Figure 3.14). For service management, Claudia uses an extension of the OVF standard (Section 3.3.1), called service descriptor file, to allow the providers to describe their services independent of technology or API, and to specify the dependencies between the services.

Claudia implements auto-scaling services based on scalability rules defined by the providers for each service in the descriptor file. Such rules define the conditions and the actions to be executed by the system. A condition can be defined basing on user-defined metrics or on hardware metrics. In this case, scalability rules encompass scaling resources up/down in the same cloud provider as well as scaling in/out the number of resources across multiple clouds.

Claudia uses OpenNebula [248] to manage the underlying infrastructure and to access multiple clouds. In other words, the cloud interoperability is achieved through the use of OpenNebula. In this case, the architecture is deployed in a single node (i.e., the master node) that is responsible for managing the clouds through the cloud infrastructure manager (i.e., VI manager).

Experimental results show that Claudia could scale the jobs of a Sun Grid Engine (SGE) queue according to the scalability rules described in the descriptor file.

3.5.1.2 SciCumulus

SciCumulus [95] is a middleware to automate the execution of workflows on the cloud following the many task computing (MTC) paradigm. It explores parameter sweep and data fragmentation parallelism in workflows. In such case, SciCumulus aims to explore workflow parallelism in the cloud focusing on implementing provenance services, to free the users to deal with the complexity of collecting distributed provenance data [95].

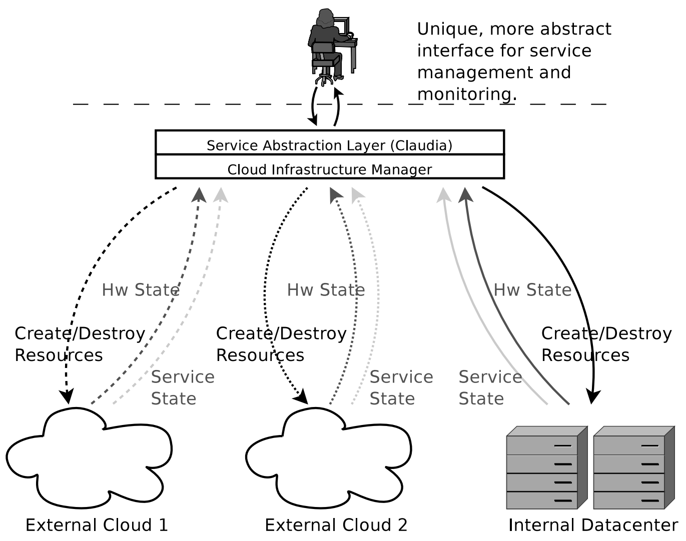


Figure 3.14: Claudia architecture [299]

The architecture of SciCumulus is organized in three layers: desktop, distribution, and execution layer, which are placed between a workflow management system (WfMS) and the cloud infrastructure (Figure 3.15). The desktop layer uses the WfMS to start the execution of the workflow's activities (steps 1 and 2). The distribution layer manages the execution of cloud activities by recognizing the tasks that compose the workflow and using a global schema to schedule them in the cloud (**??** – steps 3.1, 3.2, 4, 5, and 6). A cloud activity is a container with the application to execute, its execution strategy, its parameter values, and its input data. For each assigned instance pair (instance, cloud activity), the execution layer configures and executes the application (**??** – steps 8.1, 8.2, 9, 10, and 11).

SciCumulus was validated using the CloudSim [58] simulator to evaluate the execution time of a workflow when fragmented in 100 and 128 cloud activities. For the workflow analyzed, the results show performance gains when the number of cloud activities is at most 100, otherwise the performance degrades due to the overhead of managing the activities.

3.5.1.3 Cloud-TM

Cloud-TM [300] is a cloud architecture to build and to execute applications based on distributed transaction memory (DTM). It aims to combine the transactional memory (TM) approach with the scalability and failure resilience of redundant large-scale infrastructures.

Its architecture comprises two components: data platform and autonomic manager (Figure 3.16). Data platform exposes an API for manipulating data across distributed servers, and it implements the distributed software transaction memory (DSTM) services. The DSTM is based on Infinispan (infinispan.org). Infinispan implements automatic data partitioning across multiple nodes, enabling continuous data availability and transactional integrity even in the presence of failures. The autonomic manager, on the other hand, is responsible for scaling the data platform and for implementing self-optimizing strategies to reconfigure the data grid. In such case, there is a monitoring component that gathers data about the resources and sends them to a workload analyzer. Then, the workload analyzer filters the data, generates a workload profile, and notifies the adaptation manager.

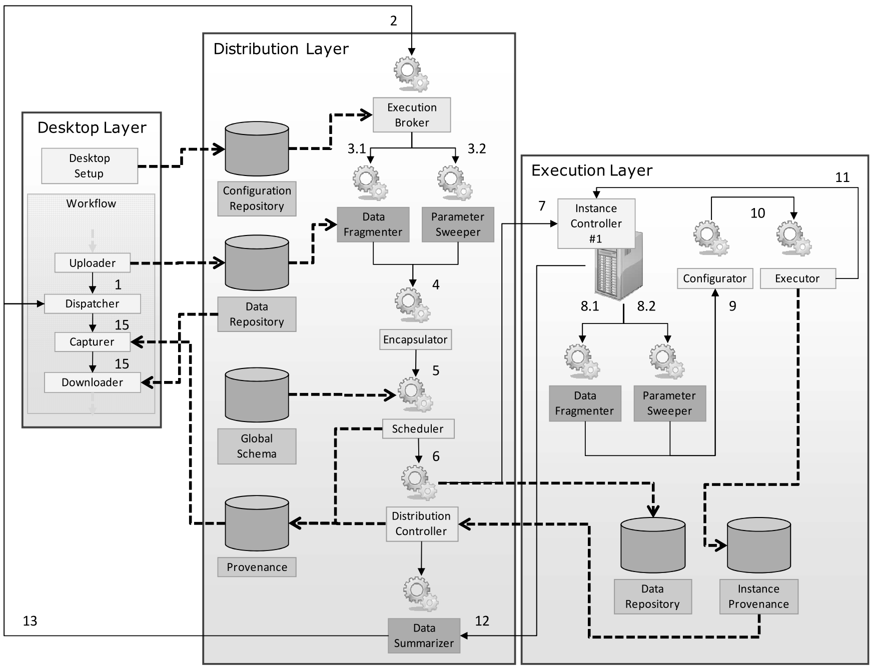


Figure 3.15: SciCumulus architecture [95]

Finally, the adaptation manager reconfigures the system to meet performance and cost requirements.

Experimental results show that changing a distributed software transactional memory solution, called D^2STM[87], to use Cloud-TM helped it to achieve linear speed-ups.

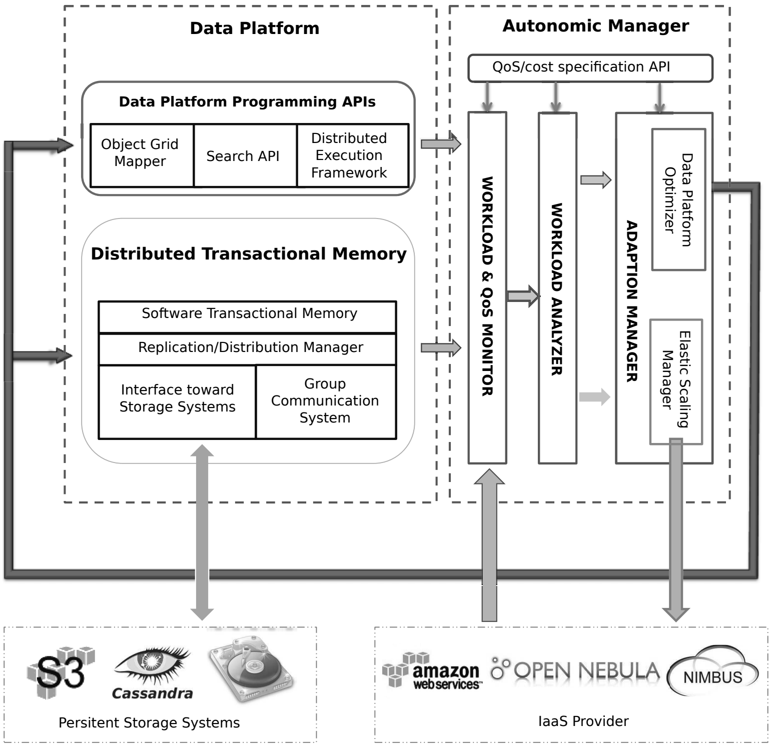

Data Platform
Data Platform Programming APIs
Object Grid Mapper
Search API
Distributed Execution Framework
Distributed Transactional Memory
Software Transactional Memory
Replication/Distribution Manager
Interface toward Storage Systems
Group Communication System
Autonomic Manager
QoS/cost specification API
WORKLOAD & QoS MONITOR
WORKLOAD ANALYZER
ADAPTION MANAGER
Data Platform Optimizer
Elastic Scaling Manager
S3
Cassandra
Persitent Storage Systems
amazon webservices
OPEN NEBULA
NIMBUS
IaaS Provider

Figure 3.16: Cloud-TM architecture [42]

3.5.1.4 mOSAIC

mOSAIC [278, 279] is an API and platform to create cloud-aware applications to run on hybrid clouds. Unlike other cloud APIs that try to abstract the cloud resources providing a uniform API and keeping the programming style closed to the native cloud API, mOSAIC assumes some characteristics of the deployed applications such as: (i) the applications are divided into components with explicit dependencies in terms of both communication and data among them; (ii) the applications use only the mOSAIC API for inter-component communication; and finally, (iii) for each component, the developers specify its requirements such as CPU and storage.

The architecture of mOSAIC comprises five major components (Figure 3.17): resource broker, cloud agency, client interface, application executor, and semantic engine. The resource broker mediates the interaction between the clients and the cloud providers. The cloud agency implements resource discovery and negotiates resource usage with the cloud providers. It is also used to monitor and to reconfigure the resources according to the performance constraints. The client interface, on the other hand, is responsible for requesting additional resources from an application executor. The application executor deploys the applications and monitors their execution. Finally, the semantic engine helps the developers on identifying a cloud for their applications. In such case, the developers semantically describe and annotate their applications, and specify related concepts and patterns using an ontology provided by mOSAIC.

mOSAIC has been used to deploy different applications developed in Java, Python, and Erlang.

3.5.1.5 TClouds

TCloud [355] is a cloud architecture that implements automated attacks detection and failure recovery services on top of the cloud infrastructure. To achieve the security levels required by the applications, TCloud adds a software layer between the applications and the clouds (Figure 3.18). This security layer clusters the resources (e.g., storage, virtual machines) as trusted or untrusted, and intermediates the access to them. The idea is that the resources classified as trusted are both resilient to attacks and to Byzantine failures. In the same manner, a platform service is implemented to represent the trustworthy services built on top of the trusted or untrusted cloud infrastructure.

TCloud was validated through a modified version of Apache Hadoop with support for Byzantine failures, running on top of the infrastructure of a public cloud, i.e., an untrusted IaaS infrastructure.

3.5.1.6 FraSCAti

FraSCAti [273] is a platform for deploying applications on multiple clouds. This platform relies on three concepts: (i) an open service model, (ii) a configurable architecture, and (iii) some infrastructure services.

The open service model is used to design and to build multi-cloud PaaS and SaaS applications that run on top of FraSCAti. This model follows a service component architecture (SCA) to define the services provided by the federated PaaS and by the SaaS applications. In this case, services are defined independent of programming languages,

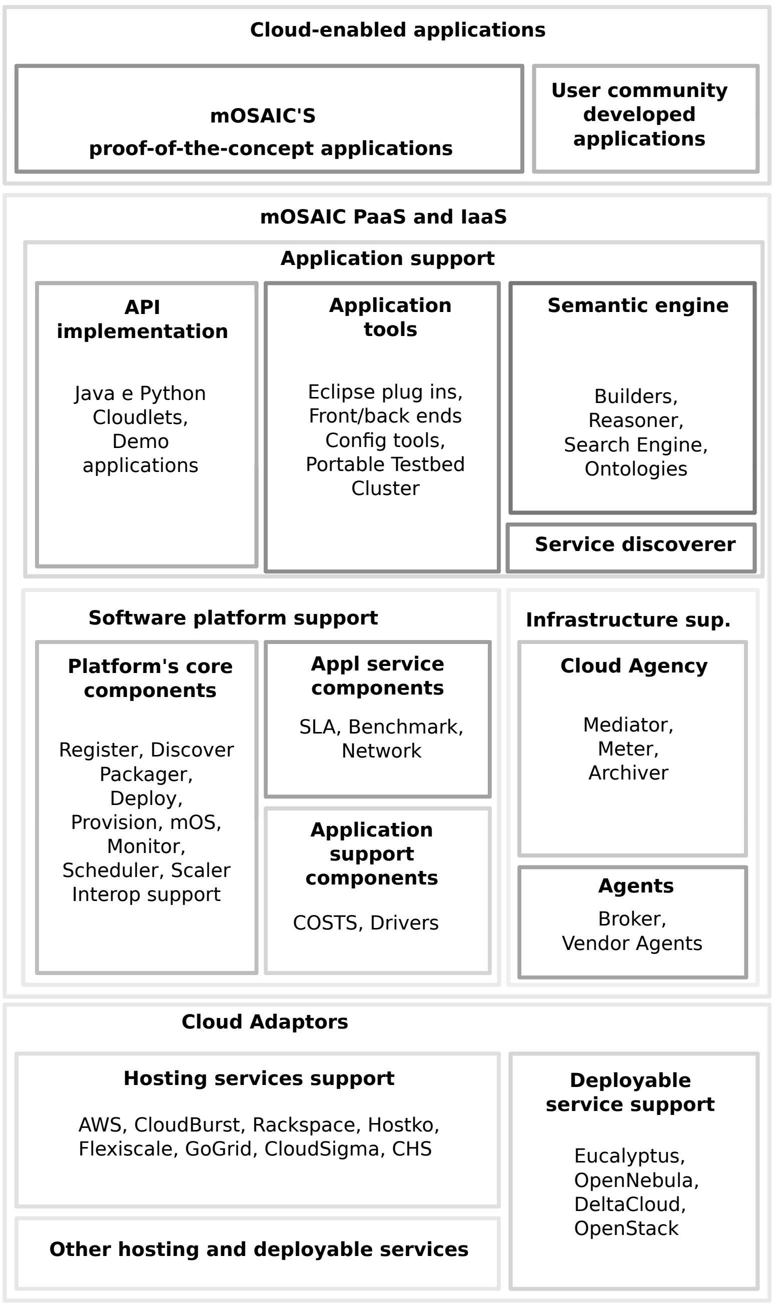

Cloud-enabled applications
mOSAIC'S proof-of-the-concept applications
User community developed applications
mOSAIC PaaS and IaaS
Application support
API implementation
Java e Python Cloudlets, Demo applications
Application tools
Eclipse plug ins, Front/back ends Config tools, Portable Testbed Cluster
Semantic engine
Builders, Reasoner, Search Engine, Ontologies
Service discoverer
Software platform support
Platform's core components
Register, Discover Packager, Deploy, Provision, mOS, Monitor, Scheduler, Scaler Interop support
Appl service components
SLA, Benchmark, Network
Application support components
COSTS, Drivers
Infrastructure sup.
Cloud Agency
Mediator, Meter, Archiver
Agents
Broker, Vendor Agents
Cloud Adaptors
Hosting services support
AWS, CloudBurst, Rackspace, Hostko, Flexiscale, GoGrid, CloudSigma, CHS
Other hosting and deployable services
Deployable service support
Eucalyptus, OpenNebula, DeltaCloud, OpenStack

Figure 3.17: mOSAIC architecture [279]

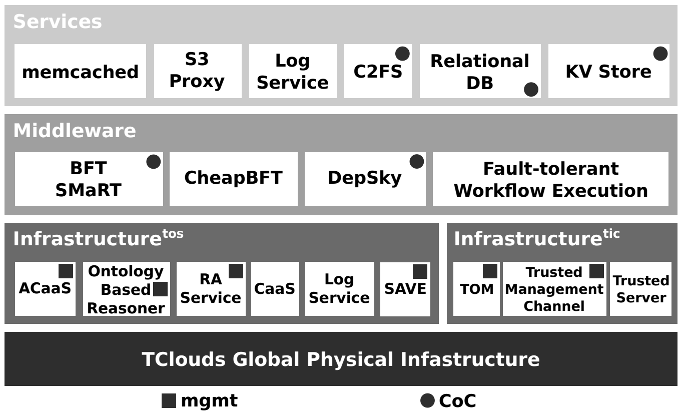


Figure 3.18: TClouds architecture [355]

protocols, and non-functional properties. In other words, SCA providers a framework that addresses portability, interoperability, and heterogeneity of service-oriented systems [271].

The configurable architecture employs software product line (SPL) to capture the characteristics and variability points of the cloud environments. In this case, the SPL is implemented as an assembly of SCA components; and the variability points are captured as a set of plug-ins (Figure 3.19). This enables the developers to meet the constraints of the target cloud environment.

Infrastructure services, on the other hand, are management functions such that deal with node provisioning and application deployment.

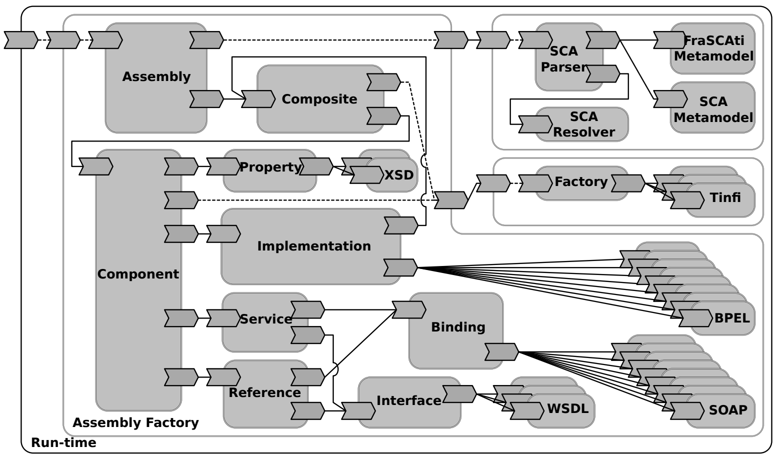


Figure 3.19: FraSCAti architecture [273]

FraSCAti was used to implement a P2P monitoring application composed of thirteen peers, each one deployed on a different cloud. In this scenario, FraSCAti met the constraints of each cloud environment due to the usage of the SPL.

3.5.1.7 STRATOS

STRATOS [274] is a cloud broker architecture for resource allocation on multiple clouds. The clouds are selected at runtime according to some objectives. In such case, the objectives are specified in terms of key performance indicators (KPIs). Thus, when a request for a resource acquisition is made the broker considers it against all the providers.

In STRATOS, the users describe the execution environment using an XML file (i.e., topology description file) and submit it to a cloud manager (Figure 3.20). A description includes functional and non-functional requirements such as the number of nodes, the metrics to monitor, and the management directives. After receiving the topology file, the cloud manager contacts the broker to instantiate the environment. Then, the broker selects the providers that meet the users' objectives; and it uses the translation layer to creates the instances in the selected providers. The monitoring component is responsible to monitor the metrics specified at the descriptor file and to send their data to both cloud manager and broker.

Experimental results show that STRATOS could reduce up to 48% of the financial cost, using both small and large instances of Amazon EC2 and Rackspace.

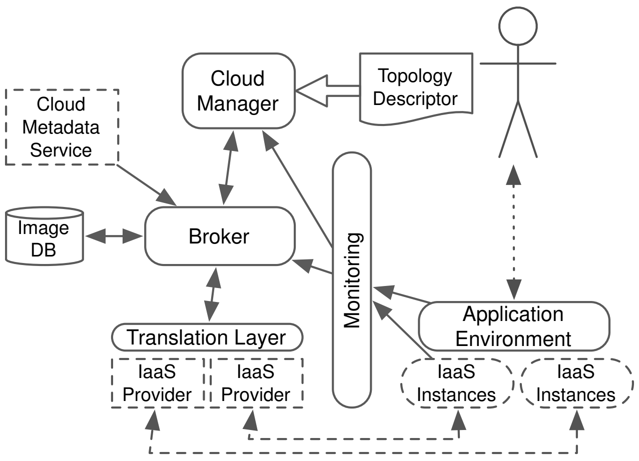


Figure 3.20: Stratos architecture [274]

3.5.1.8 COS

COS [165] is a cloud architecture to support application scaling based on migration. In COS, the applications must be developed following the actor-based model and the SALSA [354] programming language. In this case, according to the workload the architecture migrates the applications' components (i.e., actors) taking into account communication and migration costs. In other words, there is a manager that continually receives the CPU usage of the other nodes, and based on their usage, it decides either to create a new VM or to consolidate the virtual machines.

The experiment results show that using actor-based model decreases up to 14% the application migration time, when comparing to the migration time of the whole virtual machine.

3.5.1.9 Rafhyc

Rafhyc [245] is a cloud architecture to create PaaS and SaaS services over federated cloud resources. Its architecture (Figure 3.22) is organized in layers, namely Rafhyc

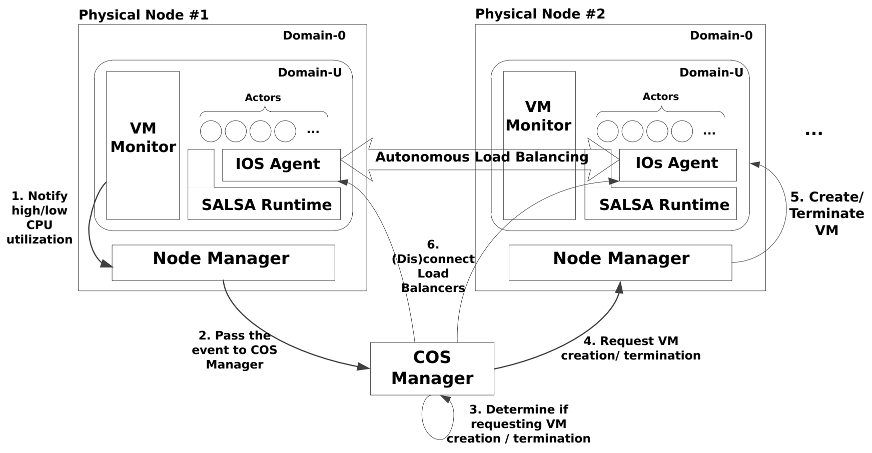

Physical Node #1
Domain-0
Domain-U
Actors
VM Monitor
IOS Agent
SALSA Runtime
Node Manager
Physical Node #2
Domain-0
Domain-U
Actors
VM Monitor
IOs Agent
SALSA Runtime
Node Manager
Autonomous Load Balancing
1. Notify high/low CPU utilization
2. Pass the event to COS Manager
3. Determine if requesting VM creation / termination
4. Request VM creation/ termination
5. Create/ Terminate VM
6. (Dis)connect Load Balancers
COS Manager

Figure 3.21: COS architecture [165]

Resilient Layer and Rafhyc Services Layer. The Rafhyc Resilient Layer provides services for executing the applications in the cloud. The Rafhyc Services Layer provides the services to access and to manage the infrastructure. Moreover, Rafhyc has a configuration manager responsible for monitoring the resources and the workload of the virtual machines. In practice, the configuration manager continuously executes a benchmark in the virtual machines to compute the efficiency and reliability of the clouds.

The experiments realized in a federated environment show that Rafhyc can manage the execution of applications in a dynamic and heterogeneous environment. In this case, the experiments focused on analyzing performance variabilities between the clouds.

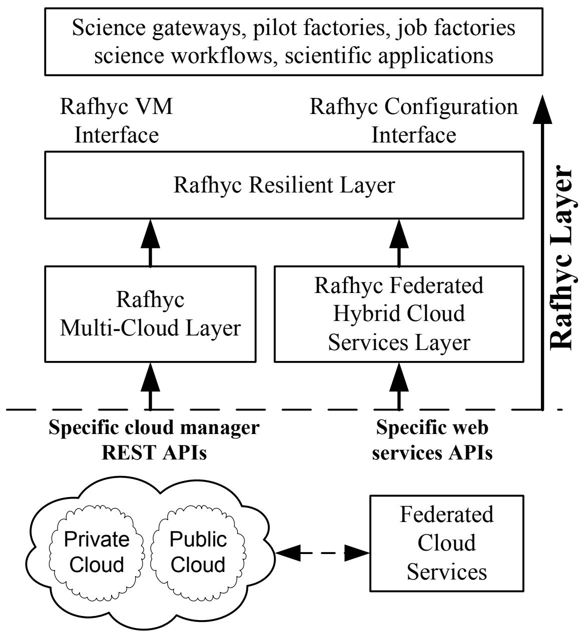

Science gateways, pilot factories, job factories science workflows, scientific applications
Rafhyc VM Interface
Rafhyc Configuration Interface
Rafhyc Resilient Layer
Rafhyc Multi-Cloud Layer
Rafhyc Federated Hybrid Cloud Services Layer
Rafhyc Layer
Specific cloud manager REST APIs
Specific web services APIs
Private Cloud
Public Cloud
Federated Cloud Services

Figure 3.22: Rafhyc architecture [245]

3.5.1.10 JSTaaS

JSTaaS [218] is a framework for enabling cloud bursting. Using an annotative strategy, the developers can create multi-cloud solutions, executing their applications across multiple clouds. In this case, JSTaaS (Figure 3.23) reads the annotations in the source code and generates a new code with the instructions to create the virtual machines and to monitor the applications. It implements cloud bursting, intercepting every method invocation at runtime, and deciding in which node the method must be executed. In case of remote execution, the framework uses a queue to schedule the execution of the method in another node.

An experiment in a hybrid cloud scenario shows an increasing in the execution time due to the network latencies between the clouds (i.e., private and public clouds).

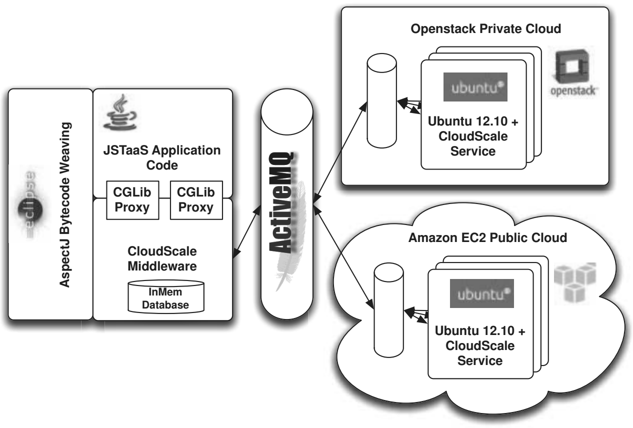


Figure 3.23: Cloud bursting architecture [218]

3.5.2 Decentralized Systems

In this section, we present five decentralized cloud architectures that are most related to this PhD thesis.

3.5.2.1 Reservoir

Reservoir [296] is an architecture to manage services defined as a collection of virtual execution environments (VEEs). Each VEE can spread over multiple sites, where each site is completely autonomous. The architecture of Reservoir comprises three elements: service manager (SM), VEE manager, and VEE host (VEEH) (Figure 3.24). The service manager is in the highest level of the architecture, and it is responsible for deploying, provisioning, and monitoring the VEEs. The VEE manager is responsible for placing the VEEs into the hosts and also for adding or removing VEEs from a given VEE group. Finally, the VEE host is responsible for managing the virtualization platform and for monitoring the VEEs. In addition, it is also responsible to allow VEEs migrations across the sites.

In Reservoir, a service acts as a platform to execute the applications in the VEEs. In this case, the developers describe the requirements of their applications using a service definition manifest. A service definition manifest includes the virtual machine images required to run the applications, as well as the elasticity and service-level agreement (SLA) rules. This service manifest is submitted to a service manager (SM) that selects the appropriate resources and instantiates the VEEs for each component defined in the manifest. In addition, the SM also monitors the execution of the applications to scale them according to the elasticity rules and to meet the SLAs. It scales the applications by adjusting the service capacities, either adding more service components or changing the resource requirements of a component according to its load.

In practice, the service manager requests the virtual execution environment manager (VEEM) to create, to resize, and allocate the VEEs, satisfying services' constraints.

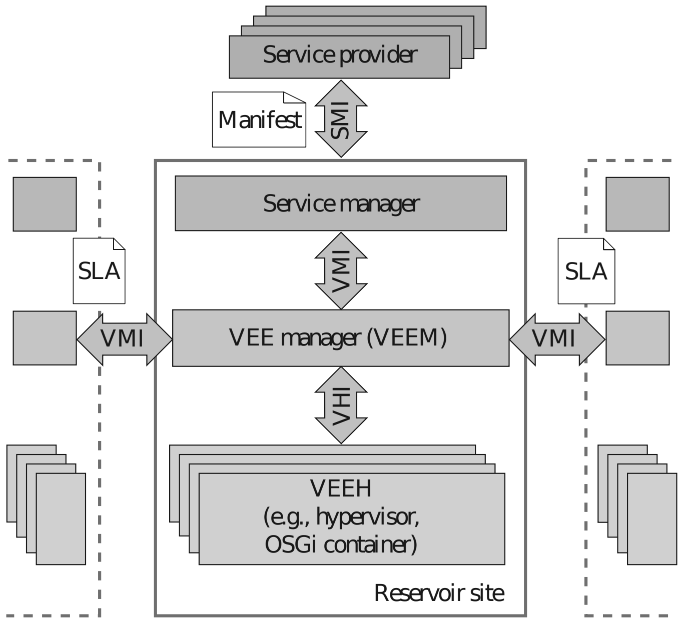


Figure 3.24: Reservoir architecture [296]

3.5.2.2 Open Cirrus

Open Cirrus [22] is a federated cloud testbed sponsored by different companies in collaboration with universities and institutions in the US, Germany, Russia, and Singapore. It is composed of ten data centers located in North America, Europe, and Asia to offer a cloud stack consisting of physical and virtual machines. The objective is to offer for researchers access to low-level hardware and software resources, which are usually not available and/or allowed in public clouds.

A Open Cirrus's data center consists of three service layers called foundation, utility, and primary domain services (Figure 3.25).

The foundation layer offers IaaS service capabilities; and it is based on Zoni. Zoni is a software responsible for managing physical resources and some cloud services such as node allocation, node isolation, software provisioning, network, and debugging of the allocated resources.

The utility layer comprises non-critical services such as monitoring, power management, networking file system, and accounting for resource utilization.

Finally, the primary domain layer offers PaaS-level services. Basically, this layer allows the users to execute Hadoop or MPI applications without needing to interact with the foundation services. It also provides services to manage virtual machine deployment across multiple data centers via an AWS-compatible API.

Moreover, Open Cirrus supports resource allocation across multiple sites through a set of global services. In this case, each service runs at one site to provide a common view of the infrastructure. These common services are single sign-on, global monitoring tools, and global storage.

According to the authors, the Open Cirrus testbed has been used to execute different applications.

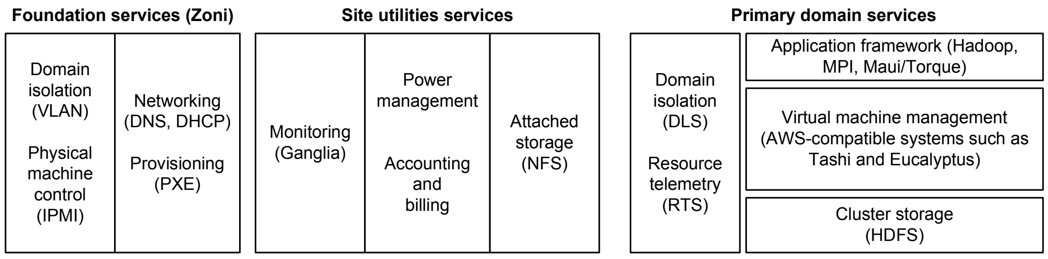
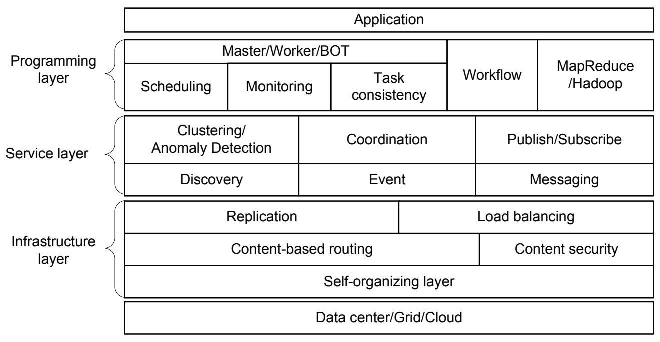


Figure 3.25: Open Cirrus architecture [22]

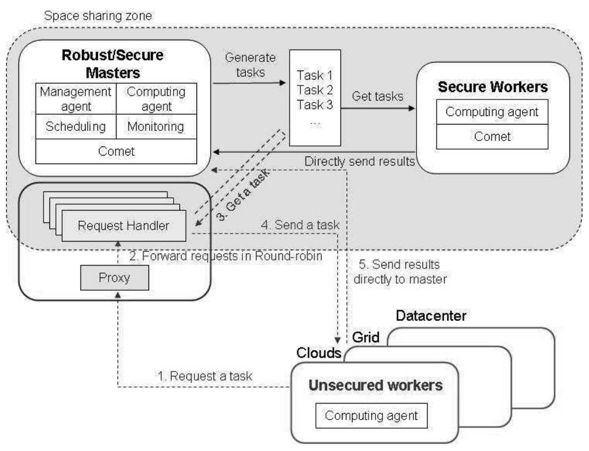
3.5.2.3 CometCloud

CometCloud [194] is a cloud architecture for cloud and grid environments. It comprises three layers namely infrastructure layer, service layer, and programming layer. The infrastructure layer uses a P2P overlay (i.e., Chord) to organize the resources and to implement data lookup. In this case, the nodes are organized according to their credentials and capabilities. In addition, each node propagates its state for its successor and predecessor. This allows the architecture to support node failure merging the state of the failed node with the one known by its predecessor and successor. The service layer implements coordination services based on a tuple space model. This coordination model creates a global space accessible for all nodes to implement membership authentication. This layer also implements publish/subscribe messaging and event services. Events describing the status of the system are clustered, and they are used to detect inconsistent states. Finally, the programming layer provides a framework for application development as well as for the execution of the applications.

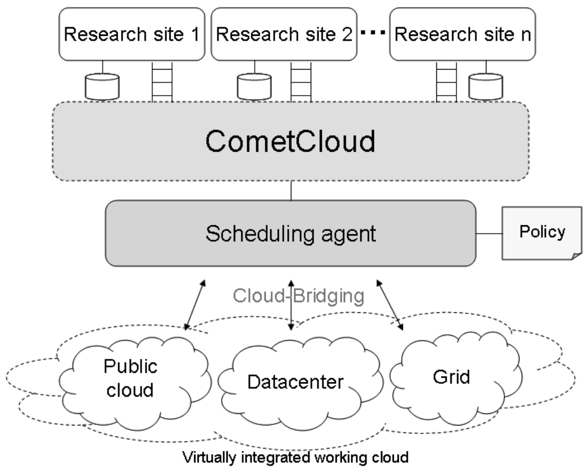
CometCloud employs a master/slave architecture to schedule the tasks across the clouds (Figure 3.26(c)). In this case, there is a scheduling agent responsible for distributing the tasks and for managing their execution in order to achieve the performance constraints. Moreover, the tasks are distributed according to a hierarchical security domain. The security domain comprises three levels. Trusted resources (i.e., dedicated resources) and management services (e.g., scheduling and resource coordination) are deployed in the first level (Figure 3.26(b)). The second level, on the other hand, comprises trusted workers that can pull tasks from the first level. Finally, the third level consists of untrusted workers (i.e., workers deployed in a public cloud) that require a proxy to access the resources in the first level. This hierarchical security model helps CometCloud on enforcing SLA guarantees, distributing the tasks first for trusted workers and for untrusted workers when the former is overloaded.



(a) Architecture layers



(b) Task distribution



(c) Cloud bridging

Figure 3.26: CometCloud architecture [194]

Experimental results showed a performance degradation due to the communication overhead between the workers and the master. The master run on a private cloud and the workers on Amazon EC2 to execute a Monte-Carlo application.

3.5.2.4 Contrail

Contrail [61] is a federated cloud architecture that aims to provide a homogeneous interface for the clouds. In such case, it employs two types of federation: horizontal and vertical. The vertical federation focuses on providing a platform to access the resources, whereas the horizontal focuses on abstracting the interaction model between the clouds.

The architecture of Contrail comprises three layers: interface, core, and adapters (Figure 3.27).

The interface layer gathers requests from users as well as from other Contrail components that rely on the federation. In this case, the services can be accessed through a command-line interface (CLI) or through a web interface. The core layer contains the modules responsible to support the federation requirements as identity management, application deployment, and SLA coordination.

The identity management implements single sign-on (SSO) services, which once authenticated, a user it not prompted to login at each cloud. Application deployment is implemented by the federation runtime manager (FRM). The FRM implements resource discovery considering different aspects such as cost and performance. It is also responsible for managing the application life-cycle. In this case, it gathers static and dynamic data of the federation through the provider watcher module. The provider watcher module monitors the applications and updates the state module.

The adapters layer contains external and internal adapters that enable the access to the services owned by both Contrail's clouds and external clouds. Internal adapters provider components for network, storage, and execution platform, while internal adapters supply provider-specific drivers for external clouds (i.e., non-Contrail clouds) by translating requests from the federation into requests that are understood by the providers.

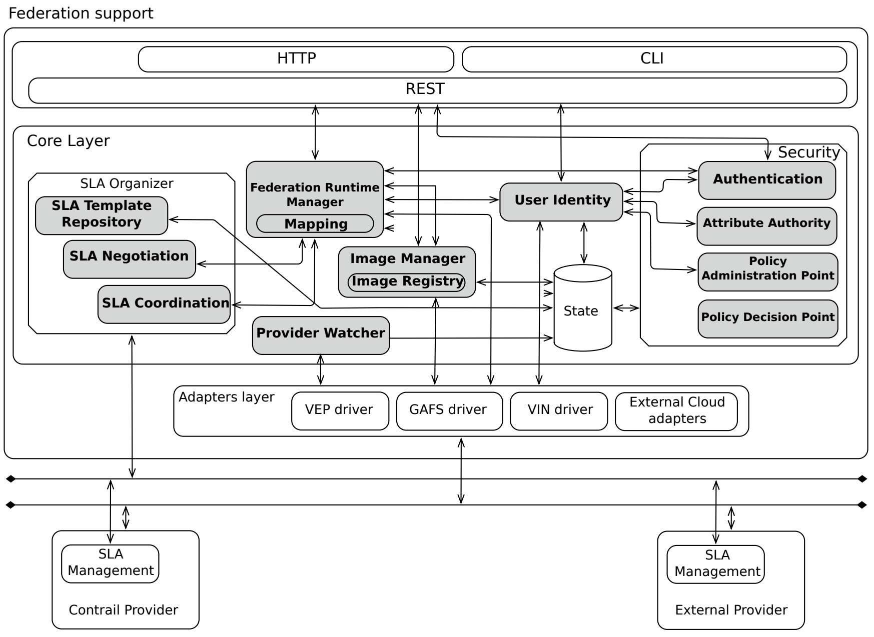


Figure 3.27: Contrail architecture [61]

Contrail has been used to deploy ConPaas [280], which is a platform-as-a-service for PHP applications.

3.5.2.5 OPTIMIS

OPTIMIS [112] is a multi-agent software architecture. It uses software agents to implement resource provisioning across multiple clouds. One of the OPTIMIS agents is the deployment engine (DE) (Figure 3.28) that is responsible to discover and to negotiate the resources required by a service. First, it identifies the providers that meet the services' requirements, then it creates a service manifest (i.e., SLA template), submits it to each cloud provider, and waits for the offers. When a request to host a service is received, a provider can reject it or make an deployment offer through the admission controller (AC). The deployment decision is based on the workload of the cloud, on the requested resources, and on the provider's objectives (i.e., profit).

Each deployment offer is evaluated by the DE that selects the best offer considering both qualitative (e.g., financial cost) and quantitative factors (e.g., energy consumption, trust). After selecting a deployment offer, the DE prepares a VMI with all the applications needed by the service and starts the deployment of the service within the selected cloud provider. With the services deployed, a service optimizer (SO) continuously monitors the service parameters to detect SLA violations. In OPTIMIS, the resources are allocated using a cloud optimizer (CO), which is responsible to optimize (e.g., consolidating the virtual machines) the cloud infrastructure.

Experimental results show that OPTIMIS was able to reduce the risk of over-provisioning up to 3.5% for unknown workloads in a private cloud.

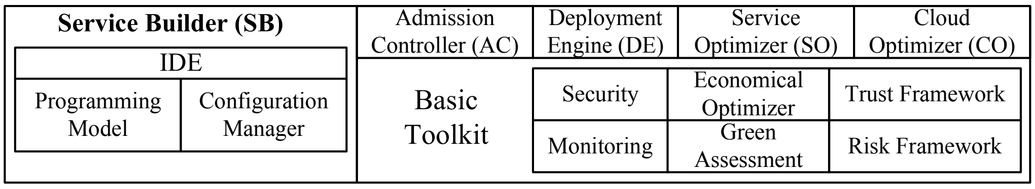


Figure 3.28: OPTIMIS architecture [112]

3.5.3 Comparative View

Table 3.3 summarizes the cloud architectures reviewed in sections 3.5.1 and 3.5.2. The first column presents the cloud computing system. The coordination model of the system is presented in the second column. The third column presents if the system can work in a multi-cloud environment. In this case, it can manage resources that belong to different clouds, and the clouds can be private, public, or hybrid. The cloud service model (Section 3.2.1) considered by the architectures is presented in the fourth column. Finally, the last column presents the type of application that is the focus of each system. In this case, *generic* means any type of applications; *cloud-aware* means application developed using one of the programming languages supported by the architecture such as Java, Python, Erlang, among others; whereas *Java* means that the system can only execute Java applications.

As can be seen, most of the systems rely on a centralized architecture. This is because these systems consider a static scenario, and they focus in helping the developers to create native cloud applications. In this case, there is still a demand for systems that can manage multiple clouds, taking into account different users' profiles and different kind of applications (i.e., native and cloud-unaware applications).

3.6 Summary

This chapter presented a detailed view of cloud computing, including the main technologies related to cloud computing systems, such as virtualization, service-level agreement (SLA), and MapReduce.

Table 3.3: Comparative view of some cloud architectures

System	Coordination model	Multi cloud	Cloud model	Application
Claudia [299]	Centralized	Yes	IaaS	Generic
SciCumulus [95]	Centralized	No	IaaS	Workflow (Simulation)
Cloud-TM [300]	Centralized	Yes	PaaS	Java
mOSAIC [278]	Centralized	Yes	PaaS	Cloud-aware
TClouds [355]	Centralized	Yes	PaaS	Hadoop
FraSCAti [317]	Centralized	Yes	PaaS	Java
STRATOS [274]	Centralized	Yes	PaaS	Java
COS [165]	Centralized	No	PaaS	Java
COSCA [180]	Centralized	No	PaaS	Java
Rafhyc [245]	Centralized	Yes	IaaS	Generic
JSTaaS [218]	Centralized	Yes	PaaS	Java
Reservoir [296]	Distributed	Yes	PaaS	Cloud-aware
Open Cirrus [22]	Distributed	Yes	IaaS	Generic
CometCloud [194]	Distributed	Yes	IaaS	Generic
Contrail [61]	Distributed	Yes	IaaS	Cloud-aware
OPTIMIS [112]	Distributed	Yes	PaaS	Cloud-aware

Virtualization techniques introduce many advantages for such systems, such as maintenance with negligible downtime, workload consolidation to maximize the usage of resources, and properly workload isolation.

MapReduce, on the other hand, provides a simple and high-level API that enables the developers to write distributed applications without needing to deal with failures, and that can automatically scale.

Nevertheless, the customers' services often rely on a single cloud infrastructure which may represent a drawback to the scalability and availability of the services. This issue can be mitigated by federating the resources that belong to different providers. This can help providers to handle the requests that exceed their capacity by delegating them to other cloud providers, avoiding over-provisioning solutions, and it can also help cloud users to increase their services availability.

Cloud federation is likely to represent the next step in the evolution of cloud computing, since it promotes the cooperation among different organizations and may contribute to push down the costs. Cloud federation enables a utility as a service model, where similar to the power generation, providers can delegate the exceed demand to their partners. Thus, cloud providers can elastically scale up/down their infrastructure by renting computational capability on-demand, and paying for them according to the business model (e.g., pay-per-usage) [70]. The challenge is how to deal effectively and automatically with failures, performance issues, and automatically deploy the services.

As in a cloud federation different platforms and technologies should coexist, this requires the development of middlewares to address the problems that may arise due to the federation characteristics, such as interoperability, heterogeneous resources, multiple allocation policies, coordination, and data location.

Finally, a federated cloud should be designed considering a dynamic scenario, where resources or even entire clouds could instantaneously appear and disappear. Considering this scenario, P2P techniques should be considered to implement discovery and services selection due to their characteristics (e.g., scalability and adaptability) in a highly transient environment.

Chapter 4

Autonomic Computing

Contents

In the beginning of the 2000s, it was clear that computing systems were extremely complex, running in heterogeneous and dynamic environments such as the Internet and interacting with other systems in complicated ways. In this scenario, even the most skilled administrators were having problems to configure, to tune, and to manage these systems properly [163, 191]. Some challenges that arise in this scenario are [191]: (i) computing systems must be scalable to provide a rich experience for users; (ii) computing systems usually run in heterogeneous and unpredictable environments under multiple administrative domains, where most of the interactions are known only at runtime; (iii) computing systems should be reliable even if they run in a unreliable environment; and finally (iv) computing systems must be aware of their execution environment.

Observing these difficulties, in March 2001, Paul Horn, from IBM, introduced the idea of autonomic computing at the National Academy of Engineering (NAE) by comparing computing systems to the human nervous system. He claimed that computing systems should be able to regulate themselves as the human body regulates itself [162]. He suggested that complex systems should independently take care of regular maintenance and optimization tasks, thus reducing the workload of the system administrators.

This chapter describes autonomic computing. First, it provides the definition of autonomic systems (Section 4.1) followed by the description of the autonomic properties (Section 4.2). Then, it presents concepts related to the architecture of autonomic systems (Section 4.3). Finally, some autonomic systems for large-scale distributed systems are presented and compared (Section 4.4), and section 4.5 summarizes this chapter.

4.1 Definition

Autonomic systems are defined as computing systems that manage themselves, adjusting to environmental changes in order to achieve the users' objectives [162]. Such systems must anticipate needs and allow users to concentrate on what they want to accomplish with the systems rather than spending their time overseeing the systems to get them there [162]. This idea comes from the humans autonomic nervous system, that can observe and adapt the human body to the environment, requiring little conscious actions [191, 236]. For instance, the nervous system autonomously controls the body's temperature, the heart rate, the blood sugar level, and other vital functions.

An autonomic computing system is essentially a self-management system. A self-management system can configure and reconfigure itself under varying and unpredictable scenarios without human intervention [27]. The main goal is to free system administrators from the details of system operation. To achieve that, self-management systems use autonomic managers (AMs) to monitor the environment and to take appropriate actions [164, 191, 236].

4.2 Properties

According to Horn [162], autonomic systems are self-management systems with eight characteristics or key elements: self-configuration, self-healing, self-optimization, self-protection, self-knowledge, context-awareness, openness, and self-adaptation. The first four elements are considered major characteristics, and the other elements are minor characteristics [162].

Self-configuration refers to the ability of an autonomic element (AE) to be seamlessly integrated into the system according to the system or users' objectives. In this context, an objective defines which states are desirable, without specifying how they should be accomplished. For example, registering a node in a Hadoop (Section 3.1.3) cluster requires changing on the Hadoop configuration in order to distribute tasks for the newest node. With self-configuration, a node can learn about the cluster and register itself, so that other components can either use it or modify their behavior to the new system's state.

Debugging and fixing problems in large-scale systems is usually a challenging task. They may require considerable amounts of time from programmers to diagnose problems and to fix them. Moreover, such systems are often composed of interdependent subsystems either developed or managed by different companies. *Self-healing* refers to the ability of a system to detect and to diagnose problems automatically. Initially, an autonomic system follows the rules defined by system administrators, but later, it begins to discover new rules on its own that help it to meet the users' goals. For instance, an autonomic system, after having detected a software bug, may decide to install software updates and to execute regression tests [191]. In this case, if the error continues, it can send a bug report to the system administrators with detailed information about the bug (i.e., context), or it can restore the system to a state that minimizes services' interruption.

Large-scale systems may have hundreds of tunable parameters that must be correctly set for the system to perform adequately. In addition, such systems are often integrated, where adjusting one subsystem may affect the entire system [191]. In the context of autonomic computing systems, *self-optimization* refers to the ability of a system to monitor itself constantly, to learn based on past experiences, and to adjust system parameters in order to satisfy the performance goals. For instance, adaptive algorithms running on each system can tune their own parameters and learn the best values that lead to the desirable performance level.

An autonomic system that pro-actively protects itself from malicious attacks and from the users who can make changes that affect the system's stability has the *self-protection* characteristic. In this context, autonomic systems continuously tune themselves to achieve desirable security levels and also to anticipate problems based on historical data.

Self-knowledge means the ability of a system to have detailed information about its environment even if it runs in an unpredictable scenario. In this case, the system should know its components as well as their status, their current capacity, and their connections with other systems to be able to manage itself properly. Moreover, it should know which resources belong to it, as well as the owners of other resources that it can borrow or lend, and finally which resources can be shared [162]. The self-knowledge property requires policies to govern the systems and to define their interactions.

Context-awareness refers to the ability of an autonomic system to know its environment and the context surrounding its activities, and to act accordingly. In this case, it can generate new rules to interact with its neighbors, and to describe itself and its available resources to other systems, as well as to automatically discover other resources in the environment. For example, in a cloud federation environment (Section 3.2.3), a context-aware autonomic system may negotiate the use of its underutilized resources with another clouds, which is geographically closed.

Besides these properties, autonomic systems must implement open standards to be correctly integrated in heterogeneous environments (*openess*). Moreover, they must anticipate their needs while keeping their complexity hidden from the users (*self-adaptation*). For instance, an application server can compare real-time access data with historical data to change the size of a cache service, anticipating performance degradation.

4.3 Architecture

Autonomic systems are often organized as a collection of distributed elements that interact to deliver services in accordance with some objectives in a control loop known as MAPE-K (Monitor, Analyze, Plan, Execute, and Knowledge) [164].

The monitoring function continuously collects, aggregates, and filters data obtained from a managed element. The analyzing function observes the environment and determines if some change needs to be made to achieve the desirable state. It takes into account the current system state, historical data, and policies information to correlate and to model complex events. The goal is to allow autonomic managers to learn about their environment and to help them to predict future states [164]. The planning function creates a plan with the actions needed to achieve the desired state. The execution function executes the plan considering the precedence order of the actions as well as state changes. The data used by these four functions are stored as shared knowledge to be used by autonomic managers. Autonomic managers obtain the knowledge retrieving it from policies, monitors, sensors or from an external knowledge source.

Figure 4.1 shows the structure of an autonomic element (AE). An autonomic element has one or more managed elements (MEs), composed of sensors and actuators, coupled with an autonomic manager (AM). The MAPE-K control loop is implemented by the AE and it is similar to a generic agent model [306], where an intelligent agent perceives its environment through sensors and uses such received data to act accordingly.

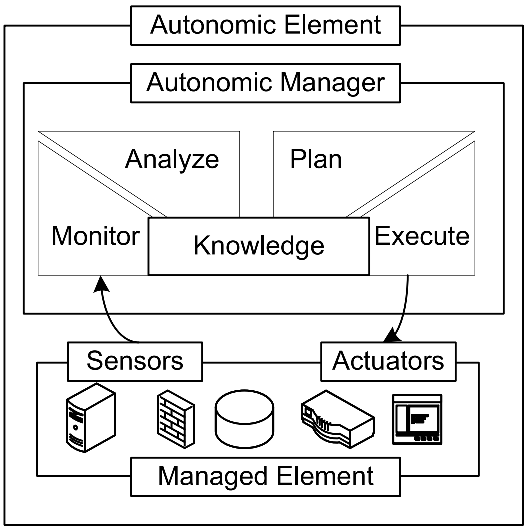


Figure 4.1: Architecture of an autonomic element [191]

The sensors are responsible for collecting data about the managed element (ME), which represent any resource (i.e., hardware or software) that has an autonomic behavior by coupling with an autonomic manager (AM) [191]. An AM is a software agent implemented with autonomic properties. It controls and represents the MEs, monitoring them through sensors and storing data into the knowledge database. Based on the system's knowledge, an AM decides the action to be taken, then it creates an execution plan, and sends it to the MEs using the actuators. After, the MEs change their state, executing the actions defined by the AM.

Each AE is responsible for managing its internal state and interactions with the environment. Its relationships are driven by goals either defined by the user or by other elements that have authority over it. The goals are expressed using event-condition-action (ECA) policies or through a utility function [163, 191]. For instance, a rule can be defined as follows: when the CPU load is less than 30%, then decrease the CPU frequency to the lowest frequency available. In addition, AEs may cooperate requesting either data or resources to achieve a common goal. In this case, each AE is responsible to request the necessary data and/or resources and to deal with failures of the required resource. Thus, AMs must be conscious of the state of their own managed elements and about the state of the environment, in special other autonomic elements in the network [163, 236].

4.4 Autonomic Computing Systems

Since the proposal of autonomic computing, many autonomic computing systems have been proposed in the literature. This section reviews some of these autonomic systems designed for large-scale distributed systems.

4.4.1 V-MAN

V-MAN [239] is a decentralized VM management system based on an unstructured P2P overlay. It implements self-optimization actions to reduce power consumption of the physical machines. The optimization actions are based on workload consolidation and on dynamic voltage and frequency scaling (DVFS) (Section 5.1). In V-MAN, each physical machine knows a subset of other physical machines included in its local view through a peer sampling service [174]. The control loop of V-MAN works as follows. Each node continually exchanges messages with its neighbors to: (i) share its local view; (ii) build a new local view, merging the old one with those received by the neighbors; and (iii) consolidate VMs in order to minimize the number of underutilized physical machines.

4.4.2 Sunflower

Sunflower [272] is an agent-based framework to coordinate workflow execution in an environment composed of grid and cloud resources. It implements self-optimization to meet performance requirements. In this case, the optimization process first tries to use the grid's resources. But, when the performance degrades and cannot meet the users' needs, it moves some tasks to the cloud.

Its architecture, depicted in figure 4.2, comprises three main layers: Sunflower console, cloud, and distributed information system. The Sunflower console provides a graphical user interface (GUI) for users to design their workflows. A workflow is submitted to the workflow manager that creates sub-workflows and submits them to different agents. The cloud layer manages computing resources. In such case, it uses Eucalyptus (open.eucalyptus.com) to allocate cloud's resources and a grid service to access grid's resources. The distributed information system organizes the nodes in groups using an ant clustering algorithm. These nodes cooperate to reach an orchestrated execution using a Petri Net policy associated with a BPEL application.

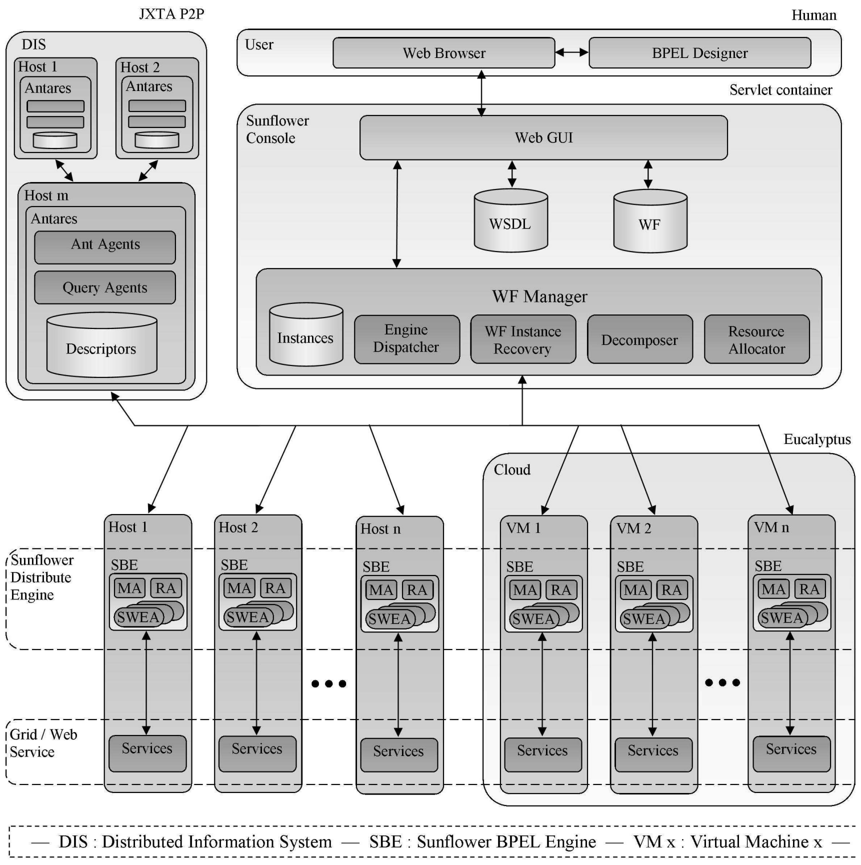


Figure 4.2: Sunflower architecture [272]

4.4.3 Market-based

In [86], the authors propose a market-based and self-optimization approach for resource allocation in a multi-application scenario. The optimization process aims to meet performance requirements, while trying to maximize the usage of the infrastructure. The control loop of this approach works as follows. First, based on a bid market, an application manager defines: (i) the minimal resources that should be allocated to meet the application's requirements; (ii) the spending rate; and (iii) the bids expressing their willingness to pay for the resources. Then, it submits these data to a resource manager. Next, the resource manager evaluates the requests and allocates the resources according to the applications' bids, grouping the highest bids in nodes with more free resources. In this case, the resource manager maximizes resource usage and minimizes allocation failures, whereas the application managers meet QoS requirements by adjusting their bids or the spending rate.

4.4.4 Component-Management Approach

In [96], Oliveira and colleagues present a self-optimization cloud architecture to reduce power consumption. In this architecture (Figure 4.3) there are two autonomic managers — Application Manager (AM) and Infrastructure Manager (IM) — responsible for managing the applications and the infrastructure. Each AM tests different configurations such as disabling logging events or consolidating virtual machines in order to accomplish its objectives (e.g., to reduce power consumption). An application is considered as a set of independent components that can be deployed in different virtual machines and integrated by AMs. In other words, AMs are responsible for synchronizing the states between the components and their hosting virtual machines to minimize power consumption.

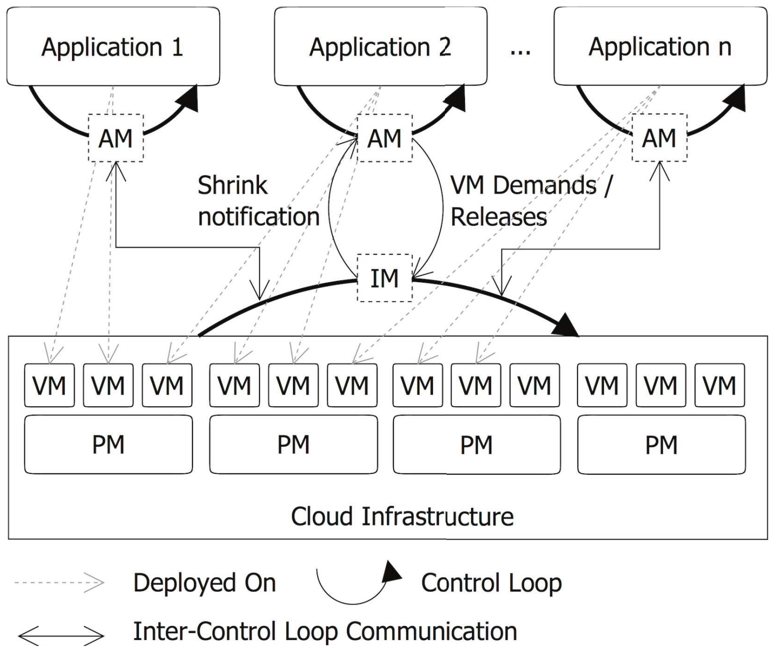


Figure 4.3: Control loop of the self-management cloud architecture [96]

4.4.5 Snooze

Snooze [111] is a hierarchical autonomic framework for VM placement. Snooze implements self-optimization and self-healing by distinguishing three types of agents (Figure 4.4): local controller, group leader, and group manager. These agents are responsible for allocating and for monitoring the virtual machines. Implementing self-optimization actions, Snooze tries to free the users (e.g., system administrators) from some system details such as where and how to deploy a VM; and to minimize the number of overloaded or underutilized physical machines. Self-healing, on the other hand, implements fault-tolerance employing heartbeat multi-cast and leader election to discover and to connect group managers and group leaders.

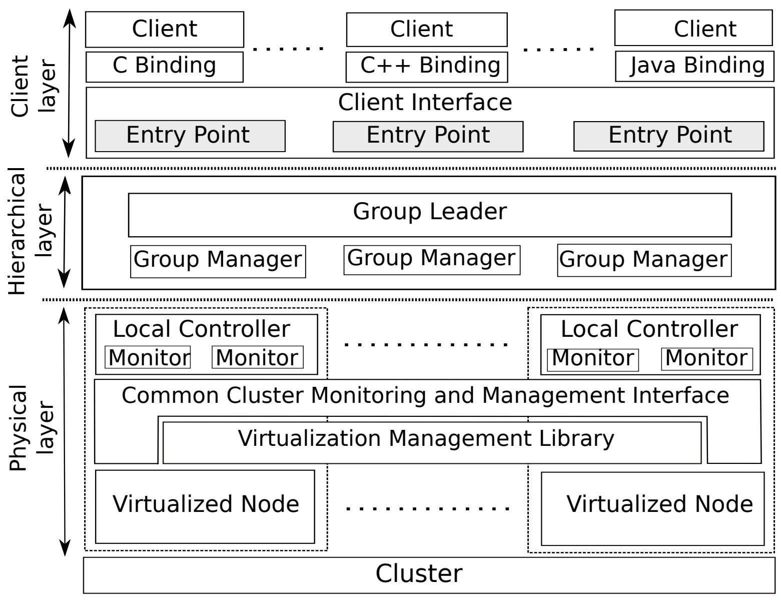


Figure 4.4: Snooze architecture [111]

4.4.6 Cloudlet

Cloudlet [321] is a multi-agent cloud computing architecture to discover and to negotiate cloud services. It implements self-optimization to free the users from the details of selecting and negotiating with different cloud providers functional and financial requirements. This architecture implements a control loop that work as follows (Figure 4.5). First, the cloud resources are registered in a database either by the cloud providers or by a crawler. Second, the users submit a query with their functional and budget requirements to the service discovery agent (SDA). Finally, the SDA negotiates with the cloud providers, and selects the services that match the user's requirements. As each cloud provider often uses different terminologies to describe its resources, the SDA uses an ontology to reason about the similarity between the users' specification and the resources' descriptions.

4.4.7 Distributed VM Scheduler

The distributed VM scheduler (DVMS) [286] is an agent-based VM scheduler that aims to optimize VM placement based on a structured P2P overlay (Figure 4.6). The P2P overlay is the Chord (Section 2.4.3). It implements self-optimization actions to select a node to host a virtual machine according to some objective. In DVMS, each physical machine has an agent responsible to monitor its resources, to communicate with its neighbors, and to consolidate the virtual machines. In this case, the control loop of DVMS works as follows. First, when a node detects a problem (e.g., overloaded or underutilized state), it sends a message to its neighbor. Then, the neighbor can join the search, if it is not already involved in another one, or it can forward the message to its neighbor. The first node that accepts the message becomes the leader and the responsible for resolving the problem (i.e., to allocate the VM). As this process may lead to deadlock due to node partitions, duplicate searches are finished by the node with the lowest identifier.

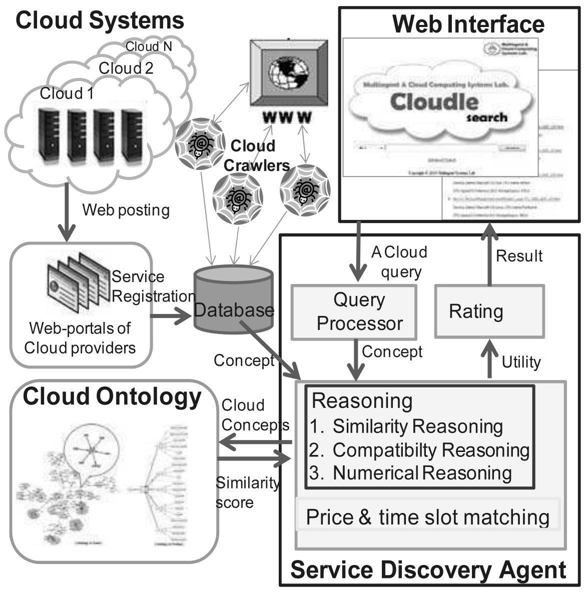


Figure 4.5: Cloudlet architecture [321]

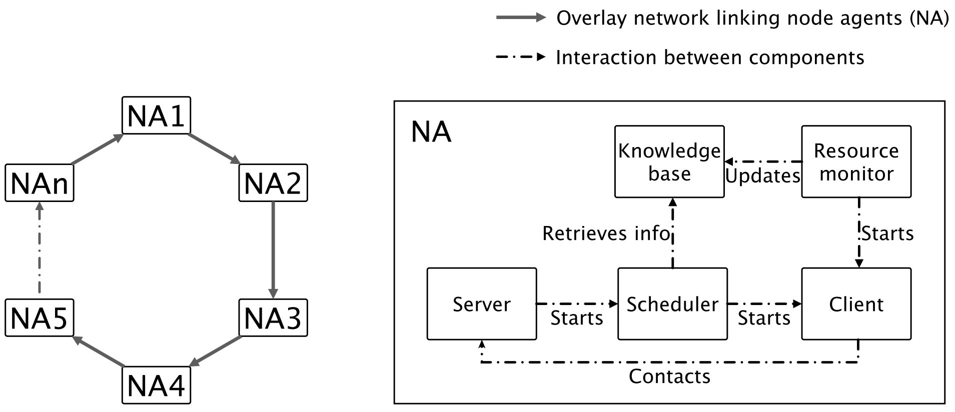


Figure 4.6: DVMS control loop [286]

4.4.8 Thermal Management Framework

In [298], Rodero and colleagues present an autonomic thermal management framework. This framework implements self-optimization actions to reduce energy consumption through workload consolidation (Section 3.1.1.4), taking into account performance constraints. Its architecture (Figure 4.7) has some actuators and sensors that are responsible for implementing self-optimization actions, using CPU pinning to handle thermal hotspots and DVFS (Section 5.1) to reduce energy consumption. In this case, when the temperature of a CPU core reaches a threshold, the system assigns its VMs to another CPU core, and decreases both the frequency and the voltage of the freed CPU. Although this may reduce the energy consumption, it may increase performance penalties due to high resource sharing rates. Indeed, the system tries to partition the workloads according to their characteristics (e.g., CPU-, memory-, and/or IO-bound) and to allocate each partition to a physical server based on an off-line profile in order to minimize both power consumption and performance losses. In other words, workloads with different characteristics are combined and assigned to the same physical server.

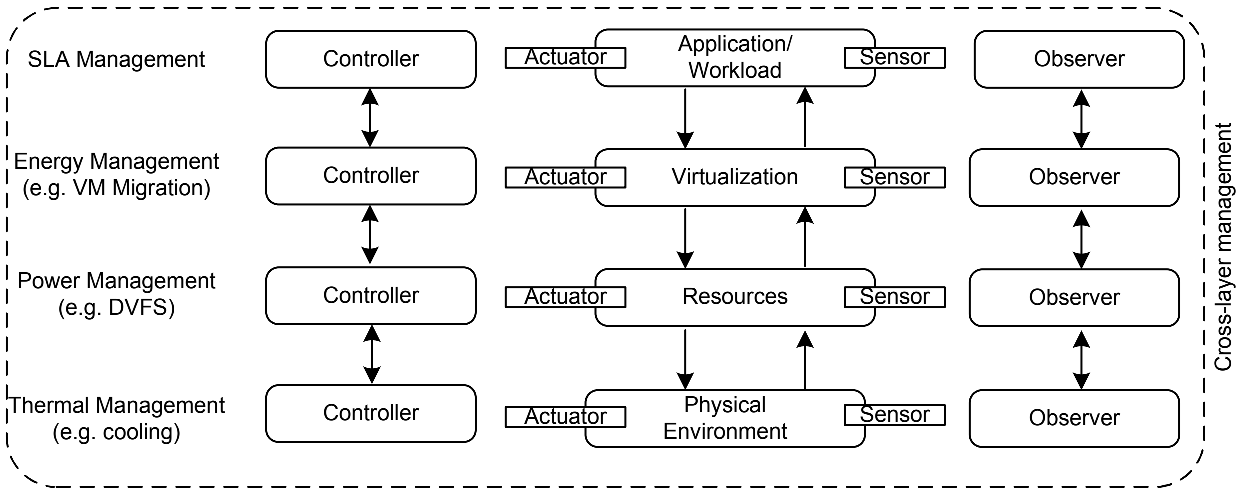


Figure 4.7: Thermal-aware autonomic management architecture [298]

4.4.9 SmartScale

SmartScale [105] is an auto-scaling cloud architecture. It scales the applications based on performance requirements and on the cost to reconfigure the resources. SmartScale implements self-optimization properties to decide when the virtual machines should be reconfigured and how this must be done. In this context, it tries to find an equilibrium between the number of required resources, the performance goals, and the cost to reconfigure the infrastructure.

The architecture of SmartScale, depicted in figure 4.8, comprises the following key components: data warehouse, predictor, workload classifier, smartscale, and virtualization manager. Its control loop works as follows. First, the data about the applications are stored in a repository (i.e., data warehouse). These data include the number of requests of each type for an application, the SLAs, and the CPU and memory usage of each virtual machine. Then, the predictor reads the repository, and uses time-series analysis to predict the number of requests of each type for the next configuration time. Next, a workload classifier groups the workloads. After, the smartscale component selects the appropriate allocating model for the virtual machines. Finally, the virtualization manager applies the new configuration.

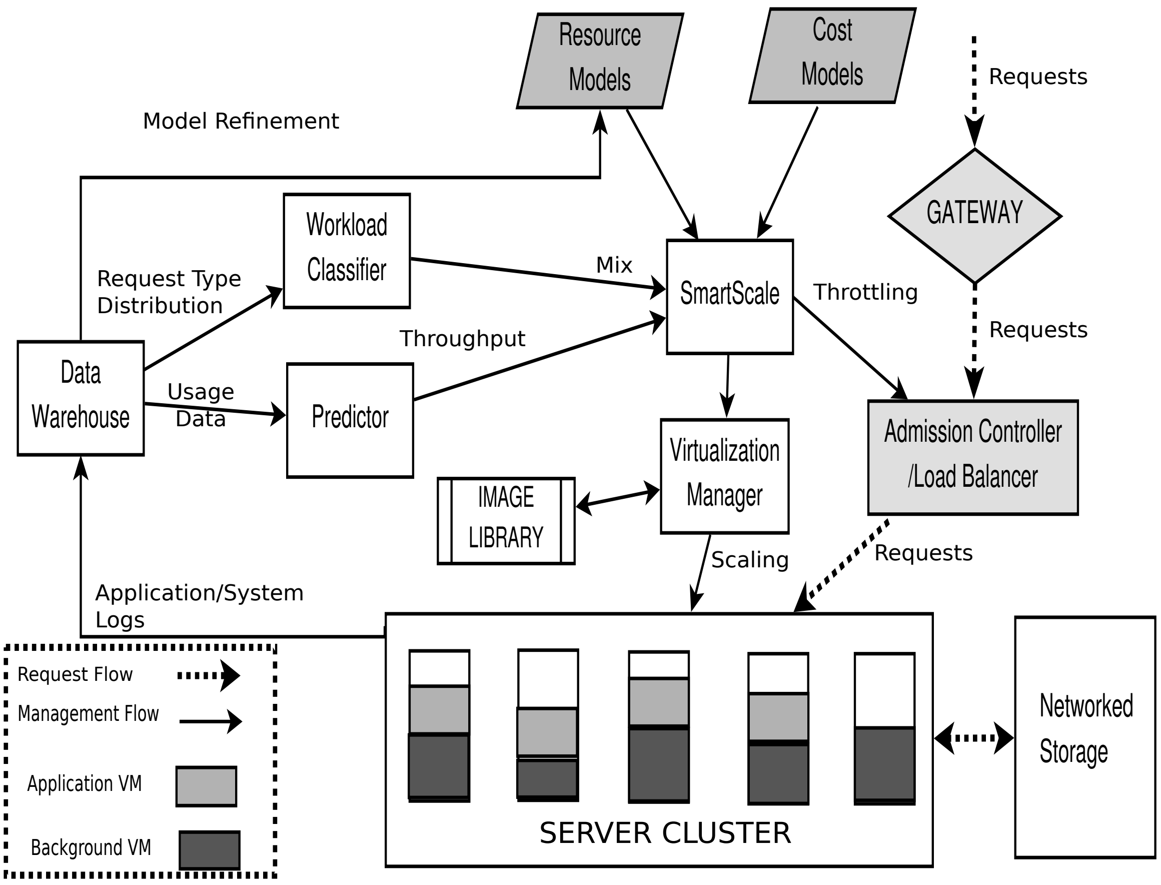


Figure 4.8: SmartScale architecture [105]

4.4.10 SLA Management

In [19], Ardagna and colleagues present a resource allocation framework that aims to minimize both power consumption and SLA violations. The framework implements a control loop to monitor the resources; to decide if it accepts a request to host a VM; and to select a physical machine for hosting the VM. A request is accepted based on a utility function that computes the expected revenue of a request or possible SLAs' penalties due to an increase in the response time of the applications. In this case, using the control loop, the system tries to maximize the data center revenue, consolidating the workload in a way that minimize both power consumption and SLA violation.

4.4.11 Comparative View

Table 4.1 summarizes the autonomic systems reviewed in this section. The first column presents the system's name or the paper. The second column presents their architecture characteristics. In this case, a centralized solution means that the system depends on centralized information to work, even though there are some distributed components. The autonomic properties implemented by each system are presented in the third column. The fourth column presents the environment for which the autonomic system was proposed. Finally, the fifth column presents the objectives of the reviewed systems.

As can be seen in table 4.1, half of the systems rely on centralized information to take decisions. This may limit its usage in large-scale systems. Furthermore, as these systems focus on self-optimization, this may represent a challenge for their availability. Moreover, depicted the vision of autonomic computing, some of its properties such as self-configuration, self-healing, and context-awareness have not been completely addressed by today's autonomic systems, while self-optimization has started to be implemented by such systems. Only Snooze [111] implements self-healing and self-optimization using a hierarchical architecture to allocate virtual machines in a private cloud.

4.5 Summary

This chapter presented the definition, properties, and important concepts related to the architecture of autonomic systems. Moreover, it described some autonomic systems designed for large-scale systems.

The self-management property of autonomic computing systems may be used to reduce considerably the burden on system managers, autonomously optimizing themselves according to the environment changes. However, to achieve this, there are still many work to do, since the majority of the autonomic systems available in the literature focuses on one autonomic property, such as self-configuration, employing a centralized control management.

Table 4.1: Autonomic computing systems

System	Architecture	Autonomic[⋆] characteristics	Environment	Objectives
V-MAN [239]	Unstructured P2P	SO	Cloud	Minimize power consumption
Sunflower [272]	Centralized	SO	Grid & cloud	Minimize SLA violation
Market-based [86]	Centralized	SO	Cloud	Minimize SLA violation
Component-management [96]	Centralized	SO	Cloud	Minimize power consumption
Snooze [111]	Hierarchical	SO & SH	Cloud	Minimize power consumption
Cloudlet [321]	Centralized	SO	Cloud	Minimize the cost of using the cloud
DVMS [286]	Structured P2P	SO	Cluster	Minimize the number of physical machines
Thermal [298]	Centralized	SO	Cloud	Minimize energy consumption
SmartScale [105]	Centralized	SO	Cloud	Minimize SLA violation
SLA management [19]	Centralized	SO	Cloud	Minimize power consumption and SLA violation

[⋆] self-healing (SH), self-optimization (SO)

Chapter 5

Green Computing

Contents

The advances in large-scale distributed systems have historically been related to improving in performance, scalability, and quality of service for the users. However, as the infrastructures increased in size and computational capacity, other issues became critical. Power management is one of these issues, because of the constantly increasing power consumption of computing infrastructures. Also, systems with high peak power demand complex and expensive cooling infrastructures [44, 183]. Depending on the efficiency of a computing infrastructure, the total number of watts required for cooling can be from three to thirty times higher than the number of watts needed for the computation [330].

According to Kumar [207], the actual power and cooling costs of servers represent almost 60% of their initial acquisition cost. A study on power consumption by servers and data centers in the US shows that the electricity used by the large-scale infrastructures in 2006 cost about USD 4.5 billion [201], and data centers are responsible for the emission of tens millions of megatons of carbon dioxide annually. This study also indicates that energy consumption has doubled, if compared to the consumption in 2000.

One of the major causes of inefficiency in data centers is the idle power wasted when servers run at low utilization. This occurs because the power consumption of CPUs even at low load utilization rate, is over 60% of the peak load [30, 50, 110, 213]. Therefore, reducing the energy consumption without sacrificing quality of service commitments is an important issue, both for economical reasons and also for making the computing infrastructure sustainable [38].

Currently, with the energy costs increasing, the focus shifts from optimizing large-scale resource management for pure performance to optimize it for energy efficiency while maintaining the level of the services [56].

Another important issue in large-scale distributed systems is the level of carbon emissions. The high amount of carbon emissions in data centers is associated with the amount of energy consumption. A recent study on cloud computing and climate change shows that the total electricity consumed by data centers in 2020 will be 1,963 billion kWh and the carbon emissions associated would reach 1,024 megatons, overtaking the airline industry in terms of carbon emissions [142]. Moreover, most of the electricity power comes from non-renewable and high-carbon fuels [29]. In this scenario, strategies that are aware of energy are gaining attention in academy and industry since they can fulfill the promise of executing applications that can be tuned to consume less energy [244].

However, this represents a challenging problem, since there are many variables that contribute to the power consumption of a resource. First, large-scale infrastructures comprise different layers, and characterizing the power consumed by each of them is usually a difficult task [265]. For instance, the power consumed by a resource may change according to its position in the data center, as well as the data center's temperature [60, 264]. Second, it is often difficult to determine the locations where a workload should be distributed while considering performance requirements. Third, in large scale, data centers' carbon footprint can vary significantly according to their load [129]. Finally, the environment impact of an application may change depending on the users' location, as electricity cost and carbon footprint are location specific [129, 230]. Moreover, it may be difficult to determine the overall power footprint of a workload since some companies still do not publish any information about their power source [85].

Green computing involves a set of methodologies, mechanisms and techniques that helps computing systems (hardware and/or software) to reduce power consumption or carbon footprint. This chapter presents an introduction to green computing, including the concepts of energy-aware computing (Section 5.1), green data centers (Section 5.2) and green data centers benchmarks (Section 5.2.1). Finally, some green data performance indicators (GPI) related to energy consumption are presented (Section 5.3).

5.1 Energy-Aware Computing

The power consumption of a system comprises two parts: a static (or leakage power) and a dynamic part. The leakage power depends on the system size and on the type of transistor. Static power consumption is related to the leakage currents that are present in any powered system and it is independent of clock rates and usage scenarios. The dynamic part, on the other hand, depends on the activity of a circuit, the usage scenario, and the clock rates. Dynamic power consumption is mainly composed of short-circuit current and switched capacitance [36]. Hence, it can be computed as showing in equation (5.1):

$$P = aCV^2f \tag{5.1}$$

where a is the switching activity, C is the capacitance, V is the supply voltage, and f is the frequency.

Energy consumption, on the other hand, is defined as the average power consumption over a period of time. Clearly, energy consumption and power consumption are closely related but it is easy to see that a reduction in power consumption does not necessarily imply a reduction in energy consumption.

Many efforts have been made to reduce both static and dynamic power consumption. As a result of these efforts, the hardware energy efficiency has significantly improved. However, whereas hardware is physically responsible for most of the power consumption, hardware operations are guided by software, which is indirectly responsible for the energy consumption [6].

Some of the software-level studies tried to minimize the power consumption by considering that the main reason for energy inefficiency is resource underutilized. One of the first approaches to try to solve this problem consists of shutting down idle nodes [193, 251] and waking up them when the workload increases or the average QoS violation ratio exceeds a threshold.

At the hardware level, improvements are made turning off components, putting them to sleep or changing their frequency using dynamic voltage and frequency scaling (DVFS) techniques [135, 149, 337]. DVFS techniques assume that applications dominated by memory accesses or involving heavy I/O activities can be executed at lower CPU frequency with only a marginal impact on their execution time. In that case, the goal of a DVFS scheduler is to identify each execution phase of an application, quantify its workload characteristics, and then switch the CPU frequency to the most appropriate power/performance mode. In other words, DVFS is a dynamic power management (DPM) technique (Figure 5.1) that uses real-time data about a system to optimize its energy consumption, without decreasing the peak power consumption [36].

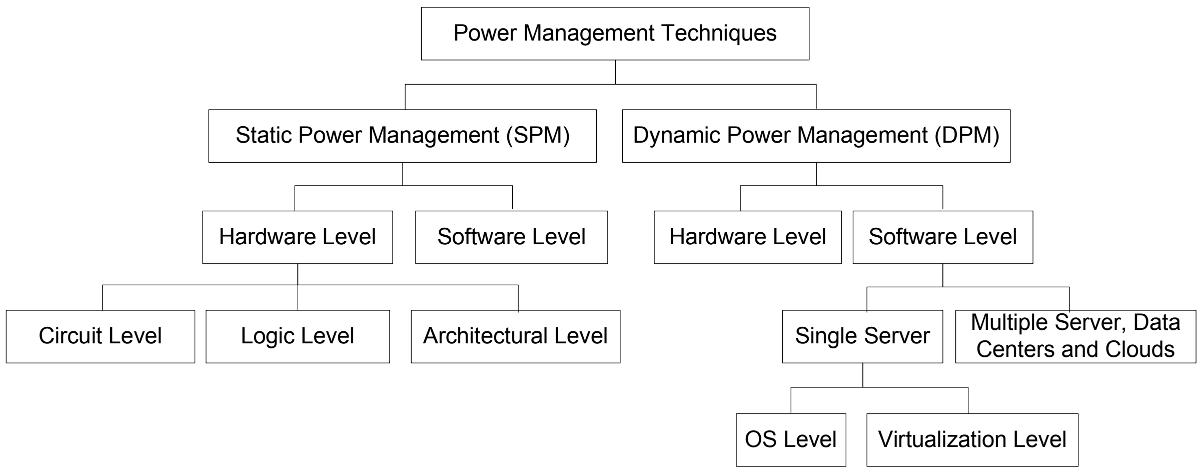


Figure 5.1: Taxonomy of power and energy management techniques [36]

The problem of energy-efficient resource management has also been investigated in the context of operating system, followed by homogeneous and heterogeneous architectures such as clusters [204, 281], P2P [343], and cloud [133, 381]. Moreover, a number of studies have explored software optimization at compiling level [53, 109, 322] in order to improve both performance and energy-efficiency.

Besides reducing power consumption, data center providers are required to reduce their carbon footprint from different organizations around the world. In this scenario, some companies such as Apple, Google, and Microsoft have already taken the initiative to achieve carbon neutrality [85, 369]. For instance, Apple (apple.com/environment) aims to run its data centers using only renewable energy [85]. Although renewable energy may have a lower carbon emission, in practice, it may be unreliable since green power supplies highly depend on the weather conditions, which are usually time-varying. Thus, this limits the broad adoption of green energy by large-scale infrastructures as they are mostly designed for high availability [76, 324].

5.2 Green Data Centers

Cloud computing systems are often built on large data centers. These data centers usually have a considerable number of servers and a sophisticated support and cooling infrastructure that consume a huge amount of power and produce a lot of heat. Energy efficiency is therefore a major concern in the design and operation of data centers.

According to Wang and Khan [366], there are two main methods to build a green data center: (a) design and build the data center with green elements; or (b) operate a conventionally built data center in a green way. These two methods are non-exclusive in such a way that data centers designed as green can be also operated in a green way.

Green data center designs normally take into consideration the cooling systems inside the chip. A sophisticated cooling solution with, for instance, localized cooling paths with runtime thermal management, will generally favor designs that include a small number of complex cores with power-hungry needs. On the other hand, a great number of simpler cores can also be used, replacing the complex cores. A simpler cooling solution will often be applied to simpler cores. In theory, both alternatives are able to deliver the same watt per second rate, at a much higher financial cost for the first alternative.

In order to operate a data center in a green way, several techniques can be used, such as: (a) reducing the data center temperature; (b) increasing the server utilization and consequently turning off idle servers and/or (c) decreasing on-the-fly the power consumption of the computing resources.

5.2.1 Green Data Center Benchmarks

Once a green data center is put into operation, there should be one or more metrics to measure its *"greeness"*. Measuring how green a data center is is important mainly for two reasons. First, it is an indicator of the effectiveness of the techniques employed. Second, it provides a way to compare several data centers and rank them according to the success of their green design.

In this scenario, there are challenges in identifying which metrics to be used, how to use them and which components to measure. Initially, power and cooling components were considered because of their size and scale of energy consumption. Nowadays, it is becoming clear that metrics should also be applied to the computing components as well. Moreover, when systems are interconnected, which is the case in a data center, defining energy efficiency is not straightforward. Saving energy in one computer can possibly cause another one to consume more energy, increasing the overall energy consumption [232].

Metrics provide insight about how resources are being used and the efficiency of that usage. They are often related to performance, availability, capacity and energy consumption for servers, I/O resources and others, to meet a given level of service and cost objectives [314].

Several metrics for assessing performance have been proposed over the past decades, which have been successfully used in benchmarks. However, most of the benchmarks available do not measure energy efficiency. Only in the past years, the need for establishing good benchmarks to measure green solutions really appeared. Efforts in this direction include the Green 500 initiative (green500.org), The Green Index (TGI) — (thegreenindex.com), the SPEC Power Initiative [210] and JouleSort [295], which will be discussed in the next subsections.

5.2.1.1 The Green500 Initiative

The Green500 list aims at increasing awareness about environmental impact and long-term sustainability of high-end supercomputers by providing a list that ranks the most energy-efficient supercomputers in the world. This ranking is based on the amount of power needed to complete a fixed amount of work [79].

The Green500 effort treats both performance and power consumption as important design constraints for high performance computing environments. Its main idea is to use the flops (Floating Point Operations per Second) metric of the LINPACK benchmark [227], which is widely used by the supercomputing community, and to combine it with a simple metric that measures the power consumed, creating the flops per watt metric. In this metric, the watts measured are the average system watts consumed during the execution of the LINPACK benchmark with a specific problem size.

Even though the flops per watt metric is having good acceptance, some authors discuss some of its potential drawbacks. For example, in [79], Feang and colleagues argue that this metric might be biased toward smaller supercomputing systems since the wattage scales linearly with the number of compute nodes whereas the flops performance scales sub-linearly for non-embarrassingly parallel problems, which is the case for LINPACK. This implies that smaller systems would have better ratings in such a metric. For this reason, only the Top500 supercomputers (top500.org) are considered in the Green500 List.

5.2.1.2 The Green Index

The green index (TGI) metric [333] aims to capture the energy efficiency of a high performance computing (HPC) system. It focuses on computing power, memory operations and I/O activities. The main idea of The green index (TGI) is to combine the outputs of several benchmarks into a single metric, which is obtained in 4 steps. In the first step, different benchmarks are executed and the performance to watt ratio is calculated. The ratios obtained in step 1 are compared in step 2 to a reference system (as in the SPEC benchmark suites), generating TGI components. In step 3, a weight is assigned to each TGI component obtained in step 2. Finally, all the TGI components are multiplied by their respective weights and the addition of all these values is The green index.

In order to evaluate TGI, the authors chose a 128-node cluster with 1024 cores as the reference system. The cluster evaluated was an 8-node cluster with 128 cores and HPCC benchmarks [233] that stress CPU, memory and I/O were used. Even though good results were obtained, these results were clearly dependent on a good choice of the weights and on a good reference system.

5.2.1.3 SPECpower

The Standard Performance Evaluation Corporation (SPEC) (spec.org) is a non-profit corporation formed to establish, maintain and endorse a set of metrics and benchmarks to be applied to a wide range of computing platforms [210].

In 2006, the SPEC community started the development of the SPECpower_ssj2008 benchmark, which was one of the first efforts to establish an industry-standard benchmark that measures power and performance characteristics of several types of servers.

The SPECpower_ssj2008 is a Java application that executes several transactions spread over 11 target loads, stressing the CPU and the memory subsystem [309]. The throughput of each target load is measured in ssj_ops, i.e., the number of transactions completed per second over a fixed period of time. The SPECpower_ssj2008 metric is then calculated by dividing the sum of the ssj_ops in each load by the sum of the average power consumed in each load.

As noted by some authors [309], the main limitation of SPECpower is that it does not consider the disk activity and this activity is an important factor in measuring the energy efficiency of a computing system.

5.2.1.4 JouleSort

JouleSort [294] is a holistic benchmark to measure the energy-efficiency of computer systems, stressing CPU, memory and the I/O subsystem. The sort application was chosen because of its simplicity, portability and capacity to stress multiple components of a given machine.

The JouleSort benchmark sorts a file with 100-byte records and 10-byte keys. It is required that the input and output files must be in non-volatile storage. The number of records to be sorted is fixed in three scales: 10^8, 10^9 and 10^{10} records. The metric used is SortedRecs / Joule. In this metric, energy is measured for power supplies of all hardware components, including the idle machines and the cooling system. The energy (Joule) is the total energy, which is the product of the average of the power consumed by the wallclock time.

5.2.1.5 Comparative View

Table 5.1 summarizes the benchmarks discussed in section 5.2. In column 1 the green benchmark is provided. The metrics are provided in column 2. Column 3 provides the workload category. Finally, the domain (column 4) of the metrics can be data center, which include all data center resources; enterprise, which consider QoS and organization process parameters; and mobile and desktop resources.

Table 5.1: Energy efficiency benchmarks and metrics

Benchmark	Metric	Category	Domain
Green 500	MFLOPs/Watt	HPC	Data center
TGI	Performance/Watt	HPC	Data center
SPECpower	Operations/Watt (T/W)	Web	Enterprise
JouleSort	Records sorted / Joule	IT Resources	Mobile, desktop, enterprise

5.3 Green Performance Indicators

Green performance indicators (GPIs) are defined as the driving policies for the data collection and analysis related to energy consumption. The idea of GPI is interesting because it can be adapted as criteria to define SLAs (Section 3.1.2), where requirements about energy efficiency of services versus the expected quality of services are specified and need to be satisfied.

In sections 5.3.1 and 5.3.2 we discuss two approaches used to categorize GPIs.

5.3.1 The Approach of Stanley, Brill, and Koomey

5.3.1.1 Overview

Stanley and colleagues [331], categorize the GPI metrics into four groups: IT strategy, IT hardware utilization, IT energy efficient hardware deployment, site infrastructure. Each of these groups consider different issues in a green data center deployment and involve different metrics and techniques.

The IT strategy category contains the strategic choices made by the enterprise/organization to achieve its goals using less energy. These choices can be: data center Tier functionality levels, centralized *vs* decentralized processing, and disaster recovery mechanisms, among others. The IT hardware utilization category involves techniques such as turning off servers and storage systems and using consolidation techniques, among others. In the IT energy efficient deployment category, power-efficient techniques used to build power supplies and more efficient chips are considered. Finally, the site infrastructure category deals with techniques used to reduce the power consumed by the whole data center. Figure 5.2 illustrates this categorization.

In the following paragraphs, the metrics DH-UR, DH-UE, and H-POM are presented.

5.3.1.2 Metrics

deployed hardware utilization ratio (DH-UR): this metric measures how many servers are running applications in relation to the total number of servers in the data

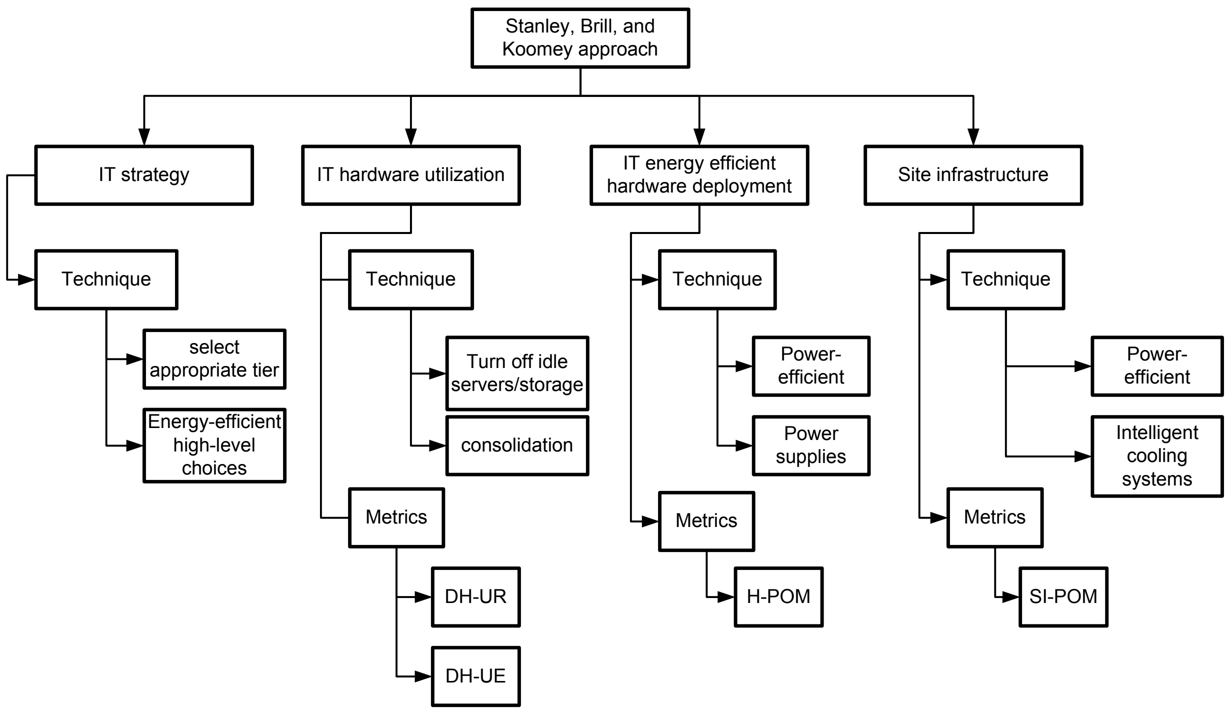


Figure 5.2: The metrics categorization of Stanley, Brill, and Koomey [331]

center. For servers, it is computed as shown in equation (5.2) and for storage it is computed as shown in equation (5.3).

$$\text{DH-UR (Servers)} = \frac{\text{Number of servers running applications}}{\text{Number of servers}} \tag{5.2}$$

$$\text{DH-UR (Storage)} = \frac{\text{Number of terabytes of storage holding frequently accessed data}}{\text{Total terabytes of storage}} \tag{5.3}$$

deployed hardware utilization efficiency (DH-UE): this metric measures how efficiently the servers are being used. It is computed as shown in equation (5.4).

$$\text{DH-UE} = \frac{\text{Minimum number of servers necessary to handle peak compute load}}{\text{Total number of servers}} \tag{5.4}$$

The minimum number of servers necessary to handle peak load is defined as: the highest percentage of compute load of each server plus any overhead incurred in virtualization, expressed as a percentage of the maximum load of a single server.

IT hardware power overhead multiplier (H-POM): the H-POM metric measures how much of the power input to a piece of hardware is wasted in power supply conversion losses or diverted to internal fans, rather than making it useful to computing components. It is computed as shown in equation (5.5).

$$\text{H-POM} = \frac{\text{AC hardware load at the plug}}{\text{DC hardware compute load}} \tag{5.5}$$

The H-POM metric measures on where the data center equipment runs on its operating curve. For instance, a server may have a power supply that is very efficient at peak load (100 % of load) but inefficient at low loads. When the server runs only a single application that requires little electric power to the processor, the power source could run very inefficiently, resulting in a high H-POM.

5.3.1.3 Final Remarks

The metrics proposed by [331] focus in the efficiency of the data center equipments without considering the characteristics of the applications deployed in Data center's servers. One possible consequence of it is that the data center can be energy-efficient, but with a high impact to applications and in SLA indicators. It is also important to consider the greenness factor of applications running in the data center because their design can incur in an inefficient use of resources.

5.3.2 The Green Grid Approach

5.3.2.1 Overview

The Green Grid (thegreengrid.org) provides metrics that can helps to improve the overall data center energy efficiency, considering resources usage efficiency (IT and non IT resources) and their power consumption, tactical and operational data center operation efficiency, including QoS parameters and the technologies used as well. Figure 5.3 presents the metrics proposed by Green Grid.

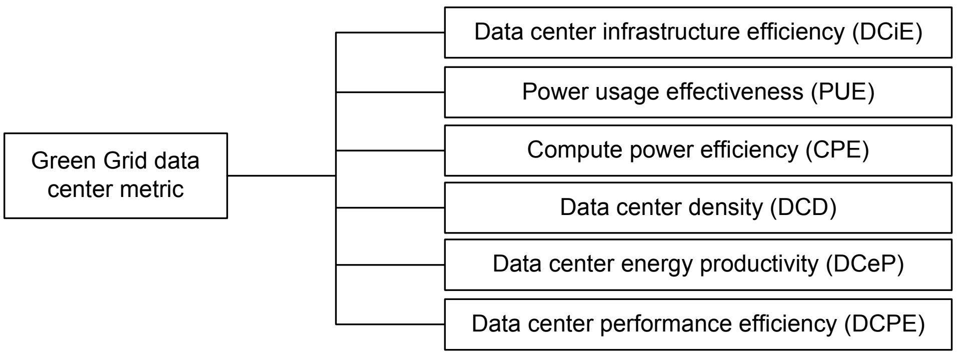

Green Grid data center metric
Data center infrastructure efficiency (DCiE)
Power usage effectiveness (PUE)
Compute power efficiency (CPE)
Data center density (DCD)
Data center energy productivity (DCeP)
Data center performance efficiency (DCPE)

Figure 5.3: The Green Grid Metrics

In the following paragraphs, the metrics DCiE, PUE, CPE, DCD, DCeP, and DCPE are presented.

5.3.2.2 Metrics

data center infrastructure efficiency (DCiE): this metric measures the energy efficiency of a data center. It quantifies how much energy the data center computing equipments consume from the total energy consumption. It is computed as shown in equation (5.6).

$$\text{DCiE} = \frac{\text{IT equipment power}}{\text{Total facility power}} \tag{5.6}$$

The IT equipment power is defined as the power consumed by equipments that are used to manage, process, store or route data within the compute space. The total facility power is defined as the power measured at the utility meter.

power usage effectiveness (PUE): this metric measures the energy efficiency of a data center, calculated as a ratio between the total facility power and IT equipment power. It is computed as the inverse of DCiE metric (Equation (5.6)) as shown in equation (5.7).

$$\text{PUE} = \frac{\text{Total facility power}}{\text{IT equipment power}} \tag{5.7}$$

compute power efficiency (CPE): this metric measures how efficiently the data center energy is consumed for computation. By using this metric, the power consumed by the idle servers is counted as overhead rather than as power that is being productively used [196]. It is computed as shown in equation (5.8).

$$\text{CPE} = \frac{\text{IT equipment utilization}}{\text{PUE}} \tag{5.8}$$

data center density (DCD): this metric measures the data center space efficiency. The DCD focus on a tactical part of the data center design, including the data center computing operational efficiency, and characteristics of the SLAs. It is computed as shown in equation (5.9).

$$\text{DCD} = \frac{\text{Power of all equipment on raised floor}}{\text{Area of raised floor (kW}/ft^2\text{)}} \tag{5.9}$$

The primary use for this benchmark is to determine if the deployment has low, medium, or high density. While this is a good metric to determine the absolute performance of a data center relative to other data centers, it fails to capture whether the deployment is being done effectively.

data center energy productivity (DCeP): this metric measures the number of useful computation that a data center produces based on the amount of energy it consumes. It is computed as shown in equation (5.10):

$$\text{DCeP} = \frac{\text{Useful work produced}}{\text{Total data center energy consumed}} \tag{5.10}$$

To the Green Grid (thegreengrid.org), useful work is defined by the equation (5.11).

$$UW = \sum_{i=1}^{M} V_i * U_i(t,T) * T_i \tag{5.11}$$

where M is the number of tasks initiated during the considered time period. V_i is a normalization factor that allows the tasks to be summed numerically. $T_i = 1$ if task i completes during the time period and $T_i = 0$ otherwise. $U_i(t,T)$ is a time-based utility function for each task, where the parameter t is the elapsed time from initiation to completion of the task, and T is the absolute time of completion of the task.

data center performance efficiency (DCPE): this metric measures how effective a data center is using power to provide a given level of service or work such as energy per transaction or energy per business function [23]. It is computed as a ratio between the effective IT workload and total facility power as shown in equation (5.12).

$$\text{DCPE} = \frac{\text{Effective IT workload}}{\text{total facility power}} \tag{5.12}$$

5.3.2.3 Final Remarks

All of the Green Grid metrics focus on measuring how the data center's resources are used efficiently considering the equipments and facilities but without considering the organization process and the characteristics of the deployed applications.

5.4 Summary

Over the years, large-scale computing infrastructures have been mostly driven by performance improvement, where power consumption and greenhouse gas (GHG) were usually ignored. However, power and carbon footprint have started to impose constraints in the design of computing systems and data centers [29, 34]. Thus, reducing power consumption and carbon footprint are some of the challenges that large computing infrastructures have to deal with. This requires the design of energy-aware solutions for economic and environment reasons. In this case, several metrics and benchmarks have been proposed where the mostly used are the PUE and the DCiE. These metrics provide a view of the whole power consumed by infrastructure and the energy-efficient of its resources. The main characteristic of these metrics is that they can be applied to any workload. Nevertheless, they do not take performance into account. In this case, some benchmarks can be used such as the ones discussed in sections 5.2.1.1 to 5.2.1.5.

Even though these benchmarks and metrics capture important aspects of energy-efficient solutions, more research is required to define holistic metrics and tools that will allow infrastructure operators and developers to evaluate their solutions in a broad sense.

Part II

Contributions

Chapter 6

Power-Aware Server Consolidation for Federated Clouds

Contents

Cloud services normally execute in big data centers. These data centers comprise a large number of computing nodes. One issue that arises in this scenario is the amount of energy demanded by such services [31]. In this context, we aim to investigate the usage of a cloud federation to help the cloud providers on reducing power consumption of the services, without having a great impact on service-level agreements.

Thus, in this chapter, we present the first contribution of this PhD thesis: a server consolidation strategy to reduce power consumption (Section 5.1) in cloud federation (Section 3.2.3), taking into account SLA (Section 3.1.2) requirements. We assume that clouds have limited power consumption defined by a third party agent, thereby a power capping strategy [36] applied to data centers, and that when a cloud is overloaded, its data center has to negotiate with other data centers before migrating the virtual machine. In this case, we address applications' workloads, considering the costs to turn servers on/off and to migrate the virtual machine (Section 3.1.1.3) in the same cloud and between different clouds. Simulation results with two clouds and 400 simultaneous virtual machines show that our approach can reduce up to 46% of the power consumption, while still meeting QoS requirements. This work was published in [215].

The reminder of this chapter is organized as follows. In section 6.1, we present the introduction and motivation behind this work. Section 6.2 presents the design of our server consolidation strategy. In section 6.3, experimental results are presented and discussed. Next, section 6.4 compares our proposal with the related works in the area. Finally, section 6.5 concludes this chapter.

6.1 Introduction and Motivation

Over the years, large-scale distributed systems have required flexible and scalable infrastructures. Benefiting from economies of scale and improvements of Web technologies, data centers have came out as a model to host large-scale distributed systems such as clouds [266]. As discussed in chapter 2, cloud computing is gaining popularity, since it usually employs virtualization techniques to provide large-infrastructures on-demand. Virtualization enables server consolidation. By employing server consolidation, data centers can consolidate the workloads into fewer physical nodes and switch off unused ones or put them in a low power consumption state mode (Section 5.1). Of course, the effectiveness of this technique depends on (a) how saturated the cloud system is and (b) how much slowdown the cloud applications will accept. If the cloud infrastructure is not saturated, VMs can be moved to cores that are close to each other (e.g., in the same physical machine) as shown in section 3.1.1.4, and power efficiency gains can be obtained with small application slowdowns. However, if the cloud infrastructure is saturated (i.e., there are very few cores with idle capacity), the power efficiency gains provided by server consolidation will come at the expense of severe performance losses in the applications since, in this case, several VMs may share the same core.

Many studies have been conducted to provide power reduction for cloud systems and some of them are based on server consolidation [36, 113, 334]. However, server consolidation in cloud computing can introduce some difficulties such as: (i) the cloud computing environment must provide reliable QoS, normally defined in terms of SLA; (ii) it is common to occur dynamic changes of the incoming requests rate; (iii) the usage pattern of the resources is often unpredictable; and (iv) different users have distinct preferences.

In this scenario, a multi-agent system (MAS) can be used where each participant is an autonomous agent that incorporates market and negotiation capabilities [374]. Agents are autonomous, proactive, and trigger actions by their own initiative. For these reasons, agents are suitable for coordinating the cloud market, detecting problems, opportunities and reacting to them. This capability can be used to negotiate resource usage by the users, the cloud providers, the energy power providers, and the carbon emission regulator agencies.

In order to tackle the issues discussed in the previous paragraphs, we propose the use of a multi-agent system (MAS) for federated cloud server consolidation, taking into account SLA, power consumption, and carbon footprint. In our approach, the users should pay according to the efficiency of their applications in terms of resource utilization and power consumption. Therefore, we propose that the price paid by the users should increase according to the whole energy consumption of the data center(s), especially when the users refuse to negotiate performance requirements. Experimental results show that our approach can reduce up to 46% of the power consumption, while still meeting the QoS requirements. Our experiments were realized through the CloudSim [58] simulator with two clouds and 400 simultaneous virtual machines. This work was published in [215].

The remainder of this chapter is organized as follows. Section 6.2 presents the proposed multi-agent system (MAS) server consolidation strategy for federated clouds. In section 6.3, experimental results are discussed. Section 6.4 presents some related works. Finally, section 6.5 presents some remarks and future work.

6.2 Design of the Proposed Solution

The main goal of our approach, called federated application provisioning (FAP), is to reduce power consumption of data centers, trying to meet QoS requirements, with limited energy defined by a third party agent (carbon emission regulator agency). We consider that data centers are concerned by an energy threshold, and they are in a federated cloud computing environment, scheduling online the execution of the users' applications. In this case, a multi-agent strategy is used to negotiate resources' allocations and the final price to execute the users' tasks.

We assume a federated cloud model composed of public and private clouds (Section 3.2.3), and that the cost to transfer an application or to migrate VMs across the clouds is known by the cloud providers.

In our cloud environment there are four distinct agents: cloud service provider (CLSP), cloud user (CLU), electric power provider (EPP), and carbon emission regulator agency (CERA) as shown in figure 6.1. In our design, the carbon emission regulator agency determines the amount of carbon emissions that both CLSP and EPP can emit in a period of time.

We also assume that each cloud is composed of one data center with one coordinator accountable for monitoring the metrics, negotiating with the other agents (CLSP, CLU, EPP, and CERA). There are also sensors to monitor power consumption, resource usage, and SLA violation as depicted in figure 6.2.

Finally, we consider that the cloud system has a communication layer such that any participant can exchange messages. Messages and QoS metrics are described in a format that is known by the agents, and a cloud provider cannot reject users' tasks.

The proposed scenario includes a set of data centers (clouds) composed by a set of virtual machines, which are mapped to a set of physical servers that are interconnected and deployed across the clouds. Let $R = \{r_1, r_2, \cdots, r_n\}$ be the set of resources in data center i with a capacity c_i^k, where $k \in R$. The power consumption (P_i) can be defined as [212]:

$$P_i = (p_{max} - p_{min}) * U_i + p_{min} \tag{6.1}$$

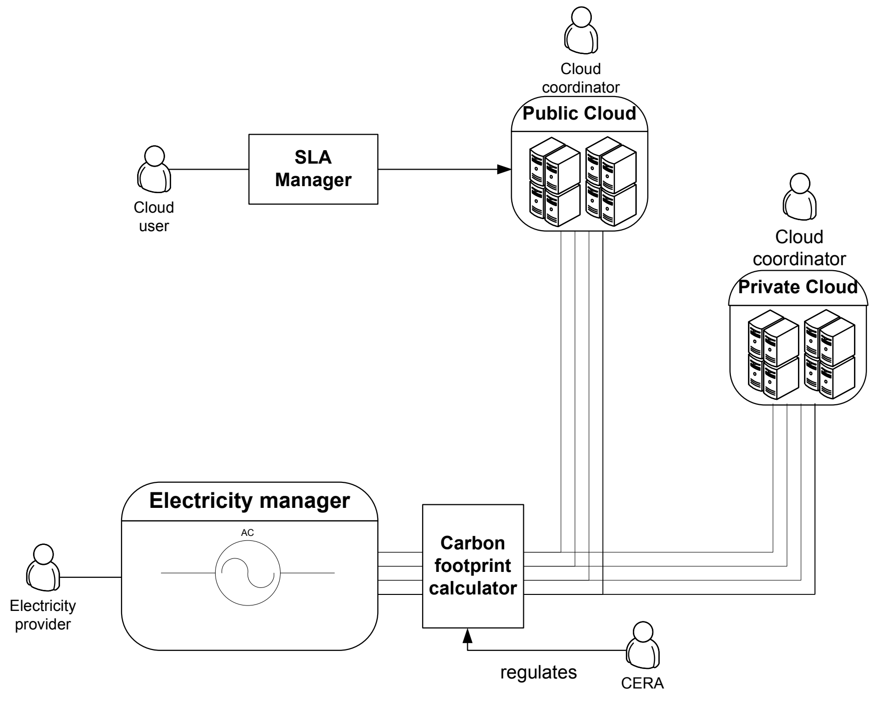


Figure 6.1: Agents of the cloud market

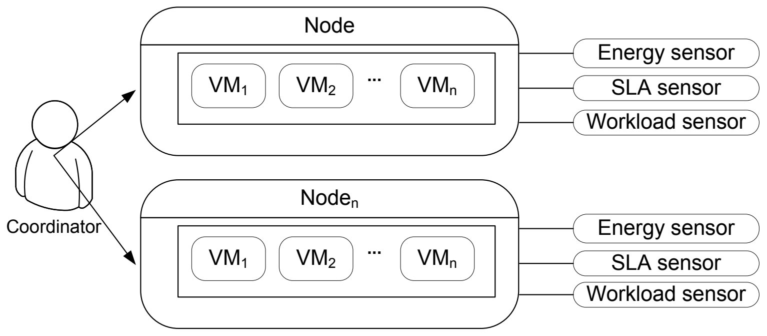


Figure 6.2: Detailed view of a data center

where p_{max} is the power consumption for the data center i at the peak load, p_{min} is the minimum power consumption in active mode, and U_i is the resource utilization of data center i as defined in equation (6.2) [212]:

$$U_i = \sum_{j=1}^{n} u_{i,j} \tag{6.2}$$

where $u_{i,j}$ is the resource usage of resource j in the data center i.

The relation between a cloud provider and a cloud user is determined by a set of QoS requirements described in the SLA (Section 3.1.2). Furthermore, data centers are subjected to an energy consumption threshold agreed among the CLSP, the EPP, and the CERA. When the energy consumption threshold is violated, this implies additional costs. To calculate the carbon footprint of the CLSP and the EPP, the CERA uses the following metrics: application performance indicators (FLOPS/kWh), data center infrastructure efficiency (DCiE), power usage effectiveness (PUE), and compute power efficiency (CPE) (Section 5.3).

Let T represent a set of independent tasks to be executed, which is subject to a set of QoS constraints such as minimum RAM memory, minimum CPU utilization, and minimum execution time. In this case, the following steps are executed:

1. when a task t_i is submitted, the cloud provider calculates the price of t_i's execution (σ_i) based on the power consumption (Equation (6.1)).

2. the cloud provider tries to place t_i in an available resource, using consolidation techniques to reduce the number of active physical servers.

3. if the cloud provider does not have enough available resources or the energy threshold will be violated, the cloud provider first contacts another cloud provider and negotiates with it the execution of this task. In this case, the price of this execution (C_t) is defined as shown in equation (6.3).

$$C_t = \sigma_t + \epsilon_t + \lambda_t \tag{6.3}$$

   where σ_t is the financial cost of executing task t based on its power consumption, ϵ_t is the cost of the power impact of a task t in the environment, and λ_t is the cost to transfer a task t to another cloud provider.

4. if the cloud provider does not succeed, it tries to consolidate its VMs considering the service-level agreements.

5. If not possible, it tries to negotiate the energy threshold with the CERA and with the EPP agents.

6. If all negotiations fail, the cloud provider finds the SLA whose violation implies in lower cost, terminates the associated task, and executes the task t_i. In this case, the price to execute the tasks is defined as shown in equation (6.4).

$$V_t = C_t + \gamma + \delta \tag{6.4}$$

where γ is the cost to violate the QoS requirements of other tasks and δ is the cost associated with the power consumption violation.

To control tasks' allocation, each cloud provider has a 3-dimensional matrix representing the tasks ($t_i \in T$), the virtual machines (vm_j), and physical servers ($r_z \in R$), where $r(i, j, z) = 1$ iff task t_i is allocated at virtual machine vm_j in resource r_z; 0 indicates that the task can be allocated in vm_j; and finally, -1 represents that the allocation is impossible.

In order to illustrate our strategy, consider a federated cloud environment with 2 clouds (DC1 and DC2) and one user that contracted one cloud to execute him/her applications. Consider that the contracted cloud (DC1) is overloaded and that the QoS requirements described in the SLAs are based on response time. In this scenario, when the user submits a set of tasks to execute, the cloud provider of DC1 first tries to execute it locally considering power consumption and its available resources. Since DC1 is overloaded, its cloud provider contacts another data center (DC2) and negotiates the execution of the tasks. If DC2 accepts, the cost of the tasks execution is calculated using equation (6.3). If DC2 refuses, then DC1 tries to consolidate its virtual machines and, if not possible, it tries to negotiate the energy threshold with the carbon emission regulator agency (CERA) and with the electric power provider (EPP) considering the following metrics (Section 5.3): application performance indicators (FLOPS/kWh), data center infrastructure efficiency (DCiE), power usage effectiveness (PUE), and compute power efficiency (CPE). If all negotiations fail, then DC1 finds the SLA whose violations implies in lower cost and terminates the execution of its associated task. Then, the cost to execute the tasks is calculated using equation (6.4).

6.3 Experimental Results

In this section, we evaluate the proposed server consolidation mechanism for federated clouds. We use the cloud simulator CloudSim [58], which is a well-established cloud simulator that has been used in many previous works [319, 375, 387], among others, for simulating resource management strategies. CloudSim is a simulation toolkit that enables modeling and simulation of cloud computing systems and application provisioning environments, with support for cloud system components such as data centers, virtual machines and resource provisioning policies.

6.3.1 Modifications in CloudSim

In order to enable federation and energy regulation capabilities, we added 4 classes to CloudSim, which are described below. CloudSim already implemented the support to measure the power consumption of the nodes.

The **CloudEnergyRegulation** class represents the behavior of the carbon emission regulator agency (CERA) agent. The CERA communicates with the data center cloud coordinator to inform the power consumption threshold.

The **DatacenterEnergySensor** class implements the **Sensor** interface that monitors the power consumption of the data center and informs the coordinator. When the power consumption is close to the limit, this sensor creates an event (i.e., CloudSim event) and notifies the coordinator. In this case, the coordinator first tries to contact another data center to transfer the virtual machines and if the data center does not accept, then the coordinator tries to consolidate them (Section 6.2).

The **FederatedPowerVmAllocationPolicy** class extends the **VmAllocationPolicy** class to implement our strategy to allocate the virtual machines across the data centers.

Finally, the **CustomerDatacenterBroker** class models the QoS requirements customer behavior, negotiates with the cloud coordinator, and requests the resources.

6.3.2 Simulation Environment

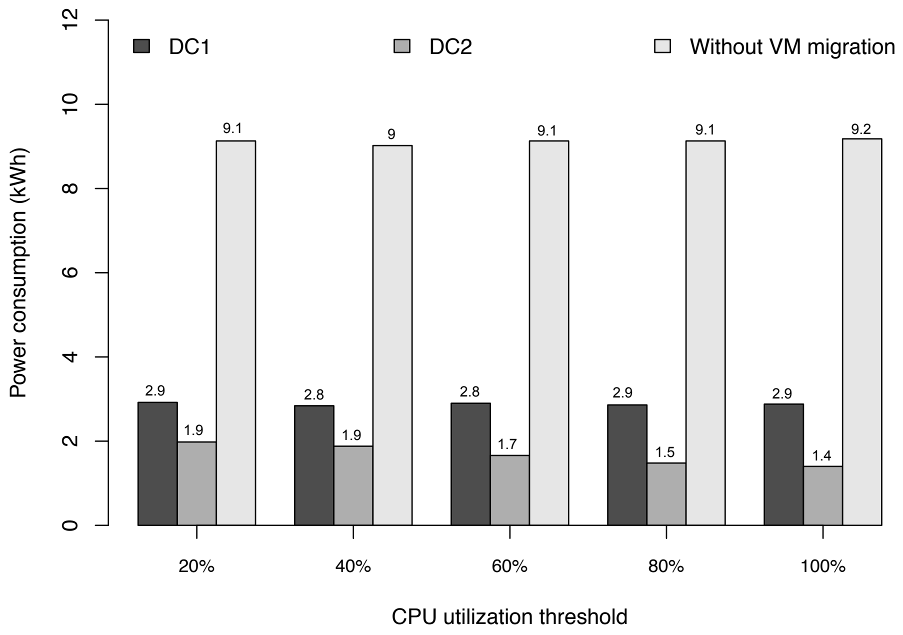
In order to evaluate the effectiveness of our federated application provisioning (FAP) technique, we used a simulation setup that is similar to the one described in [58]. Our simulation environment included two clouds, each one with one data center (DC1 and DC2) that had 100 hosts each. These hosts were modeled to have one CPU with four cores with 1000 MIPS, 2GB of RAM and 1TB of storage. The workload model included provisioning and allocating for 400 virtual machines. Each virtual machine requested one CPU core, 256MB of RAM and 1GB of storage. The CPU utilization distribution was generated according to the Poisson distribution, where each virtual machine required 150 MIPS and 1 to 10 minutes to complete execution, assuming a CPU utilization of 20, 40, 60, 80 and 100% and a global energy consumption threshold of 3 kWh of energy per data center. Initially, the provisioner allocates as many as possible virtual machines in a single host, without violating any constraint of the host. The SLA was defined in terms of response time (10 minutes).

The energy consumption threshold of 3 kWh of energy per data center was chosen based in the results of the power management technique, presented in [58].

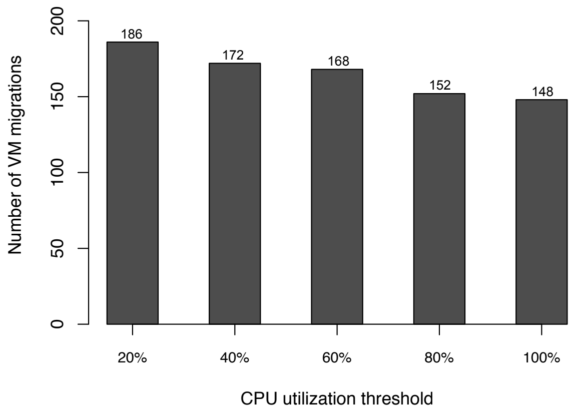
6.3.3 Scenario 1: workload submission to a single data center under power consumption threshold

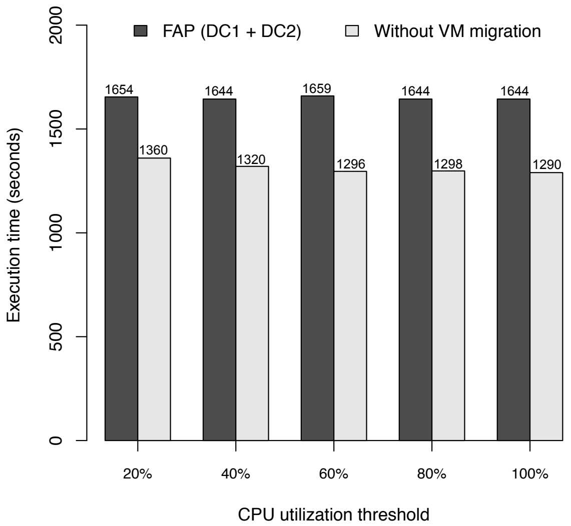
In this scenario, tasks are always submitted to data center 1 (DC1). If needed, VMs are migrated from DC1 to DC2. The simulation was repeated 10 times and the mean values for energy consumption without our mechanism using only DC1 (trivial), and with our federated application provisioning (FAP) approach are presented in figure 6.3(a).

Figure 6.3(a) shows that the proposed provision technique can reduce the total power consumption of the data centers, without SLA violation. In this case, an average reduction of 46% in the power consumption was achieved since the data center 1 (DC1) consumed more than 9 kWh with the trivial approach (without VM migration) and no more than 4.8 kWh was consumed in total by both data centers with our approach (2.9 kWh for DC1 and 1.9 kWh for DC2). In order to achieve this, data center 1 (DC1) tried first to maximize the usage of its resources and to consume the limit of its energy power threshold, without violating the SLAs. Hence, the data center 2 (DC2) was only used in imminence of SLA violation or when the energy consumption was close to violate the limit. In all cases, the energy consumption for DC1 remained close to the limit.



(a) Power consumption of the data centers with and without the FAP approach





(b) Number of VM migrated from DC1 to DC2 using the FAP approach

(c) Execution time of the tasks with and without the FAP approach

Figure 6.3: Case study 1: power consumption with 2 data centers under limited power consumption

Figure 6.3(b) presents the number of VMs migrated when our mechanism is used. It can be seen that the number of migrations decreases as the threshold of CPU usage increases. This result was expected since with more CPU capacity, the allocation policy tends to use it and to allocate more virtual machines in the same host.

In figure 6.3(c), we measured the wallclock time needed to execute 400 tasks, with and without our mechanism (FAP). It can be seen that the proposed provision technique increases the whole execution time. This occurs because of the overhead caused by VMs migrations between the data centers, and the negotiations between the CLU and the CLSP. Nevertheless, this increase is less than 22%, where the wallclock execution times without and with the FAP mechanism are 21.5 minutes and 27.4 minutes, respectively, when using the whole CPU capacity. We consider that this increase in the execution time is compensated by the reduction in the power consumption (Figure 6.3(a)).

6.3.4 Scenario 2: distinct workload submission to different overloaded data centers

In this scenario, we consider two users, with distinct SLAs and each user submits 400 tasks to different data centers (DC1 and DC2). Our goal is to observe the rate of SLA violation when the workload of both data centers is high. The energy consumption of the data centers is presented in figure 6.4(a).

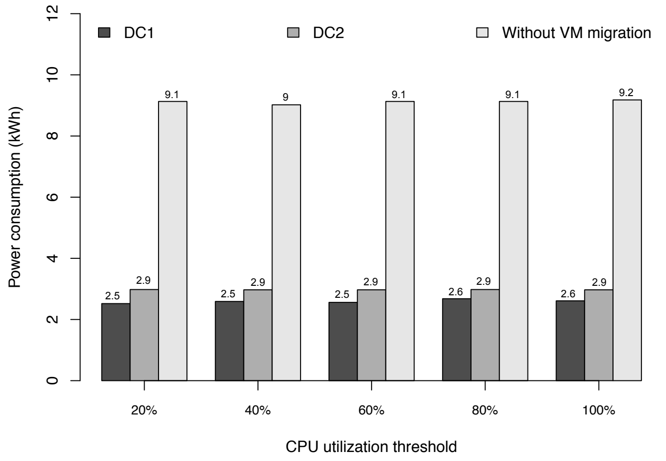
In figure 6.4(a), we can see that, even in a scenario with overloaded data centers, our mechanism can maintain the power consumption below the threshold (3 kWh) for each data center. Using the whole CPU capacity, the power consumption decreased from 9.2 kWh to 5.5 kWh (DC1 + DC2), reaching a reduction of 40% in the power consumption.

Figures 6.4(b) and 6.4(c) show the number of VM migrations between the data centers and the wallclock time to execute 800 tasks when both data centers are overloaded. Comparing with the scenario with one overloaded data center (DC1), the number of VM migrations decreased, keeping almost the same penalty in the execution time (23%) due to the negotiations overhead between the agents and by server consolidations.

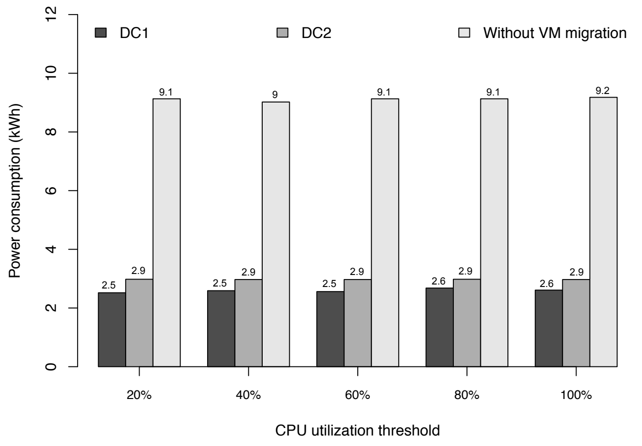
The number of SLA violations with two overloaded data centers was lower than with just one data center (DC1) as we can see in figure 6.4(d). With the CPU utilization threshold of 80%, the SLA violation decreased from 43.9% (DC1) to 31.4% (DC1 + DC2), reaching 28% of reduction in the SLAs violations. This shows the appropriateness of VM migration between different data centers in an overloaded scenario.

6.4 Related Work

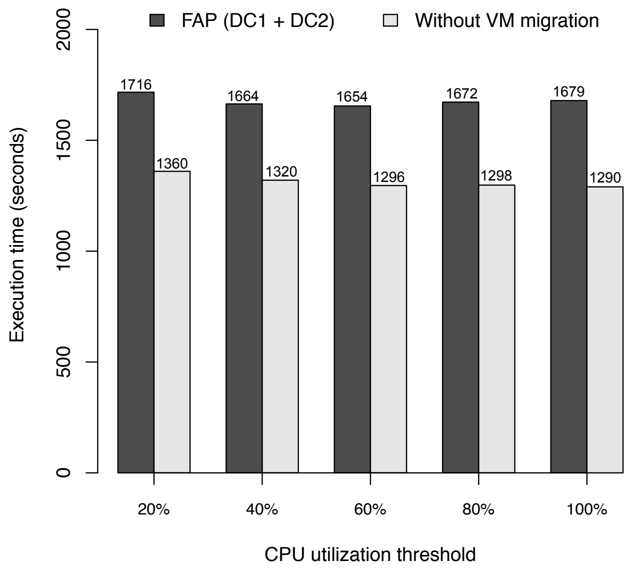
Many studies have tried to improve the power efficiency of a computing system by minimizing the static power consumption while trying to increase the performance proportionally to the dynamic power consumption. As a result, the hardware energy efficiency has significantly improved (Section 5.1). However, whereas hardware is physically responsible for most of the power consumption, hardware operations are guided by software, which is indirectly responsible for the energy consumption [6].



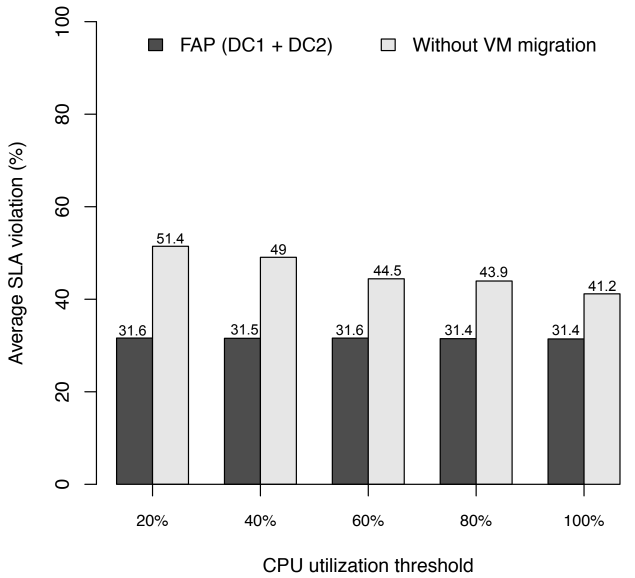
(a) Power consumption of the data centers with and without the FAP approach



(b) Number of VM migrated between the data centers with the FAP approach



(c) Execution time of the tasks with two overloaded data centers with and without the FAP approach



(d) Average SLA violation with and without the FAP approach

Figure 6.4: Case study 2: power consumption of two overloaded data centers under limited power consumption

Some of the studies reported in literature try to minimize the power consumption from the data center perspective, considering that the main reason for energy inefficiency is resource underutilization. One of the first approaches to try to solve this problem consists of shutting down idle nodes [193, 251] and waking them up when the workload increases or the average QoS violation ratio exceeds a threshold.

At the hardware level, improvements are made turning off components, putting them to sleep or changing their frequency using dynamic voltage and frequency scaling (DVFS) techniques (Section 5.1). For example, in [195], a power-aware DVFS based cluster scheduling algorithm is presented taking into account performance constraints. The proposed algorithm selects the appropriate supply voltages that minimize energy consumption of the resources. Simulation results show that the proposed scheduling algorithm can reduce the power consumption with an increase in the execution time.

In [155], the authors describe a virtual machine placement framework called Entropy. Entropy aims to minimize the number of physical hosts to allocate the virtual machines, without violating any constraints (e.g., memory size and number of CPUs). Its placement process comprises two phases. The first phase identifies the nodes that have sufficient resources (i.e., RAM and CPU) to host a VM, and the second one allocates the VMs trying to minimize both the number of physical hosts and the number of VM migrations. These two phases use constraint programming to find out a feasible global solution. Experiments realized in the Grid'5000 testbed (www.grid5000.fr) show that constraint programming outperforms the first-fit decreasing (FDD) algorithm with regard to the number of VM migrations and power savings. In [103], the authors use Entropy to allocate VMs in a federated cloud environment, taking into account power consumption and CO_2 emissions. Experimental results show a power saving of almost 22% when considering only power and a saving of almost 19% when the allocations considering both power and CO_2 emissions. The experiments considered two federated clouds, and they used two synthetic workloads.

In [356], the authors present pMapper, a power and VM placement framework. Its architecture comprises three different managers: performance, migration, and power. In pMapper, a sensor collects the current performance and power characteristics of both virtual and physical machines. Then, it sends these data to the performance and power managers. After, the performance manager analyses the data, and based on SLA violations, it suggests to resize the VMs. Similarly, the power manager based on the current power consumption suggests power throttling actions (e.g., DVFS). Based on these suggestions, an arbitrator component selects a configuration, and defines the physical machines to host the VMs, as well as the characteristics of each VM. Finally, the managers resize and migrate the VMs. Since heterogeneous platforms are considered, each manager consults a knowledge database to determine the cost of a VM migration in the performance of its applications, as well as in the power consumption. pMapper implements three algorithms called: min Power Parity (mPP), min Power Placement with history (mPPH), and PMaP. The mPP algorithm attempts to allocate the VMs in order to minimize the total power consumption, without taking into account the current placement of the VMs. Hence, the mPP algorithm results in a high number of migrations. The mPPH, on the other hand, extends the mPP to minimize VM migrations. However, its efficiency with regard to minimizing the power consumption is low. Finally, the PMaP tries to find out an allocation that minimizes both power consumption and VM migrations. Experimental results show that the mPP and mPPH algorithms can reduce 25% of the power consumption when the utilization ratio is at most 75% of the cluster's capacity.

In [75], a market-based multi-agent resource allocation model is presented. The resource allocation model aims to provide an effective resource allocation policy through a genetic algorithm in a cloud environment. Buyer and service provider agents determine the bid and ask prices using interactions to find an acceptable price, considering the demands, the availability of clouds' resources and the constraints of the cloud users and/or service providers. Simulation results show that this approach could increase the welfare of both buyers and cloud providers.

In [93], the authors address the coordination of multiple autonomic managers for power and performance trade-offs in a real data center environment, with a real HTTP traffic and time-varying demand. By turning off servers under low load condition, the proposed approach achieved power savings of more than 25% without incurring in SLA penalties.

In [212], two energy-conscious task consolidation heuristics (ECTC and MaxUtil) are used to maximize resource utilization for power saving. The cost of the ECTC heuristics is computed considering the energy consumption to run a group of parallel tasks. The MaxUtil heuristic tries to increase the consolidation density. Simulation results show that the proposed heuristics are able to save energy by 18%(ECTC) and 13%(MaxUtil).

In [58], CloudSim is used to simulate VM provisioning techniques. Experimental results compare the performance of two energy-conscious resource management techniques (DVFS and an extension of DVFS policy). In the DVFS policy, VMs were resized according to the host's CPU utilization. In the extension of DVFS, VMs were migrated every 5 seconds using a greedy algorithm that sorts the VMs in decreasing order of CPU utilization. In both of them, each VM was migrated to hosts that kept resources utilization below a threshold. Experimental results show that the total power consumption of a data center reduced up to 50%, but with an increase in the number of SLA violations.

In [393], Zhou and colleagues propose and evaluate a service scheduling approach, called Random Dynamic Scheduling Problem (RDSP), to reduce energy consumption in cloud computing environments. This approach uses Monte Carlo sample historical data to approximate to the predictable user demand and a probabilistic model to express QoS requirements in a homogeneous environment, where the servers' power consumption is constant. Using numeric validation and Monte Carlo sampling to estimate the users' demand, the results show that the proposed scheduling strategy was able to decrease the power consumption of the server when the user demand is predictable.

In [59], the authors present a VM consolidation policy for cloud computing. The proposed policy aims to minimize power consumption taking into account QoS requirements. And, it extends the Minimum Power policy [35] in order to minimize VM migrations and to maximize resource usage. In this case, different from the Minimum Power policy, a VM is migrated only when its node is overloaded and with SLA violations. Experimental results show a reduction in the power consumption (up to 34%) and in the execution time (63%). Moreover, the new policy increases SLA guarantees. The experiments were realized through the CloudSim simulator considering one data center with 800 physical nodes.

Table 6.1 summarizes the ten approaches discussed in the previous paragraphs. In the last line, we present the characteristics of our work. As can be seen in this table, two approaches [75, 93] use multi-agent systems to reduce power consumption and costs. One of them targets a cloud environment and the other one a cluster computing environment.

Five works [58, 59, 103, 212, 393] reduce power consumption in cloud computing considering SLAs. Only one proposal [103] deals with cloud federation to implement workload consolidation. Nevertheless, it does not implement negotiation mechanisms between the data centers, and VMs are always migrated when one cloud is overloaded. None of these ten proposals tackle federated cloud environments for power-aware allocations that are SLA-conscious, and the workload migration requires negotiation between the data centers.

Table 6.1: Comparative view of cloud server consolidation strategies

Paper	Target	Federated	Multi-agent	Migration	Negotiation	SLA
[195]	Cluster	No	No	No	No	No
[155]	Cluster	No	No	Same DC	No	No
[103]	Cloud	Yes	No	Among DCs	No	Yes
[356]	Cluster	No	No	Same DC	No	No
[75]	Cloud	No	Yes	No	No	No
[93]	Cluster	No	Yes	No	No	No
[212]	Cloud	No	No	No	No	Yes
[58]	Cloud	No	No	Same DC	No	Yes
[393]	Cloud	No	No	No	No	Yes
[59]	Cloud	No	No	Same DC	No	Yes
This work [215]	Cloud	Yes	Yes	Among DCs	Yes	Yes

6.5 Summary

In this chapter, we proposed and evaluated a server consolidation approach for efficient power management in federated clouds, taking into account energy consumption and QoS requirements.

Using simulated data with two clouds and 400 simultaneous virtual machines, we showed the benefits of distributing the workload across clouds, under limited power consumption. In this case, the best gain was obtained when one cloud was overloaded, and it migrates part of its workload to the second one, reducing the power consumption from 9.3 kWh (cloud 1) to 4.8 kWh (cloud 1 and cloud 2) with an increase of less than 22% in the execution time. The proposed approach is consistent with other researches that also envisioned the usage of transient resources [19, 289, 324, 379].

Even though we achieved good results with our approach, other variables should also be considered such as the workload type, the data center characteristics (i.e., location, power source), and the network latency as these variables can affect the whole power consumption of a data center. Moreover, resource heterogeneity should also be considered as data centers usually comprise heterogeneous resources that can have different power consumption and capabilities. This requires energy and performance-aware load distribution strategies, and we leave this extension for future work.

Chapter 7

Biological Sequence Comparison at Zero-Cost on a Vertical Public Cloud Federation

Contents

The previous chapter showed us that cloud federation can help the providers on reducing the power consumption of the clouds, by migrating the tasks according to the data center workload and the SLA violation ratio. In this chapter, we aim to use the cloud federation considering the software developers viewpoint. In other words, we aim to aggregate resources from different type of cloud providers (e.g., PaaS and IaaS) to execute a biological sequence comparison application at reduced-cost. Therefore, we present and evaluate a cloud architecture to execute a native cloud application on federated clouds at zero-cost. Our architecture follows a hierarchical and distributed management strategy to connect and to manage the services offered by PaaS and IaaS clouds, thus, being an architecture for vertical federated clouds (Section 3.2.3.2). The application is a biological sequence comparison application, which was implemented following the MapReduce (Section 3.1.3) programming model.

Experimental results show that, by using the resources of five different public clouds, we could run our application over a huge genomics database in time that is comparable with the one obtained in execution on multi-core clusters and Cell/BEs, showing the appropriateness of our vertical cloud architecture.

The reminder of this chapter is organized as follows. In section 7.1, we present the introduction and motivation behind this work. After, section 7.2, briefly introduces biological sequence comparison and the Smith-Waterman (SW) algorithm. Next, section 7.3 presents the design of our architecture. In Section 7.4, experimental results are presented and discussed. Section 7.5 presents some of the related works that have executed the SW algorithm on different platforms. Finally, section 7.6 concludes this chapter.

7.1 Introduction and Motivation

Usually, scientific applications comprise a very large number of independent tasks that can be executed in parallel. These applications are called embarrassingly parallel or simply parameter sweep applications. For example, bioinformatics applications usually perform genomics database searches to find the similarity between a query sequence and the sequences in a database. As a genomics database may have hundreds of thousands of sequences, this work can be performed in parallel, where each task compares the same query sequence with a different sequence in the database. In other words, all tasks execute the same program, with different input sequences. This can be implemented with some programming models or frameworks available for processing large datasets. MapReduce is one of such programming models that can execute in many computing infrastructures, often using Hadoop (Section 3.1.3).

One of the most popular bioinformatics algorithms is the Smith-Waterman [327] (SW) algorithm. Smith-Waterman is an exact algorithm that can obtain the best score and alignment between two sequences of size n in quadratic space and time. Due to the quadratic complexity, it may require large amounts of memory to store its dynamic programming matrices and also a considerable computing time.

A great number of efforts has been made to accelerate the SW algorithm using high performance computing platforms. These efforts include clusters ([261, 290]), Cell/BEs ([7, 385]), and GPUs ([97, 229]), among others. These platforms could significantly reduce the execution times of SW by using elaborate and often complex programming techniques.

Recently, cloud computing has been considered to execute HPC applications due to its characteristics such as pay-per-usage and elastic environment. In order to support a large number of consumers or to decentralize the solution, clouds can be combined forming a cloud federation. As discussed in section 3.2.3 and in chapter 6, in a federated environment, clouds interact and negotiate the most appropriate resources to execute a particular application/service. This choice may involve the coordination and orchestration of resources that belong to more than one cloud, which will be used, for instance, in order to execute huge applications.

Although cloud federation has many advantages (Section 3.2.3), using it to execute huge applications is usually a difficult task for many reasons. First, clouds' resources are usually heterogeneous and they may change over time. Second, the clouds often have different APIs, which requires extra work from the users to understand and to use them.

Third, the resources mostly have unpredictable performance, even for resources of the same type. Fourth, most of the users' applications are cloud-unaware, which may not fit the constraints of a cloud environment.

In this context, MapReduce (Section 3.1.3) may help us to dismiss these difficulties as it provides an application model that can run on multiple clouds. Also, its implementation is decoupled from the cloud provider or any particular infrastructure. Finally, it can easily handle performance changing and node failures.

As noted in section 2.5, clouds usually employ the pay-as-you-go model. Nonetheless, the majority of the clouds also provide resources at zero-cost, in a category called usually as "free-quota". The resources from the free-quota category have limited capacity such as low CPU frequency and a small amount of RAM memory. Thus, we want to investigate if a federated execution exclusively in the free-quota (i.e., at zero-cost) can yield good execution times.

Therefore, in this chapter we describe our approach to execute the SW algorithm in a cloud federation. Since the cloud providers are unaware of the federation, our proposal is classified as a multi-cloud (Section 3.2.3.2). Our approach has two main components: (a) an architecture that can transparently connect and manage multiple clouds, thus creating a federated cloud environment; and (b) an implementation of the MapReduce version of SW in this architecture. The proposed architecture is hierarchical in such a way that one of the clouds is the cloud coordinator, which distributes the tasks to be executed; and the other clouds are the workers responsible for executing a MapReduce version of the SW algorithm over the piece of computation received from the coordinator. This work was published in [214].

The reminder of this chapter is organized as follows. First, we provide a brief introduction for biological sequence comparison followed by a description of the Smith-Waterman algorithm (Section 7.2). Next, we present the proposed architecture (Section 7.3) and the experimental results realized in a public cloud federation scenario (Section 7.4). Finally, we discuss some of the related works that have executed the SW algorithm in different platforms (Section 7.5) followed by some considerations and future work.

7.2 Biological Sequence Comparison

A biological sequence is represented by a linear list of residues, which are nucleotide bases (for DNA and RNA sequences) or amino acids (for protein sequences). To compare biological sequences, we need to find the best alignment between them, which is to place one sequence above the other making clear the correspondence between similar residues from the sequences [104].

Given an alignment between two sequences s and t, a score can be associated for it as follows. For each two bases in the same column, we associate (a) a punctuation ma, if both characters are identical (match); or (b) a penalty mi, if the characters are different (mismatch); or (c) a penalty g, if one of the characters is a space (gap). The score is the sum of all these values and the maximal score is called the similarity between the sequences.

Figure 7.1 illustrates a possible alignment and score between DNA sequences: S_0 = ACATGTCCGAG and S_1 = ATTGTCAGGAG.

$$\underbrace{\begin{array}{cccccccccccc} A & C & A & T & G & T & C & C & G & - & A & G \\ A & - & T & T & G & T & C & A & G & G & A & G \\ \hline +1 & -2 & -1 & +1 & +1 & +1 & +1 & -1 & +1 & -2 & +1 & +1 \end{array}}_{score = 2}$$

Figure 7.1: Computing the alignment and the score of two biological sequences, where $ma = +1$, $mi = -1$ and $g = -2$

If proteins are being compared, a substitution matrix of size 20 x 20 is used to store the match/mismatch punctuation. The most commonly used substitution matrices are PAM and BLOSUM [255].

7.2.1 The Smith-Waterman Algorithm

The Smith-Waterman algorithm (SW) [327] is an exact algorithm based on dynamic programming to obtain the best local alignment between two sequences in quadratic time and space. It is divided in two phases: create the similarity matrix and obtain the best local alignment.

The first phase receives as input sequences s and t, with $|s| = m$ and $|t| = n$, where $|s|$ represents the size of a sequence s. Typically, s and t can range from few characters to thousands of characters. The notation used to represent the *n-th* character of a sequence *seq* is $seq[n]$ and, to represent a prefix with n characters, from the beginning of the sequence, we use $seq[1..n]$. The similarity matrix is denoted as A and each element contains the similarity score between the prefixes $s[1..i]$ and $t[1..j], i < m + 1 \land j < n + 1$.

At the beginning, the first row and column are filled with zeros. The remaining elements of A are obtained from equation (7.1). The SW score between sequences s and t is the highest value contained in matrix A.

$$A_{i,j} = \max \begin{cases} A_{i-1,j-1} + (\text{if } s[i] = s[j] \text{ then } ma \text{ else } mi) \\ A_{i,j-1} - g \\ A_{i-1,j} - g \\ 0 \end{cases} \tag{7.1}$$

The second phase is executed to obtain the best local alignment. The algorithm starts from the cell that contains the highest value and follows the arrows until a zero-valued cell is reached. A left arrow in $A_{i,j}$ (Figure 7.2) indicates the alignment of $s[i]$ with a gap in t. An up arrow represents the alignment of $t[j]$ with a gap in s. Finally, an arrow in the diagonal indicates that $s[i]$ is aligned with $t[j]$.

In the next section, we describe how the SW is executed in a multi-cloud environment.

	*	G	A	G	C	T	A	T	G	A	G	G
*	0	0	0	0	0	0	0	0	0	0	0	0
T	0	0	0	0	0	1	0	1	0	0	0	0
A	0	0	1	0	0	0	2	0	0	1	0	0
T	0	0	0	0	0	1	0	3	1	0	0	0
A	0	0	1	0	0	0	2	1	2	2	0	0
G	0	1	0	2	0	0	0	1	2	1	3	1
G	0	1	0	1	1	0	0	0	2	1	2	4
T	0	**0**	0	0	0	2	0	1	0	1	0	2
A	0	0	↖**1**	0	0	0	3	1	0	1	0	0
G	0	1	0	↖**2**	0	0	1	2	2	0	2	1
C	0	0	0	0	↖**3**	1	0	0	1	1	0	1
T	0	0	0	0	1	↖**4**	2	1	0	0	0	0
A	0	0	1	0	0	2	↖**5**	3	1	1	0	0

Figure 7.2: Smith-Waterman similarity matrix for sequences s = TATAGGTAGCTA and t = GAGCTATGAGG. In this example, the alignment has $score = 5$.

7.3 Design of our Federated Cloud Architecture

Our architecture assumes a hierarchical and multi-cloud environment. In addition, we assume that resource usage is limited. This limitation is not due to the cloud environment itself, but it is associated with financial constraints. In an extreme scenario, the use of cloud resources by the applications is limited by the free-quota offered by each cloud.

In our architecture, there is a *coordinator* responsible for the whole execution, and for interacting with the user and the multiple clouds. Each $cloud_i$ has a *cloud master* and a set of *slave instances*. The *cloud master* is responsible for executing the tasks assigned to its cloud; and the *slave instances* are the ones that actually execute the tasks.

The proposed architecture, depicted in figure 7.3, comprises four layers: *User*, *Storage*, *Communication*, and *Execution*.

The *User* layer provides a graphical user interface (GUI) to submit the MapReduce applications, as well as the input files. In the case of the SW application, there are two input files. The first file contains a set of biological query sequences and the second one contains a genomics database, which is composed of a huge set of biological sequences.

At the *Storage* layer, the input files are persisted into a storage system and the tasks are created using the MapReduce model. In this case, we create one task for each entry of the map function and enqueue the task into the *coordinator* queue in the reduce function. In other words, a task executes one SW comparison.

The *Communication* layer implements transparent communication among the clouds and transparent access to the cloud database. In this layer, generic requests are translated to specific requests according to the *cloud master* environment. This creates a homogeneous API to access the federated resources, and it reduces the needs of the users knowing about the limitation of each cloud provider such as maximum request size, request timeouts, and commands to interact with the environment.

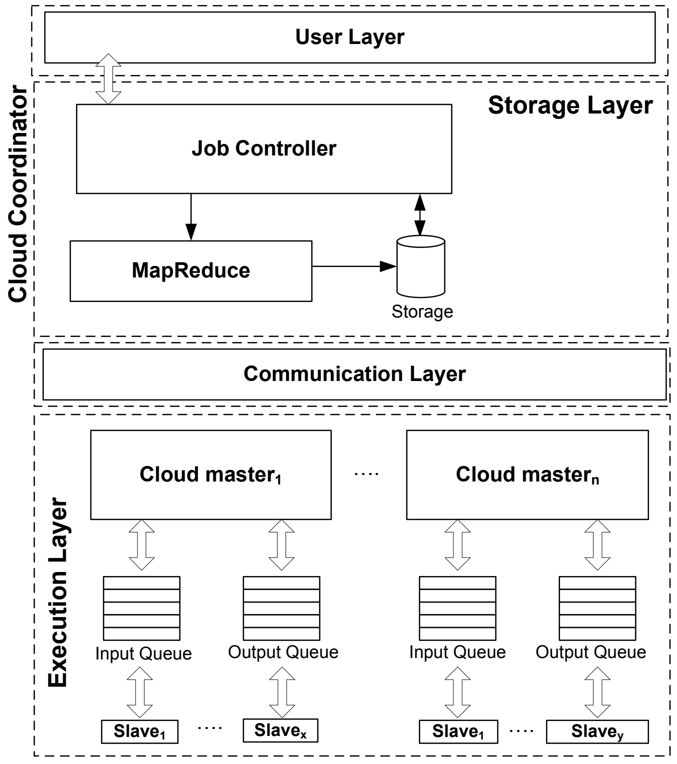


Figure 7.3: Federated cloud architecture to execute MapReduce applications

In each cloud, there is an input and an output queue which are shared by the *slave instances*, containing the tasks to be executed. At the *Execution* layer, each *cloud master* receives the tasks and stores them into the input queue. Slave machines retrieve tasks from this queue, process them and store the result into the output queue.

The architecture uses the HTTP protocol to implement message exchanges between the masters and the slaves. A well-defined interface is defined to accept either data requests or computational task requests. The body of a message can be either XML or JSON following the REST architecture style [116]. The HTTP methods are used as following: (i) the POST method is used to submit a task; (ii) the GET method is called to obtain information about the cloud or about the status of the a task; and (iii) the DELETE method is used to delete a task. Figure 7.4 presents the types of messages and their associations.

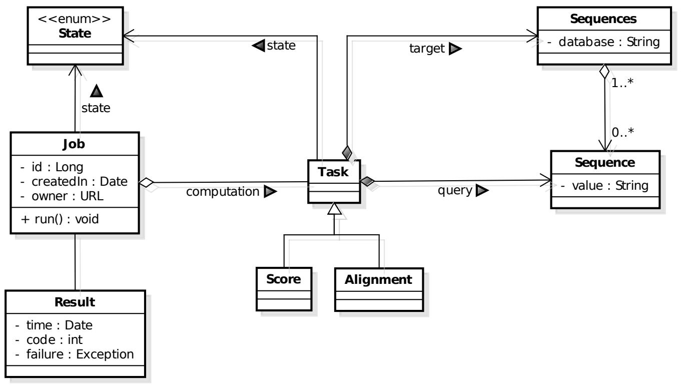


Figure 7.4: Type of the messages exchanged in our multi-cloud architecture

As shown in figure 7.4, a *Job* has at least one *Task*, which represents the computation work to be executed, a *State* and an *owner*. The owner of a job is always the user who submitted the job. The states of a task can be: *created*, *submitted*, *executing*, and *finished*. When a task finishes its execution, it updates the *Result* attribute with the following data: the total execution time, the result code (i.e., 200 - successful or 400 - error), the output, and the failure data.

7.3.1 Task Generation with MapReduce

The tasks are generated following the MapReduce approach. The input of the map phase is an input file and its values are the tasks to be executed. In this case, the map function is a data preparation phase. It creates all tasks with the necessary data in such a way that the reduce function can submit it to the available cloud masters. The reduce function is just one notification to the cloud master informing about the tasks to execute.

When the *cloud master* receives a notification about the tasks, it responds informing that it is alive and ready to receive them. Then, the *coordinator* submits the tasks to the *cloud master* as long as the response of its notification is 202 (i.e., accepted). The state of all submitted tasks is changed to *submitted* and the task monitor starts the execution. The *coordinator* monitors the execution of the tasks by contacting the masters, and in case of failure or if another master responds that it is idle, the *coordinator* selects the tasks in the states *created* or *submitted*, and assigns them to the idle *cloud master*.

When the *cloud master* receives the tasks, it stores them in its input queue, which is accessed by the slaves instances. Then, the slaves execute the tasks and write the result into the output queue. Finally, the master gets the results from the output queue and sends them to the *coordinator*.

In order to illustrate our approach, consider that a user wants to compare protein sequences with a genomics database, as depicted in figure 7.5. In this scenario, he/she submits a file with the sequences to be queried and the database to be compared using a Web application, at the *User* layer (Figure 7.5 (1)). The Web application creates a job with tasks of type *score* and sends it for the *coordinator*, at the *Storage* layer. When the *coordinator* receives this job, it analyses it and executes the following steps. First, it uses the MapReduce module (Figure 7.5 (2)) to persist the sequences (database and sequences to be queried) in the data storage (Figure 7.5 (3)). Second, it creates small tasks to be executed by the clouds using the MapReduce model (Figure 7.5 (3)). Third, it submits the tasks to the *cloud master* of each cloud (Figure 7.5 (4)). After the submission, the *coordinator* monitors the execution of the tasks by contacting the cloud masters and, and if necessary, it re-assigns tasks to another master. Finally, when the computation finishes, the master sends the results to the *coordinator* (Figure 7.5 (5)), which notifies the user about the result of his/her tasks (Figure 7.5 (6)).

7.3.2 Smith-Waterman Execution

To implement the Smith-Waterman algorithm, we used the MapReduce model, where in the map phase the slave instances dequeue the tasks from the input queue, and execute them, whereas in the reduce function they retrieve the result of the map function and iterate over it to find the maximum score. Then, they write the output into the output

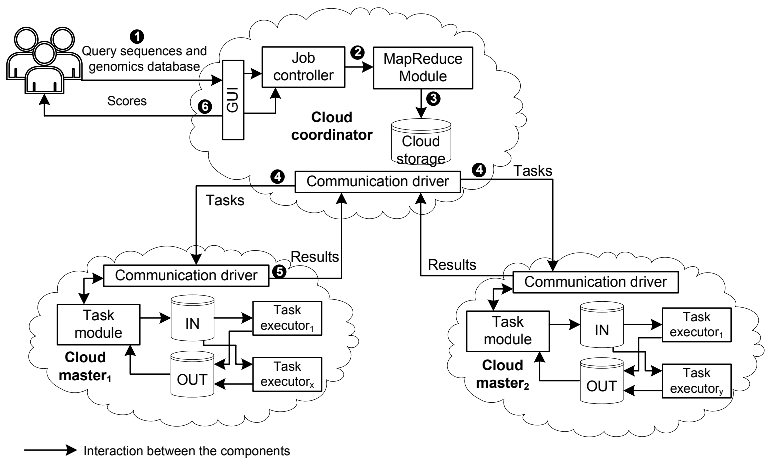


Figure 7.5: Comparing protein sequences with a genomics database on multiple clouds

queue. When a task starts, its state changes to *running* and the *cloud master* is notified about it. Figure 7.6 illustrates this process.

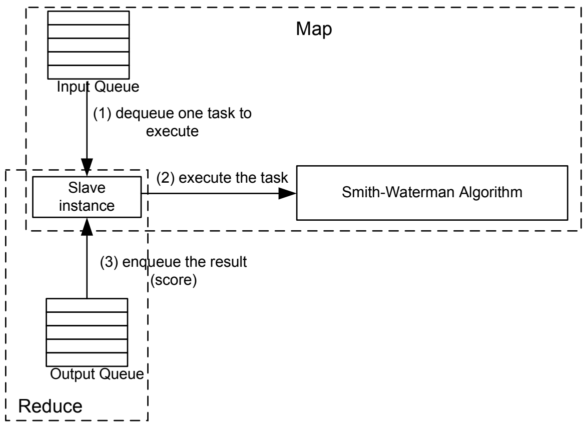


Figure 7.6: Smith-Waterman execution following the MapReduce model

7.4 Experimental Results

The architecture and the SW algorithm were implemented in Java. We evaluated these implementations in a multi-cloud environment composed of five public clouds: Amazon Elastic Compute Cloud (aws.amazon.com/ec2), Google App Engine (appengine.google.com), OpenShift (openshift.redhat.com), Heroku (heroku.com), and PiCloud (picloud.com). In our tests, we used only the free quota of these clouds and that resulted in the configuration depicted in table 7.1.

We executed SW in each cloud separately (i.e., standalone mode) and in the federated mode (5 clouds). In the following paragraphs, we will discuss how our architecture was implemented in each cloud.

In EC2, Heroku and PiCloud, we implemented the architecture as proposed in figure 7.3 using the standard Hadoop as the MapReduce implementation.

Table 7.1: Configuration of the clouds to execute the SW algorithm

Cloud	**Configuration**
Amazon Elastic Cloud (EC2)	24 applications in the micro instances
Google App Engine (GAE)	10 applications with 2 instances per application
Heroku	1 application deployed in the cedar stack configuration
OpenShift	1 application deployed in the express configuration
PiCloud	1 application deployed in one Intel Xeon 2.66GHz, 8GB RAM

In OpenShift, we opted to use a combined Hadoop+JMS strategy. In this case, the master application and slave instances use the Java Message System (JMS) to share the work to be executed. When the master receives a task, it performs the steps described in section 7.3, and the slaves are JMS consumers of the master queue.

Google App Engine (GAE) imposed many restrictions to design the application using the free quota such as: (i) the request payload was limited to 32KB; (ii) the maximum number of requests was 10,000 and (iii) the applications do not share resources (i.e., the database). Since genomics databases usually have hundreds of MBytes and the database must be shared, these restrictions invalidated the placement of the database inside GAE. Therefore, we opted to place the cloud master in Amazon EC2 and the database in Amazon S3, generating a hybrid version in this case.

For the multi-cloud, Amazon EC2 was chosen as the coordinator and the other clouds acted as cloud masters.

In our tests, we compared up to 24 query sequences with the database UniProtKB/Swiss-Prot (November 2011), publicly available at uniprot.org, composed of $532,794$ sequences (252.7 MB) with our SW cloud implementation (Section 7.3.2), using the substitution matrix BLOSUM50 [255]. Table 7.2 presents the accession numbers and the sizes of the real query sequences used in the tests. As can be seen in this table, the sequence sizes ranged from 144 amino acids (shortest sequence) to $5,478$ amino acids (longest sequence).

Table 7.2: Query sequences compared to the UniprotKb/Swiss-Prot genomics database

Sequence	**Length**	**Sequence**	**Length**	**Sequence**	**Length**
P02232	144	P01111	189	P05013	189
P14942	222	P00762	246	P07327	375
P01008	464	P10635	497	P25705	553
P03435	567	P42357	657	P21177	729
O60341	852	P27895	1000	P07756	1500
P04775	2005	P19096	2504	P28167	2005
P0C6B8	3564	P20930	4061	P08519	4548
Q7TMA5	4743	P33450	5147	Q9UKN1	5478

Figure 7.7 presents the wallclock execution times for the five standalone clouds and the multi-cloud approach. In this test, we are comparing the 24 protein sequences listed

in table 7.2 with the entire UniProtKB/Swiss-Prot genomics database. Therefore, we executed 12,787,056 Smith-Waterman comparisons.

In figure 7.7, we can see that, for the standalone clouds, the lower execution time was achieved by EC2 (4,800 seconds). The execution time in a multi-cloud configuration was 3,720 seconds, which is 22.55% lower than the best standalone execution time. The lower execution time in the Amazon EC2 reflects the small overhead of the infrastructure for communicating the master, the storage, and the consumers, all developed as Hadoop jobs. The overhead in this case is small because master and storage are in the same cloud. Moreover, the throughput between EC2 and S3 is on average 3Gbps.

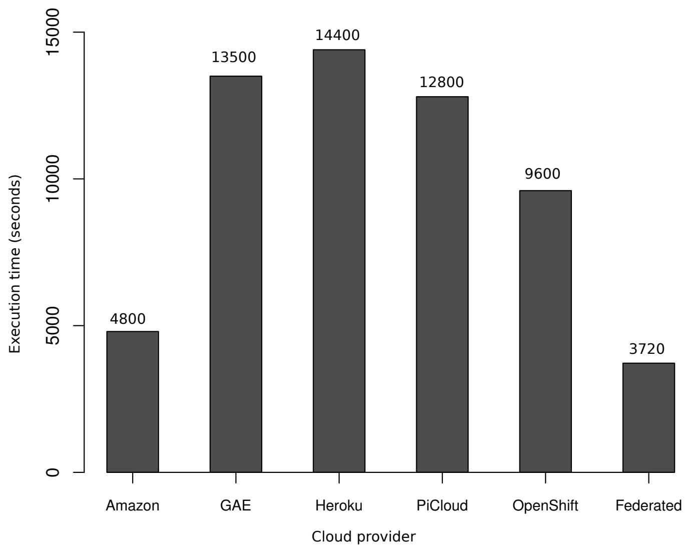


Figure 7.7: Execution time for 24 sequence comparisons with the Uniprot/SwissProt database

We also compared the execution time of our SW implementation running on EC2 with the execution time of the SSEARCH program. The SSEARCH program is an implementation of SW that belongs to the FASTA suite and is publicly available at *www.ebi.ac.uk/Tools/sss*. We downloaded the SSEARCH binary on November 2011 and executed it sequentially on an Intel Core 2 Duo 2.4Ghz, 8GB RAM. Due to time constraints, we only compared the longest sequence (Q9UKN1 in table 7.2) in this test. The sequential comparison took 5 hours, 44 minutes and 14 seconds (20,682 seconds) whereas the EC2 execution, at zero-cost, took 13 minutes (780 sec) as shown in figure 7.8. Therefore, EC2 achieved a speedup of 26.51x over the SSEARCH sequential execution in this comparison.

Figure 7.9 presents the GCUPs (billions of cell updates per second) for the five standalone clouds and the federated cloud execution. In this figure, we can see that the best value was achieved in the multi-cloud configuration (0.5 GCUPs), only within the free quota, which is comparable to the ones obtained by multi-core cluster and Cell/BE approaches (Table 7.3), but with the difference that in our approach we compared 24 query sequences with one huge database (UniProtKB/Swiss-Prot). For the standalone clouds, the best value was achieved by EC2 (0.25 GCUPs). For the other clouds, the small GCUPs value reflects the overhead with data transfer from EC2 to each master. The GCUP obtained when comparing the longest sequence with the UniProtKB/Swiss-Prot database using our architecture was 1.35, as presented in table 7.3.

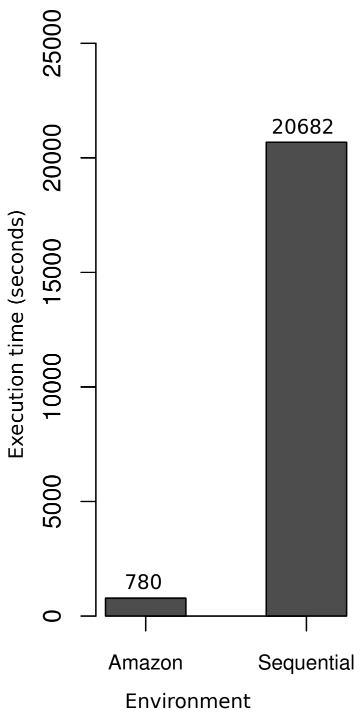


Figure 7.8: Sequential execution time for the longest sequence (Q9UKN1) with SSEARCH (November 2011) compared with the standalone execution time in Amazon EC2

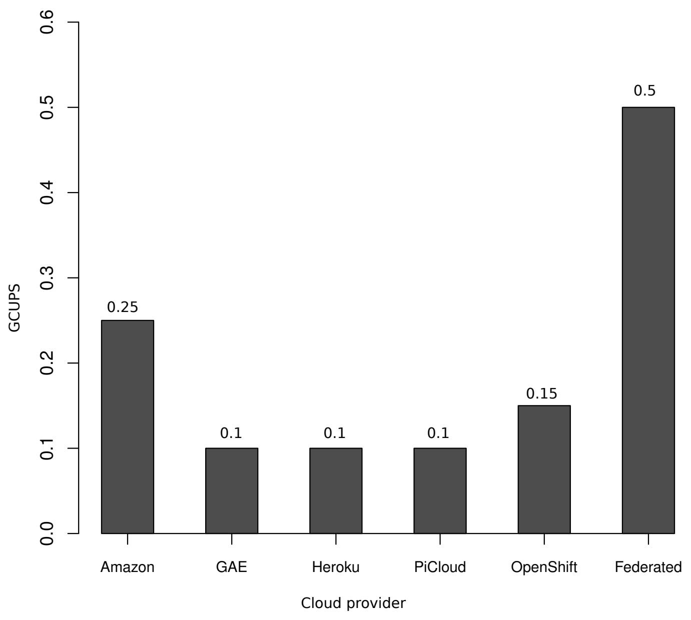


Figure 7.9: GCUPS of 24 query sequences comparison with the database UniProtKB/Swiss-Prot (November 2011) using our SW implementation

7.5 Related Work

The efforts that have been made to reduce the execution time of the SW application include (i) classical parallel platforms, such as clusters; (ii) accelerators such as Cell/BE and GPUs; and, more recently, (iii) clouds. Table 7.3 lists eight proposals of SW implementations and compares them to our proposal. We must note that there are many other implementations of SW in clusters, Cell/BEs, and GPUs. Here, we chose two implementations in each platform to give a picture of the performance improvements that have been achieved. As far as we know, there are only two implementations of SW in Hadoop for cloud environments [127, 176].

In column 2, the computing platform is provided, which can be Cell/BEs, cluster of Cell/BE, multi-core clusters, GPUs, and cloud.

The type of comparison is provided in column 3, which can be between two sequences (*seq* x *seq*) or *query* x *dbase*. Genomics databases (*dbase*) are composed of a great number of sequences and each processing element compares the same sequence (*query*) with a subset of sequences from the database. Therefore, a high amount of comparisons is made in the (*query* x *dbase*) case, but the size of the similarity matrices is not big. The *seq* x *seq* comparison, on the other hand, compares only two sequences and the size of the similarity matrix is usually big.

Column 4 provides the grain of computation. All *seq x seq* comparisons are fine grain comparisons, i.e., all the processing elements participate in the computation of a single similarity matrix. Coarse grain computations are normally made in *query x dbase* comparisons, where each processing element independently computes a set of similarity matrices.

As output, the SW algorithm can provide the score or the score and the alignment (Column 5). In the first case, only the first phase of SW is executed.

The number of processing elements used in the comparisons is listed in column 6. This number varies, and it depends on the platform used. Note that, even though only one GPU was used in the GPU SW implementations, these GPUs have multiple cores. For instance, the NVidia GTX 560Ti has 384 cores. In the first cloud implementation (line 7 of table 7.3), a cluster of 768 cores was used and in the second one (line 8 of table 7.3), 20 units of Amazon EC2 were used.

Column 7 lists the maximum speedups reported in the papers, which cannot be used for direct comparison since each approach uses a different general-purpose processor as a baseline to calculate speedups. Nevertheless, the reported speedups can give us an idea of the gains that can be obtained by each approach.

To measure the performance of SW, the metric GCUPs is often used. This metric calculates the rate at which the cells of the similarity matrix are updated. When a query sequence is compared to a genomics database, the sizes of the query sequence and the size of the whole database are taken into consideration. GCUPs range from 0.42 to 8.00 for the Cell/BE and from 0.25 to 4.38 for clusters. The best GCUPs were obtained with GPUs (from 23.3 to 58.8 GCUPs). Neither speedups nor GCUPs were provided for the cloud computing implementations.

In the last row, we present the details of our multi-cloud approach. Like most of the approaches, we calculate the SW score between a query sequence and a genomics database with coarse-grained computations. Unlike the other approaches, we propose and use a federated cloud architecture to execute our comparisons. Also, we used five public clouds and, only within the free quota, we could launch 37 applications. The GCUP obtained with our proposal (1.35 GCUPs) is comparable with the multi-core clusters and Cell/BEs approaches. As far as we know, ours was the first attempt to execute the SW algorithm in a vertical cloud federation.

7.6 Summary

In this chapter we proposed and evaluated a hierarchical multi-cloud architecture to execute the Smith-Waterman (SW) algorithm in a cloud federation. By only using the free-quota of five public clouds, we could execute the SW to compare 24 protein sequences with the UniProtKB/Swiss-Prot database in approximately one hour and this result is comparable to the ones obtained in SW cluster and Cell/BE executions. We also presented results where the multi-cloud approach was 22.55% faster that the best standalone cloud execution (EC2), showing the advantages of the cloud federation. In addition, we compared the EC2 execution with the sequential SSEARCH execution (November 2011). In this case, the EC2 execution achieved a speedup of 26.51x over SSEARCH.

Even though our approach could execute a huge application in multiple public clouds, there are some issues with it, which are: (i) the usage of a centralized coordinator to distribute the tasks, and (ii) the lack of fault-tolerance strategies for the coordinator and for the masters. The first issue may limit the scalability of the architecture and its usage in a dynamic environment. On the one hand, failure of the coordinator may require the re-execution of the whole application as the architecture does not provide a way to discover the tasks distributed to each master. In this case, the masters will continue the execution of their tasks, but the result will be inaccessible for the users. On the other hand, masters' failures will cause the re-execution of the tasks assigned to them. In both cases, the slaves continue the execution. These two issues are tackled in chapter 10, where we propose a decentralized and fault-tolerant architecture for cloud federations.

Table 7.3: Comparative view of the approaches that implement SW in HPC platforms

Paper	Platform	Comparison	Grain	Output	# Proc. Elements	Best Speedup	GCUPs
[385]	Cell/BE	queryxdbase	coarse	score	6 SPEs	—	8.00
[7]	cluster Cell/BE	queryxdbase	fine,coarse	score,align.	84 SPEs	55x	0.42
[290]	cluster	seqxseq	fine	score,align.	60 procs	39x	0.25
[261]	cluster	queryxdbase	coarse	score	24 cores	14x	4.38
[229]	GPU	queryxdbase	coarse	score	GTX295	—	29.7
[97]	GPU	seqxseq	fine	score,align.	GTX560	—	58.21
[127]	cloud	queryxdbase	coarse	score	768 cores	—	—
[176]	cloud	queryxdbase	coarse	score	20 EC2 Units	—	—
This work [214]	federated cloud	queryxdbase	coarse	score	5 clouds (37 apps)	26x (1 cloud)	1.35

Chapter 8

Excalibur: A User-Centered Cloud Architecture for Executing Parallel Applications

Contents

The previous chapter showed us that developing and deploying cloud applications is a difficult task, even for experienced software developers, due to the various constraints and the complex configuration tasks. Thus, we decided to investigate the usage of a cloud environment to execute cloud-unaware applications, considering the view of inexperience cloud users, without having to change the applications to meet clouds' environment constraints.

Therefore, in this chapter, we propose and evaluate a cloud architecture, called Excalibur, to execute applications on IaaS clouds. Our architecture aims to hide the configuration tasks from the users and to implements an auto-scaling strategy. Experimental results show that the proposed architecture could dynamically scale the application up to 11 virtual machines, reducing both the execution time and the cost of executing a genomics workflow when deployed on an instance type selected by the users.

The remainder of this chapter is organized as follows. In section 8.1, we present an introduction and the motivation behind this work. Section 8.2 presents the design of our cloud architecture. Experimental results are presented and discussed in section 8.3. Section 8.4 presents and compares some related works. Finally, section 8.5 concludes this chapter.

8.1 Introduction and Motivation

As stated in previous chapter, nowadays, the cloud infrastructure may be used to execute HPC applications, due to its characteristics such as resources on-demand, pay-as-you-go model, and full access to the underlying infrastructure [1]. However, executing high performance computing applications in the cloud still faces some difficulties such as the differences in HPC cloud infrastructures and the applications were not written for cloud.

Hence, cloud infrastructures require a new level of robustness and flexibility from the applications, as hardware failures and performance variations become part of its normal operation. In addition, cloud resources are optimized to reduce the cost for cloud providers often without providing performance guarantees at low cost for the users. Furthermore, cloud providers offer different resources and services that have costs and performance defined according to their purpose usage. In this scenario, the users face many problems. First, re-engineering applications to fit the cloud model requires expertise in both domains: cloud and high performance computing, as well as a considerable time to accomplish it. Second, selecting the resources that meet the applications' needs demands data about both applications and resources. Thus, deploying and executing an application in the cloud is still a complex task [179, 392].

Although some efforts have been made to help in solving these problems, most of them target software developers [262, 315], and they are not straightforward for inexperienced users [179].

Therefore, in this chapter, we propose and evaluate a cloud architecture, called Excalibur, to execute applications in the cloud with three main objectives: (a) provide a platform for high performance computing applications in the cloud for users without cloud skills; (b) dynamically scale the applications without user intervention; and (c) meet the users requirements such as performance at reduced cost. This work was published in [217].

The remainder of this chapter is organized as follows. Section 8.2 presents our cloud architecture. In section 8.3, experimental results are discussed. Section 8.4 presents related work and discusses cloud architectures to perform high performance computing applications. Finally, section 8.5 presents final considerations and future work.

8.2 Architecture Overview

The proposed architecture aims to simplify the use of the cloud and to run applications on it without requiring re-design of the applications. To achieve these goals, we consider that a cloud can be used to increase the capacity of local resources or combined with other clouds, i.e., hybrid cloud environment (Section 3.2.2).

In this chapter, an application represents a user's demand/work, being seen as a single unit by the user. An application is composed of one or more tasks which represent the smallest work unit to be executed by the system. The tasks that form an application can be connected by precedence relations, forming a workflow. The workflow is defined to be a set of activities, and these activities can be tasks, as said above, or even other workflows. Moreover, a partition is a set of independent tasks with respect to the precedence relation. The terms application and job are used interchangeably in this chapter.

We propose an architecture composed of micro-services. A micro-service is a lightweight and independent service that performs single functions and collaborates with other services using a well-defined interface to achieve some objectives. Micro-services make our architecture flexible and scalable since services can be changed dynamically according to the users' objectives. In other words, if a service does not achieve a desirable performance in a given cloud, it can be deployed in another one without requiring service restart.

Our architecture, called Excalibur, has three layers: *resource management*, *application*, and *user* layer as depicted in figure 8.1. The *resource management* layer comprises the services responsible for managing the resources (e.g., VM and/or storage). A resource can be registered by the providers or by the users through the *service registry*. By default, the resources provided by the public clouds are registered with the following data: *resource type* (e.g., physical or virtual machine, storage), *URL*, *costs*, and *purpose* usage (e.g., if the resource is optimized for CPU, memory or I/O-bound applications). The *resource management* service is responsible to validate these data and to keep them up-to-date. First, it gets a list of the resources through the *service registry*. Then, it asks the clouds the state of each resource and updates its state in the system. This is necessary because a resource registered at time t_i may not be available at time t_j, $t_j > t_i$ for many reasons. The *monitoring* and *deployment* services, on the other hand, are responsible to deploy and to monitor the applications. Monitoring is an important activity in our architecture for three reasons. First, it collects data about the resources. Second, it can be used to detect failures — sometimes the providers terminate services when they are stressing the CPU, RAM memory, or both. And finally, it supports our auto-scaling mechanism.

We provide a uniform view of the clouds through a *communication API*. This is necessary because each cloud provider often has different interfaces to access its resources.

On top of the *resource management* layer, the *application* layer provides services to schedule the jobs (*provisioning*), to control the data flows (*workflow data event*), to provide data streaming service (*data streaming processing*), and to execute the applications. The architecture uses MapReduce (Section 3.1.3) to distribute the applications. This does not mean that only MapReduce applications are supported, as in chapter 7, but only that the applications are distributed following the MapReduce model.

The *coordination* service manages the resources, which can be distributed across different cloud providers, and provides a uniform view of the system such as the available resources and the system's workload.

The *provisioning* service creates a workflow with the activities to set up the environment, and it creates an execution plan for the applications. In practice, the *provisioning* service communicates with the *coordination* service to obtain data about the resources and to allocate them for the applications. After that, it submits the workflow to the *workflow management* service.

An execution plan comprises the application to execute, its input data (i.e., data sources), the resources to execute it, a state (*initializing*, *waiting data*, *ready*, *executing*, and *finished*), and a characteristic that can be *known* or *unknown* by the system. A characteristic represents the application's behavior such as CPU, memory, or I/O-bound.

The *workflow management* service coordinates the execution of the workflow and creates the data flows (i.e., data streams) in the *workflow data event* service.

The *workflow data event* service is responsible to provide the data for the execution plans. A data flow has a source and a sink and it can supply data for multiple execution plans simultaneously. This avoids multiple accesses for the distributed file system (DFS) to fetch the same data.

Finally, the *user* layer has two services: *job submission* and *job stats processing*. The users submit their jobs through the *job submission* service. A job has the following data: the tasks which compose it, the constraints, the data definition (input and output), and the data about the cloud providers (e.g., name and access key). The users can monitor or obtain the results of their jobs through the *job stats processing*.

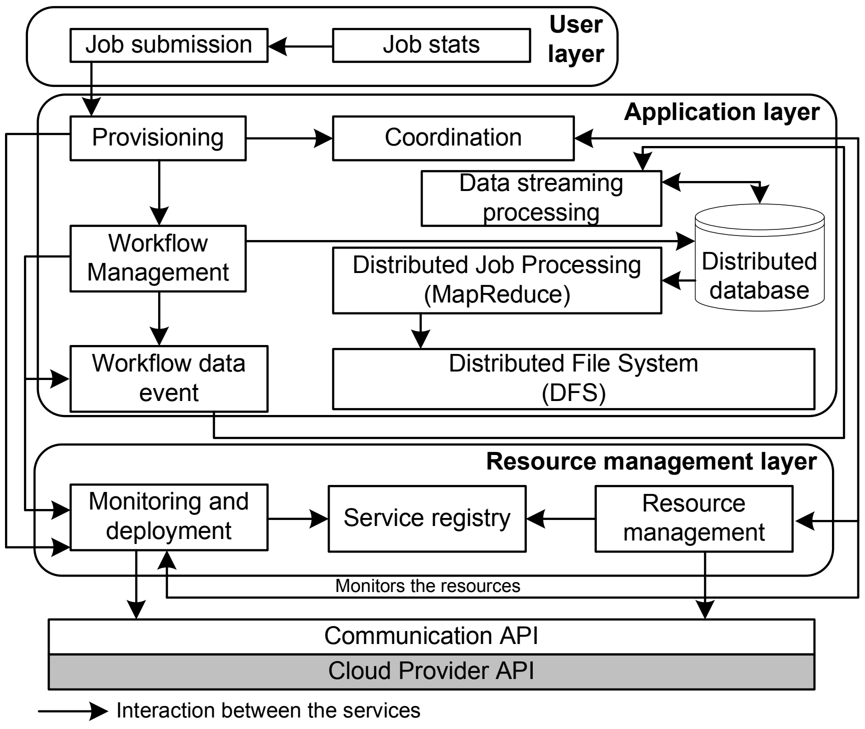


Figure 8.1: Excalibur: services and layers

Scaling cloud-unaware applications without having technical skills requires an architecture that abstracts the whole environment, taking into account the users' objectives. In the next subsections, we explain how the proposed architecture achieves these objectives.

8.2.1 Scaling Cloud-Unaware Applications with Budget Restrictions and Resource Constraints

The applications considered in this chapter are workflows but some parts of the workflow can be composed of a set of independent tasks that can be run in parallel. These independent tasks are the target of our scaling technique. For example, figure 8.2 shows a directed acyclic graph (DAG) with four tasks (T_1, T_2, T_3, and T_4) and the temporal precedence relations among the tasks. In this example, tasks T_2 and T_3 can be executed in parallel, after task T_1 finishing, as expected by any workflow engine. However, task T_3 is composed of three independent subtasks (s_1, s_2 and, s_3).

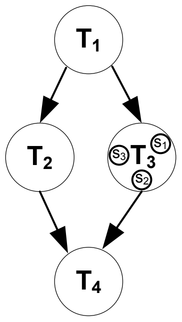


Figure 8.2: A DAG representing a workflow application with 4 tasks, where one of them, T_3, is composed of three independent subtasks

In this case, the subtasks can be grouped in p partitions and assigned to different nodes. One important problem here is to determine the size and the number of partitions. Over-partitioning can lead to a great number of short duration tasks that may cause a considerable overhead to the system. Hence, over-partitioning can result in inefficient resource usage. To avoid this, the size of one partition (P_p) is estimated as [15]:

$$P_p = \lfloor \frac{N_q * R}{T} \rfloor \tag{8.1}$$

where N_q is the workload size; T is the estimated CPU time for executing N_q in the partition; and R is a parameter for the maximum execution time for partition P_p. A partition size can be adjusted according to the node characteristics. For instance, if the resource usage by a partition P_p is below a threshold, P_p can be increased.

Partitions exist due to the concept of *splittable* and *static* files. It is the user who defines which data are splittable and how to split them when the system does not know. Splittable data are converted to JSON records and persisted onto the distributed database, so a partition represents a set of JSON records. Static data, on the other hand, are kept in the local file system.

Listing 8.1 illustrates the input file of task T_3. This file represents a FASTA file, and each line starting with the > character represents a genomics sequence, thus a subtask. The FASTA file is converted to the JSON format as shown in listing 8.1. A genomics sequence has a name (e.g., ERR135910.3), a description (e.g., 2405:1:1101:1234:1973:Y/1), and a value (e.g., NAAGGGTTTGAGTAAGAGCATAGCTGTTGGGACCCGAAAGATGGT-GAACT). In this case, the system will create a table with name *sequences* in the distributed database and will store onto it the genomics sequences. Moreover, three partitions can be created to execute in parallel.

```
>ERR135910.3 2405:1:1101:1234:1973:Y/1
NAAGGGTTTGAGTAAGAGCATAGCTGTTGGGACCCGAAAGATGGTGAACT

>ERR135910.5 2405:1:1101:1170:1994:Y/1
NTCAACGAGGAATTCCTAGTAAGCGNAAGTCATCANCTTGCGTTGAATAC

>ERR135910.6 2405:1:1101:1272:1972:Y/1
NTAGTACTATGGTTGGAGACAACATGGGAATCCGGGGTGCTGTAGGCTTG
```

Listing 8.1: Example of a splittable file format (i.e., FASTA file)

```
{
 "sequences" : [ {
   "name" : "ERR135910.3",
   "description" : "2405:1:1101:1234:1973:Y/1",
   "value" : "NAAGGGTTTGAGTAAGAGCATAGCTGTTGGGACCCGAAAGATGGTGAACT"
 }, {
   "name" : "ERR135910.5",
   "description" : "2405:1:1101:1170:1994:Y/1",
   "value" : "NTCAACGAGGAATTCCTAGTAAGCGNAAGTCATCANCTTGCGTTGAATAC"
 }, {
   "name" : "ERR135910.6",
   "description" : "2405:1:1101:1272:1972:Y/1",
   "value" : "NTAGTACTATGGTTGGAGACAACATGGGAATCCGGGGTGCTGTAGGCTTG"
 } ]
}
```

Listing 8.2: Example of a JSON with three genomics sequences

8.2.2 Reducing Data Movement to Reduce Cost and Execution Time

Data movement can increase the total execution time of an application (i.e., makespan) and sometimes it can be higher than the computation time due to the differences in networks' bandwidth. In this case, we can invert the direction of the logical flow, moving the application to as close as possible of the data location. Actually, we distribute the data using a DFS and the MapReduce strategy.

Although MapReduce is an elegant solution, it has the overhead of creating the *map* and the *reduce* tasks every time a query must be executed. We minimize this overhead by using a data structure to keep the data in memory. This increases the memory usage and requires data consistency policies to keep the data updated. However it does not increase financial costs. We implemented a data policy that works as follows. Each record read by a node is kept in memory and its key is sent for the *coordination* service (Figure 8.1). The *coordination* service stores the key/value pairs and the information where they were read. When a node updates a record, it removes it from its memory and notifies the coordinator. Then, the *coordination* service asynchronously notifies all nodes that have the key to remove it from its memory. This is not a strong consistency policy, since we assume that a node partition does not affect system's consistency, and the tasks are idempotent. For instance, consider a scenario where a key k was loaded by three nodes (N_1, N_2, N_3) and a partition occurred after the readings. The task represented by the key k will be executed by the nodes, but its result will be the same.

8.2.3 Reducing Job Makespan with Workload Adjustment

In an environment with incomplete information and unpredictable usage patterns as the cloud, load imbalance can have high impact in the total execution time and in the monetary cost. For instance, assigning a CPU-bound task to a memory optimized node is not a good choice. To tackle this problem, we propose a workload adjustment technique that works as follows. For execution plans in the *ready* state and with an *unknown* application's characteristics, the scheduler selects similar execution plans and submits them for each available resource (i.e., CPU, memory or I/O optimized) and waits their execution to finish. When the first execution finishes, the scheduler checks if there are similar execution plans in the *ready* state and submits them. Execution plans are similar if they execute the same application and almost the same number of tasks.

When there are no more execution plans in the *ready* state, the scheduler assigns one that is *executing*. Note that, in this case, the cost can increase, since we have more than one node executing the same task. In fact, we minimize this, finishing the slowest node according to the difference between the elapsed time and the time to charge for its usage.

8.2.4 Making the Cloud Transparent for the Users

As our architecture aims to make the cloud transparent for the users, it automates the whole setup process. However, for some users, this is not sufficient since some jobs still require programming skills. For instance, consider the following scenarios: (i) a biologist who wants to search DNA units that have some properties in a genomics database, and to compare these DNA units with another sequence that he/she has built; (ii) a social media analyst who wants to filter tweets using some keywords.

Normally, these works require a program to read, to parse, and to filter the data. However, in our solution, the users only have to know the structure or their data and to use a domain specific language (DSL) to perform their work. Listings 8.3 and 8.4 show how these works can be defined, where *b*, *P1*, *P2*, *T*, and *w* are users' parameters.

```
execute T with (select reads from genomic-database where P1 = X and P2 =
    Y) -seq = b
```

Listing 8.3: Defining a genomics analysis application

```
select tweet from tweets where text contains (w)
```

Listing 8.4: Defining a Twitter analysis application

In these cases, a data structure (i.e., a file) is seen as a *table* whose fields can be filtered. Although there are similar approaches in the literature such as BioPig [262] and

SeqPig [315], they still require programming skills to register the drivers and to load/store the data. In other words, to use them, the users have to know the internals of the system.

In order to illustrate our architecture, consider the bioinformatics scenario described above. In this case, the biologist submits a YAML Ain't Markup Language (YAML) file depicted in listing 8.5, with the application, the requirements, and the input data (e.g., the genomics database and the built sequence) using a console application (a client of the *job submission* service) at the *user* layer (Figure 8.3 (1)). The *job submission* sends the job description to the *provisioning* service at the *application* layer (Figure 8.3 (2)). When the *provisioning* service receives the application, it requests the *coordination* service resources the resources that match the users' requirements (Figure 8.3 (3)). Then, the *coordination* service asks the *resource management* service such resources, and returns them for the *provisioning* service (Figure 8.3 (4)). Next, the *provisioning* service creates a workflow (i.e., deployment workflow) with the activities to configure the environment and an execution plan for the application. Next, it submits them to the *workflow management* (Figure 8.3 (5)). The deployment workflow's activities are: (i) selects the cheapest virtual machine to setup the environment; (ii) gets non splittable files (e.g., a reference genome) to store them in the local file system; (iii) gets the splittable files (e.g., the genomics database) and persists them into the DFS; (iv) creates a VMI of the configured environment; and (v) finishes the VM used to configure the environment. After, the *workflow management* service executes the deployment workflow. In other words, it stores the splittable files (Figure 8.3 (6)) into the distributed database, deploys the applications (Figure 8.3 (7)), and asks the *workflow data event* service (Figure 8.3 (8)) to create the data streams. Then, the *workflow data event* registers the data streams in the *data stream processing* service (Figure 8.3 (9)). Finally, the *workflow management* service executes the application (Figure 8.3 (10)).

In this scenario, a partition has a set of genomics sequences read from the DFS by the *workflow data event* and it is assigned to an execution plan. During the execution, the *provisioning* service monitors the applications through the *monitoring* service and if the execution time of a partition reaches the expected time it creates more VMs to redistribute the workload. After all tasks have finished, the user receives the output through the *job submission* service.

```yaml
---
id: "cmsearch"

requirements:
  - memory: "4"
  - architecture: "I386_64"
  - platform: "LINUX"

applications:
  application:
  - command: "cmsearch -o ${hits} ${rfam} ${query_sequence}"

data-def:
  data:
  - id: "hits"
    is-output: true
    is-splittable: true
    path: "$HOME/hits.txt"

  - id: "rfam"
    is-output: false
    is-splittable: false
    path: "$HOME/Rfam/11.1/Rfam.cm"

  - id: "query_sequence"
    is-output: false
    is-splittable: true
    path: "$HOME/query_sequence.fa"
```

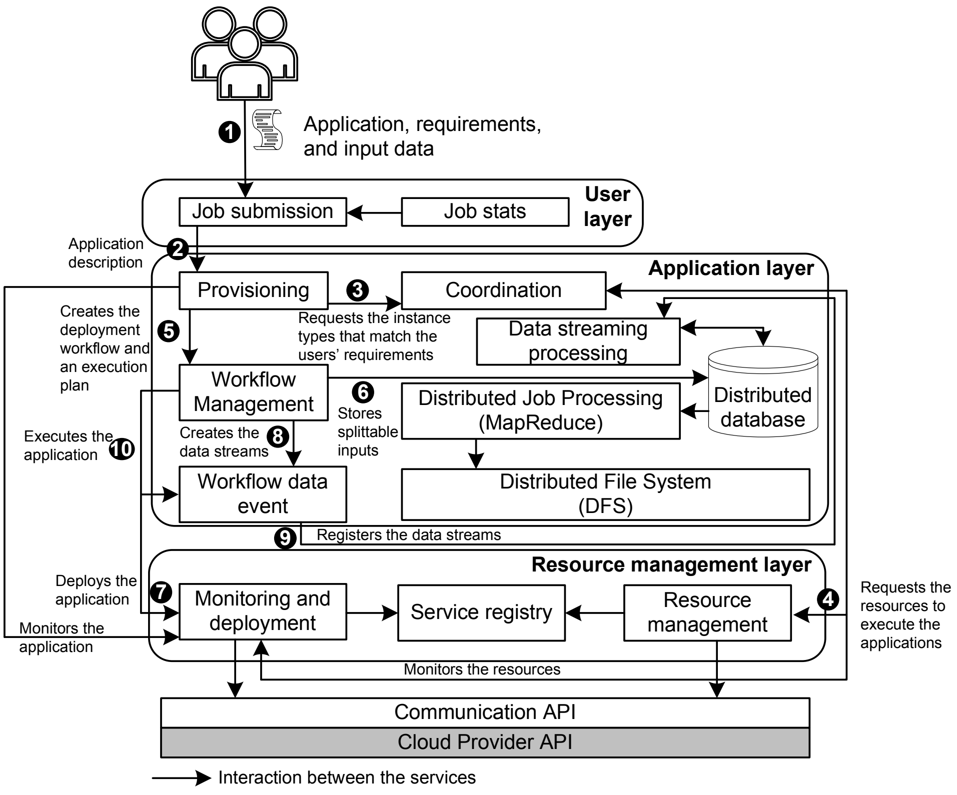
Listing 8.5: Excalibur: example of a YAML file with the requirements and one application to be executed on the cloud

8.3 Experimental Results

We deployed an instance of our architecture on Amazon EC2. Our goal was to evaluate the architecture when instanced by a user without cloud computing skills.

We executed a genomics workflow that aims to identify non-coding RNA (ncRNA) in the fungi *Schizosaccharomyces pombe* (*S. pombe*). This workflow, called Infernal-Segemehl, consists of four phases (Figure 8.4): (i) first, the application Infernal [260] maps the *S. pombe* sequences onto a nucleic acid sequence database (e.g., Rfam [143]); (ii) then, the sequences with no hit or with a low score are processed by segemehl [160]; (iii) next, SAMTools [220] is used to sort the alignments and to convert them to the SAM/BAM format. Finally, (iv) the RNAFold [159] application computes the minimum free energy of the RNA molecules obtained in step (iii).

We used the Rfam version 11.1 (with 2278 ncRNA families) and *S. pombe* sequences extracted from the EMBL-EBI (1 million reads). Rfam is a database of non-coding RNA families with a seed alignment for each family and a covariance model profile built on this seed to identify additional members of a family [143].



1
Application, requirements, and input data
User layer
Job submission
Job stats
2
Application description
Application layer
Provisioning
3
Coordination
Requests the instance types that match the users' requirements
Data streaming processing
Creates the deployment workflow and an execution plan
5
Workflow Management
6
Stores splittable inputs
Distributed Job Processing (MapReduce)
Distributed database
Creates the data streams
8
Executes the application
10
Workflow data event
Distributed File System (DFS)
9
Registers the data streams
Resource management layer
Deploys the application
7
Monitoring and deployment
Service registry
Resource management
4
Requests the resources to execute the applications
Monitors the application
Monitors the resources
Communication API
Cloud Provider API
Interaction between the services

Figure 8.3: Executing an application using the Excalibur cloud architecture

Although, in its higher level, this workflow executes only four applications, it is data oriented. In other words, each step processes a huge amount of data and, in all tools, each pairwise sequence comparison is independent. So, the data can be split and processed in parallel.

Listing 8.6 shows an input file with two applications and their input/output files. In this execution, the users requested at least 80GB of RAM memory and the Linux operating system to execute two tasks: *cmsearch* (Infernal, figure 8.4) and *segemehl.x* (segemehl [160]). The *cmsearch* task receives as input a nucleic acid sequence database (*$HOME-/Rfam/11.1/Rfam.cm* in listing 8.6) and one file with the genomics sequences (*$HOME-/Spombe/reads_spombe2.fa*), and its outputs should be stored in *$HOME/infernal/infernal_hits.txt* and in *$HOME/infernal/infernal_hits_table.txt*. The *segemehl.x* task, on the other hand, receives as input one index (*$HOME/genome/chrs_mm10.idx*), a reference genome (*$HOME/genome/chrs_mm10.fa*), and one database sequence (*segemehl_database*). The *segemehl_database* comprises the *S. pombe* sequences without hit or which a score lower than 34. These sequences should be obtained from both the Infernal's output (*$HOME/infernal/infernal_hits.txt*) and the (*S. pombe* reads (*spombe_reads*, through the expression (Line 51) given by the users.

In this context, each splittable data (ids: *infernal_hits, infernal_hits_table*, and *spombe_reads* in listing 8.6) represents a table that can be filtered based on their structure. For example, figure 8.5 shows the top five lines of the *infernal_hits_table* output file. This file consists of 18 fields [259], each one with a name (e.g., target name, accession, and score), that are used to filter the data. In other words, at runtime, the system creates a table named *infernal_hits_table* with these fields, enabling filter operations.

```
---
id: "infernal-segemehl"

requirements:
  - memory: "80"
  - architecture: "i86_64"
  - platform: "Linux"

applications:
  application:
  - command: "cmsearch -o ${infernal_hits} --tblout ${infernal_hits_table} ${rfam} ${
       spombe_reads}"
    order: 1
  - command: "segemehl.x -i ${idx} -d ${genome} -q ${segemehl_database} > ${output}"
    order: 2

data-def:
  data:
  - id: "infernal_hits"
    is-output: true
    is-splittable: true
    path: "$HOME/infernal/infernal_hits.txt"

  - id: "infernal_hits_table"
    is-output: true
    is-splittable: true
    path: "$HOME/infernal/infernal_hits_table.txt"

  - id: "rfam"
    is-output: false
    is-splittable: false
    path: "$HOME/Rfam/11.1/Rfam.cm"

  - id: "spombe_reads"
    is-output: false
    is-splittable: true
    path: "$HOME/Spombe/reads_spombe2.fa"

  - id: "idx"
    is-output: false
    is-splittable: false
    path: "$HOME/genome/chrs_mm10.idx"

  - id: "genome"
    is-output: false
    is-splittable: false
    path: "$HOME/genome/chrs_mm10.fa"

  - id: "segemehl_database"
    is-output: false
    is-splittable: false
    query: " select sequence from spombe_reads where sequence not in (select sequence from
         infernal_hits where score >= 34)"
```

Listing 8.6: Users' description of the Infernal-Segemehl workflow

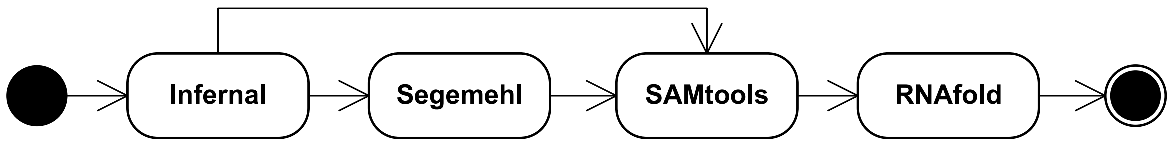


Figure 8.4: The Infernal-Segemehl workflow

```
#target name     accession query name accession mdl mdl from   mdl to seq from   seq to strand trunc pass   gc  bias  score  E-value inc   description of target
#--------------- --------- ---------- --------- --- -------- -------- -------- -------- ------ ----- ---- ---- ----- ------ --------- ---   ---------------------
ERR135910.3845   -         5S_rRNA    RF00001    cm        9       59        1       50      + 5'&3'    4 0.54   0.2   39.2   2.7e-07 !     2405:1:1101:20720:2186:Y/1
ERR135910.4258   -         5S_rRNA    RF00001    cm       12       62        1       50      + 5'&3'    4 0.52   0.1   38.5   4.2e-07 !     2405:1:1101:2670:2486:Y/1
ERR135910.8417   -         5S_rRNA    RF00001    cm       12       62        1       50      + 5'&3'    4 0.52   0.1   38.5   4.2e-07 !     2405:1:1101:6991:2660:Y/1
```

Figure 8.5: Infernal's target hits table

The Amazon EC2 micro instance (t1.micro) was used to setup the environment (e.g., install the applications, to copy the static files to the local file system), and to create a virtual machine image (VMI).

In addition to the cloud's executions, we also executed the workflow in a local PC (Table 8.1) to have an idea of the cloud overhead.

Table 8.1: Resources used to execute the Infernal-Segemehl workflow

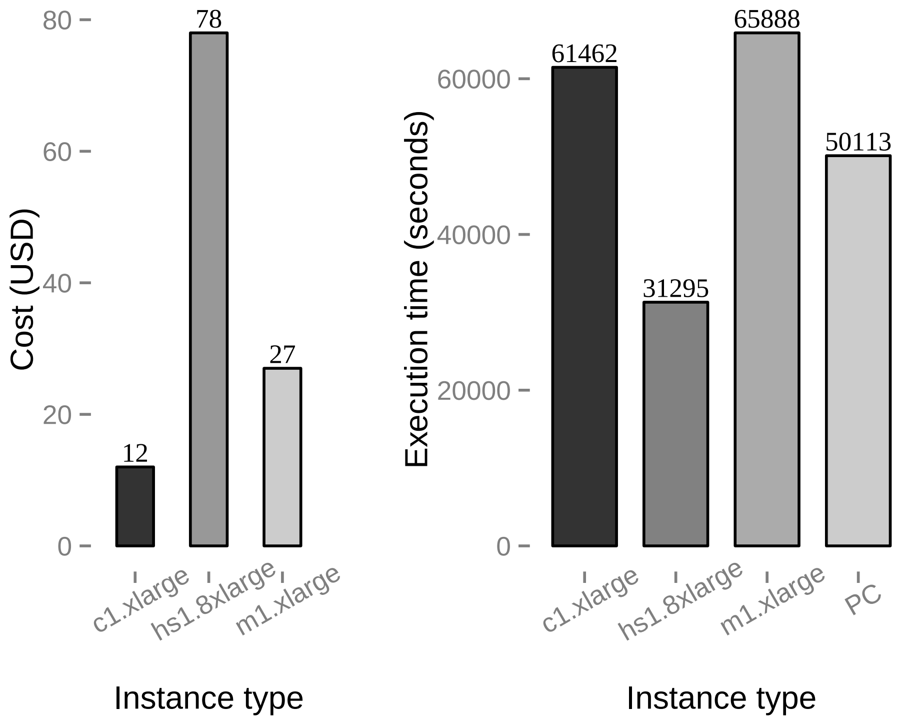
Instance type	**CPU**	**RAM**	**Cost ($/hour)**
PC	Intel Core 2 Quad CPU 2.40 GHz	4 GB	Not applicable
hs1.8xlarge	Intel Xeon 2.0 GHz 16 cores	171 GB	4.60
m1.xlarge	Intel Xeon 2.0 GHz 4 cores	15 GB	0.48
c1.xlarge	Intel Xeon 2.0 GHz 8 cores	7 GB	0.58
t1.micro	Intel Xeon 2.0 GHz 1 core	613 MB	0.02

8.3.1 Scenario 1: execution without auto-scaling and based on users' preferences

This experiment aims to simulate the users' preferences. In this case, the users are responsible for selecting an instance type based on their knowledge about the applications or on the amount of computational resources offered by the instance types. We executed the workflow illustrated in figure 8.4 in the first four instances listed in table 8.1.

Figure 8.6 shows the costs and the execution time for the four instances. The time was measured from the moment the application was submitted until the time all the results were produced (i.e., wallclock time). Therefore, it includes the cloud overhead (e.g., data movement to/from the cloud, VM creation, among others). The instance *hs1.8xlarge*, which was selected based on the application requirements ($\geq$ 88GB of RAM), outperformed all other instances. Although it was possible for the users to execute the workflow without requiring cloud skills, they paid a high cost (USD 78.00). This happened because the users (i.e., biologists) specified that their applications would need more than 88GB of RAM and in fact, they used only 3GB of RAM. Moreover, the *hs1.8xlarge* is suitable for applications that demand high I/O performance operations.

Considering this scenario, the cloud is not an attractive alternative for the users due to its execution times. They were 22% and 31% higher than the local execution (PC table 8.1). Even in the best configuration (*hs1.8xlarge*), the execution time was only 60% lower with a high monetary cost. These differences are owing to the multi-tenant model employed by the clouds.



(a) Monetary cost (b) Execution time

Figure 8.6: Cost and execution time of the Infernal-Segemehl workflow (Figure 8.4) on the cloud, when allocating the resources based on users' preferences

8.3.2 Scenario 2: execution with auto-scaling

This experiment aims to evaluate if the architecture can scale a cloud-unaware application.

Based upon the previous experiment (Figure 8.6), the system discarded the I/O optimized instance (*hs1.8xlarge*) due to its high cost (Table 8.1) and also because the application did not really require the amount of memory defined by the user. In a normal scenario, this instance is selected only if the monitoring service confirms that the application is I/O intensive.

To scale the application, the system created p partitions using the equation (8.1) with R equals to one hour and with T equals to nine hours. These values represent the expected execution time for one partition and for the whole workflow. They were defined because Amazon charges the resource per hour and because, in the previous experiment, the best execution time took approximately 9 hours to finish (Figure 8.6). This means that this experiment aimed to at least decrease the cost. In this case, 9 partitions were created.

As in the beginning, the system had not sufficient data to decide if the workflow was memory or CPU-bound, it submitted two similar partitions — for two instance types (*m1.xlarge* and *c1.xlarge*) — to realize which one was the most appropriate to execute the applications.

Figure 8.7 shows the execution time for each partition in the selected instance types. As soon as the execution of the partition assigned to the *c1.xlarge* instance finished, the system created one VM for each partition in the *ready* state and executed them. Although there were only seven partitions in the ready state and one in execution (*execution* state), the architecture duplicated the partition in execution, since its execution time in the *m1.xlarge* instance was unknown. After one hour, three more instances were created to redistribute the tasks as shown in figure 8.8.

Due to the cloud infrastructure, which provided in nearly real-time the requested resources, and the auto-scaling mechanism, which selected the resources based on the partitions' characteristics, we decreased the cost (5 times) and the makespan (63%) using 10 *c1.xlarge* instances (80 vCPUs) and one *m1.xlarge* (4 vCPUs) compared to instance type specified by the users (*hs1.8xlarge* in table 8.1). The makespan was reduced from 31, 295 seconds (Figure 8.6(b)) to 10, 830 seconds (Figure 8.8).

Our strategy differs from scaling services (e.g., Amazon CloudWatch – aws.amazon.com/cloudwatch/) offered by the cloud providers, since the users do not have to select an instance type nor to set up the environment.

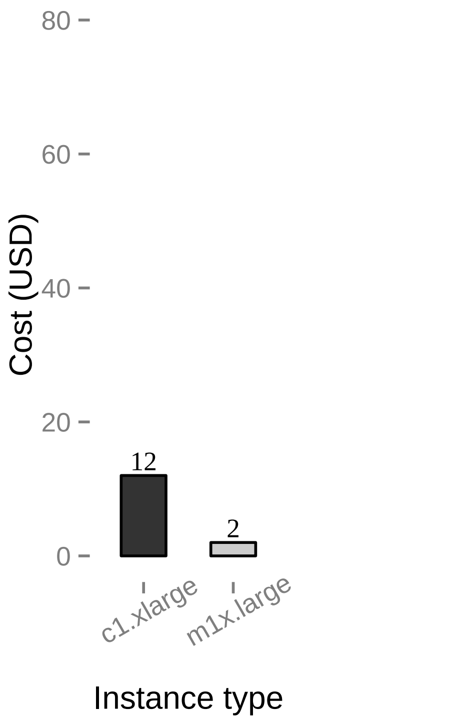
8.4 Related Work

In the last years, many works have described the challenges and opportunities of running high-performance computing in the cloud [1, 179, 392].

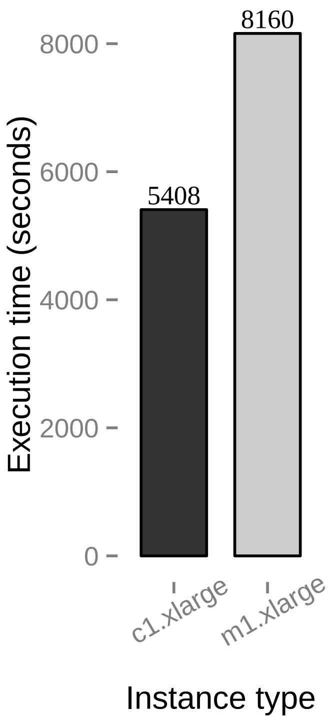
Therefore, many works have focused on developing new architectures to execute users' applications in the cloud considering both cost and performance. For instance, the Cloud Virtual Service (CloVR) [15] is a desktop application for automated sequences analysis using cloud computing resources. With CloVR, the users execute a VM on their computer, configure the applications, insert the data in a special directory, and CloVR deploys an instance of this VM on the cloud to scale and to execute the applications. CloVR scales the application by splitting the workload in p partitions through the equation (8.1), and executing the Cunningham BLAST runtime [371] to estimate the CPU time for each BLAST query.

Iordache and colleagues [166] developed Resilin, an architecture to scale MapReduce jobs in the cloud. The solution has different services to provision the resources, to handle jobs flow execution, to process the users requests, and to scale according to the load of the system. Doing bioinformatics data analysis with Hadoop requires knowledge about the Hadoop internal and considerable effort to implement the data flow.

In [262], a tool for bioinformatics data analysis called BioPig is presented. In this case, the users select and register a driver — bioformatics algorithms — provided by the tool and write their analysis' jobs using the Apache Pig (pig.apache.org) data flow language.



(a) Cost to execute the workflow using 10 *c1.xlarge* instances and 1 *m1.xlarge* instance



(b) Execution time for one partition when executed in the *c1.xlarge* and *m1.xlarge* instances. One partition was defined to finish in 1 hour with the deadline of 9 hours for the whole workflow

Figure 8.7: Monetary cost and execution time of the Infernal-Segemehl workflow (Figure 8.4) on the cloud with the auto-scaling enabled

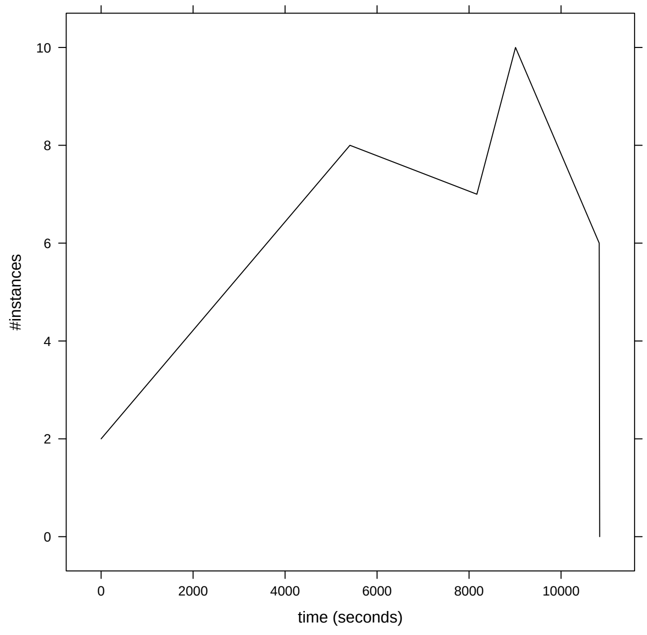


Figure 8.8: Scaling the Infernal-Segemehl workflow

SeqPig [315] is another tool that has the same objective of BioPig. The differences between them are the drivers provided by each tool. These tools reduce the needs to know Hadoop internal to realize bioinformatics data analysis.

Table 8.2 summarizes the approaches discussed in the previous paragraphs. As can be seen in this table, one architecture, CloVR[15], can both execute and scale workflow in the cloud. The others [166, 262, 315] can execute MapReduce applications. Hence, the auto-scaling is supported by the in-built approaches implemented by MapReduce's frameworks such as Apache Hadoop.

The closest works to ours are CloVR [15], BioPig [262], and SeqPig [315]. Our work differs from these approaches in the following ways. First, the users do not need to configure a VM in their computers to execute the applications in the cloud. Second, our architecture tries to match the workload to the appropriate instance type. Third, the data flow is defined using an abstract language freeing the users to write any code. The language is the same as used by BioPig and SeqPig but with the difference that the users write the data flow by only considering the data structure. For instance, to filter the sequences using BioPig or SeqPig the users have to register the loaders, the drivers, and to write a script to execute the analysis, which can be more appropriate for software developers.

Table 8.2: Comparative view of user-centered cloud architectures

Paper	Workflow	Auto-scaling	User support	Application
CloVR [15]	Yes	Yes	Script	BLAST
Resilin [166]	No	Yes	Hadoop	MapReduce applications
BioPig [262]	No	Yes	Pig	MapReduce applications
SeqPig [315]	No	Yes	Pig	MapReduce applications
Excalibur [217]	Yes	Yes	DSL + Pig	More general applications

8.5 Summary

In this chapter, we proposed and evaluated a cloud architecture, named Excalibur. Excalibur aims to execute cloud-unaware applications in the cloud without requiring programming or cloud skills from the users. Following an autonomic approach, Excalibur: (a) set up the whole cloud environment; (b) dynamically scaled the applications to reduce both the monetary cost and the execution time of a genomics workflow (Figure 8.4); and (c) enabled the users to describe the dependencies between the applications based on the structure of their data (i.e., input and output data).

We instantiated our architecture on Amazon EC2 considering two different scenarios: one with the auto-scaling enabled and another with it disabled. In the first case, the user was responsible for selecting an instance type to execute the applications, whereas in the second case, an instance type was selected based on historical data and on characteristics of the applications. Using 11 virtual machines, Excalibur reduced the execution time by 63% and the cost by 84%.

Therefore, in this chapter, we considered a single cloud, and the users were responsible for selecting an instance type to start the execution of the applications. Moreover, Excalibur assumed that the clouds were homogeneous with regard to the resources' constraints. In other words, that the constraints of the resources were equal between the clouds belonging to a cloud provider. Nonetheless, nowadays, the clouds are usually heterogeneous even when they belong to the same cloud provider. Hence, we need a method to handle clouds' heterogeneity properly, and this motivated the work that we will be described in the next chapter.

Chapter 9

Resource Selection Using Automated Feature-Based Configuration Management in Federated Clouds

Contents

In the previous chapter, we implemented an architecture and a domain specific language (DSL) that allowed the users to execute a set of ordinary applications on a cloud. The architecture focused on users without cloud computing skills, and implemented an auto-scaling strategy to trie to speed up the execution of the applications. Therefore, we considered a simplified environment, where the constraints associated with the resources were the same in all clouds belonging to a provider, and we only considered a single cloud.

For that reason, in this chapter, we are interested in using multiple clouds, taking into account different user profiles. In other words, we aim to help the users on selecting appropriate clouds' resources to execute their applications on heterogeneous and federated clouds. Hence, we present and evaluate an engineering method based on software product line (SPL) to achieve these goals. The SPL-based engineering method enables a declarative and goal-oriented strategy, allowing resource selection and deployment on multiple clouds.To the best of our knowledge, our engineering method is the first that (a) supports the description of clouds' services independent of cloud providers; (b) enables automatic resource selection and configuration on various clouds, considering temporal and functional dependencies between the resources, which leaves the environment in a consistent state; (c) offers a level of abstraction suitable for different user profiles (i.e., system administrators, software developers, and ordinary users).

The remainder of this chapter is organized as follows. In section 9.1, we present the introduction of this chapter. Section 9.2 presents the motivations behind this chapter, as well as the challenges we are addressing in this chapter. Sections 9.3 and 9.4 provide a briefly introduction for multi-objective optimization problem (MOOP) and feature model (FM). In section 9.5, we present the proposed cloud model followed by the feature models to handle commonalities and variabilities of the clouds. Experimental results are presented and discussed in section 9.7. Section 9.8 presents and compares related works. Finally, section 9.9 concludes this chapter.

9.1 Introduction

Resource selection in the cloud normally represents a difficult task, mostly because the clouds offer a wide range of resources [200, 392]. Moreover, such resources are usually suited for different purposes, and they may have multiple constraints [102, 173]. Learning how to deal with these options may require days or even weeks, without guarantees of meeting the users needs [179, 346]. In this scenario, the users mostly take decisions based on very few data, which can lead to under or over resource provisioning and can also increase the financial cost. For example, to execute an application in an IaaS cloud, the users have to: (a) select an instance type, a disk type, and an operating system; (b) install and configure all necessary applications; (c) transfer all data to the cloud; (d) create a virtual machine image (VMI) to avoid to install and to configure the applications in all nodes, if multiple nodes are needed; and finally, (e) execute their application. However, choosing an instance type demands data about the characteristics of the applications as well as data about the technical requirements and purpose usage of the instances (i.e., virtual machines). In this case, a mismatching between the application characteristics and the instance purpose usage commonly results in a high-financial cost [73, 167, 267], as we also observed in the previous chapter. In addition, some instance types require specific virtualization technique (Section 3.1.1.2) to work correctly. This demands technical knowledge about the virtualization techniques and about the virtual machine images (Section 3.4.1) available for usage. Nonetheless, the users normally lack such knowledge, which often lead to invalid configuration requests, increasing the time to configure a computing environment in the cloud. Hence, the users must have system administration and cloud computing skills.

In order to tackle these problems, we propose a product line engineering (PLE) process to help the users on dealing with configuration options and to enable a declarative strategy. In this case, the users specify their needs and the system automatically (i) selects the resources and (ii) sets up the whole environment accordingly. Product line engineering is a strategy to design a family of products, with variations in features, and with a common architecture [82]. A feature means a user requirement or a visible functionality of a product [182]. In other words, a product line engineering approach aims to develop a platform and to use mass customization to create a group of similar products that differ from each other in some specific characteristics. These different characteristics are called *variation points* and their possible values are known as *variants* [282].

Figure 9.1 shows how our engineering strategy works. First, the domain engineers create an abstract model to describe the commonalities and variabilities of the clouds. After, in the *domain design* phase, the system engineers define the configuration knowledge (CK), the software product line (SPL) architecture, and benchmark the clouds to obtain qualitative and quantitative attributes of the clouds' resources, such as performance and cost. In the next phase, *domain implementation*, the engineers refine the feature models with these data and publish it to be used by the users. In other words, the engineers create the concrete feature models that describes the configurations available in the clouds. Then, the users specify the requirements of the computing environment they want, as well as their objectives (such as to maximize performance and/or to minimize cost). Finally, the configurations that are optimal with respect to the users' objectives are automatically selected and instantiated through the clouds' APIs. In this scenario, a computing environment is always a product of a valid model and non-technical users can obtain optimal configurations.

Considering the scenario illustrated in the first paragraph of this section and following the product line engineering method, the users only need to describe their requirements, e.g., the number of CPU cores, the amount of memory, the storage size, and their objectives; and the system allocates and configures the whole environment based on the feature models.

The advantages of our software product line (SPL) engineering method are manifold. First, it enables auto-scaling strategies to reuse existing configurations. Second, it avoids invalid configurations or configurations that do not match some objectives (e.g., maximize performance at minimal cost) without requiring from users cloud computing or system administration skills. Third, it provides a uniform view of the clouds translating specific cloud terms to concepts independent of cloud providers. Fourth, it can support the users on provisioning their resources based on multiple parameters such as the location where the resources should be deployed; the software packages, or the cloud provider. Fifth, it

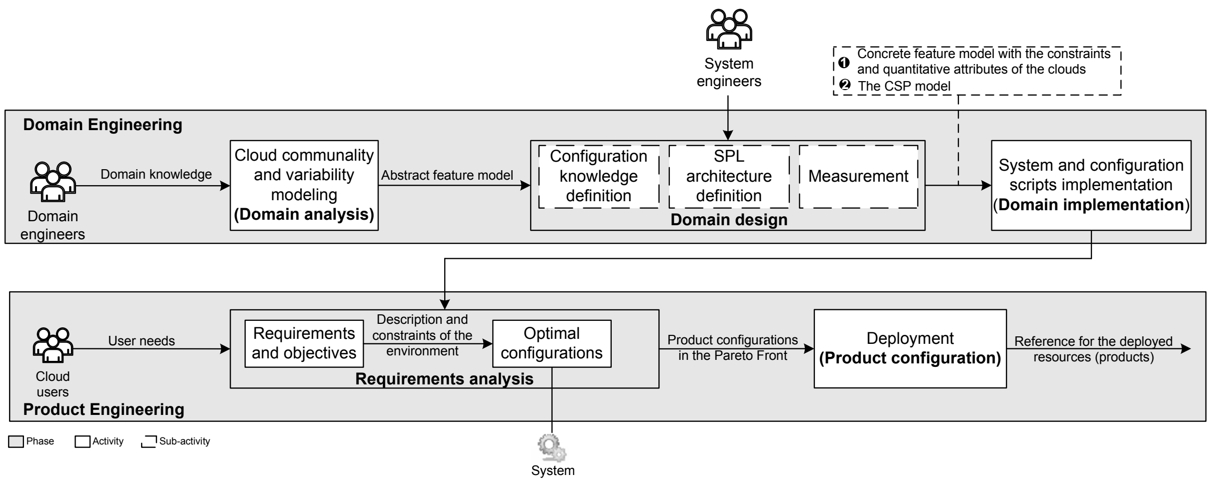

System engineers
❶ Concrete feature model with the constraints and quantitative attributes of the clouds
❷ The CSP model
Domain Engineering
Domain engineers
Domain knowledge
Cloud communality and variability modeling (Domain analysis)
Abstract feature model
Configuration knowledge definition
SPL architecture definition
Measurement
Domain design
System and configuration scripts implementation (Domain implementation)
Cloud users
User needs
Requirements and objectives
Description and constraints of the environment
Optimal configurations
Requirements analysis
Product configurations in the Pareto Front
Deployment (Product configuration)
Reference for the deployed resources (products)
Product Engineering
Phase
Activity
Sub-activity
System

Figure 9.1: An engineering method to handle clouds' variabilities

can be used to document the whole execution environment (i.e., hardware and software) without needing virtual appliances.

Similar strategies have been considered by many works in the domain of cloud computing. Some of these works [67, 288] have focused on handling the variabilities of the platform-as-a-service (PaaS) layer (Section 3.2.1). In this case, they aim to support the developers on deploying their application in the cloud or to support them on developing cloud-aware applications, i.e., in writing applications that use cloud's services. Other studies [102, 339, 340] have employed feature models to describe the variabilities of virtual machine images (i.e., virtual appliance (VA)) with different objectives. For instance, some works aim to reduce energy consumption [102, 339] of virtual machines taking into account performance constraints and the time to set up one VM; other works aim to reduce the amount of storage [389] used with pre-built virtual machine images. Cloud service selection [373] and configuration of multi-tenant applications (SaaS layer) have been addressed by some works such as [304, 305, 313] that aim to help the users on customizing SaaS applications. In [373], the users define multiple models of a service (e.g., storage) and submit them to a system that selects one that meets their objectives. Finally, feature model has been used to capture the variabilities of one specific IaaS provider [130]. Our approach differs from these works in the following ways: (i) it addresses the configuration options at the IaaS layer independent of cloud provider, and in a way that different user profiles could express their preferences, enabling model reuse; (ii) it considers multiple service selections (e.g., virtual machine, storage); (iii) it uses feature models to handle the variabilities of the execution environment and to enable a multi-cloud scenario without employing virtual machine image; and (iv) it considers the deployment of the selected configurations (products) in the clouds. To the best of our knowledge, there is no approach in the literature that considers all these points. In other words, the contribution of our work is the following: it handles the variabilities at the IaaS layer, including support for the whole environment (hardware and software) independent of cloud provider; and it enables resource allocation in a multi-cloud scenario.

In the rest of this chapter, we present and evaluate our method to handle multi-cloud variabilities at the infrastructure layer. The method uses extended feature model (EFM) with attributes (Section 9.4) to describe the resources and their qualitative and quantitative characteristics. We employ constraint programming (CP) to select the resources through the Choco [178] constraint satisfaction problem solver. Choco [178] was used since it is a well-known satisfaction problem solver, and also because it has been successfully used by other works for such purpose (Section 9.8). We developed a prototype to evaluate the whole process (Figure 9.1) and our model, considering two different cloud providers and multiple users' objectives (Section 9.7).

In section 9.2, we describe the motivation and challenges addressed by our model followed by an overview of multi-objective optimization problem (MOOP) (Section 9.3).

9.2 Motivation and Challenges

Currently, with the existing techniques, before executing an application in the cloud, the users have to select the most suited resources for their applications and to configure them accordingly. Otherwise, the cost and the execution time can be very high. In other words, the users have to know the characteristics and the technical requirements of the resources as well as the behavior of their applications. Moreover, to reduce the risks of losing their work due to cloud failures, it is interesting to consider a multi-cloud scenario. A multi-cloud scenario may also be necessary since a single cloud may have a limited number of resources or its resources may not meet the users' constraints.

In this context, the overall objectives of the users can be defined as: (i) minimize the financial cost to execute applications across multiple clouds, and (ii) maximize the usage of configurations that have both best resource capabilities and performance without needing to deal with low-level technical details.

Nevertheless, there exist some open important challenges to address in order to help users on achieving these objectives. Some of these challenges are:

challenge 1: capturing clouds heterogeneity and constraints. Normally, clouds' resources are offered within different geographic regions with different costs and constraints. In addition, they are often deployed in heterogeneous data centers in the same region. In other words, clouds' variabilities may happen at any level (e.g. region, data center, resource). Hence, the users have to read extensive documentation in order to ensure that their desired resources are available in the cloud and also to understand the constraints of each resource. For example, at the time of writing this thesis, the Amazon EC2 cloud at Virginia has four availability zones (data centers) but only three of them support SSD disks. Thus, in order to know these restrictions the users have to test each zone. Furthermore, the clouds usually employ different terms to describe their resources. For instance, Amazon EC2 uses Elastic Computing Unit (ECU) as a metric to express the CPU capacity of a virtual machine, while Google Compute Engine uses Google Compute Engine Units (GCEUs). Finally, the clouds often employ high-level terms to describe the performance of their resources such as low, moderate, and high, which limits a decision based only on the resources' descriptions.

challenge 2: matching application requirements with the resources characteristics. Determining an optimal-configuration environment for an application in the cloud can be a difficult task for many reasons. First, cloud environments usually lack both performance stability and visibility due to hardware heterogeneity as well as the strategies used by hypervisors when allocating the virtual machines [269]. Cloud performance variation can range from 20% for CPU to 268% for memory [269]. This leads to a gap between the service levels expected by the users with the level delivered by the clouds [211]. This mostly occurs because the clouds' SLAs are normally based on resource availability without performance guarantees. In this context, the users have to benchmark their applications on multiple resources, considering various optimization parameters. For instance, figure 9.2 shows the average network bandwidth of an Amazon EC2 instance type designed to deliver 10 Gbps of network throughput. However, this throughput is disabled by default and to enable it the users must (a) install and activate a network driver in the instance's operating system; (b) stop the virtual machine and enable a feature called *enhanced networking*. In addition, they must know that this feature is only allowed on hardware-assisted virtualization type and to enable it, they have to call a method of the EC2's API. In other words, the users cannot enable this feature through the EC2 console (Web interface); and they must know all the technical details of the instance types. Thus, application benchmarking can increase the financial costs, it is often time-consuming, and demands system administration skills. Moreover, in some case, a global view of the environment is required in order to minimize the cost. For example, considering the scenario depicted in figure 9.2, the users must know the location of their resources to decide if data transfer between the resources must be done using an internal or an external IP address, as using the internal IP address implies zero costs. Finally, for some constraints, the users should balance between performance, cost, and availability as some requirements are only available in non-optimal configurations.

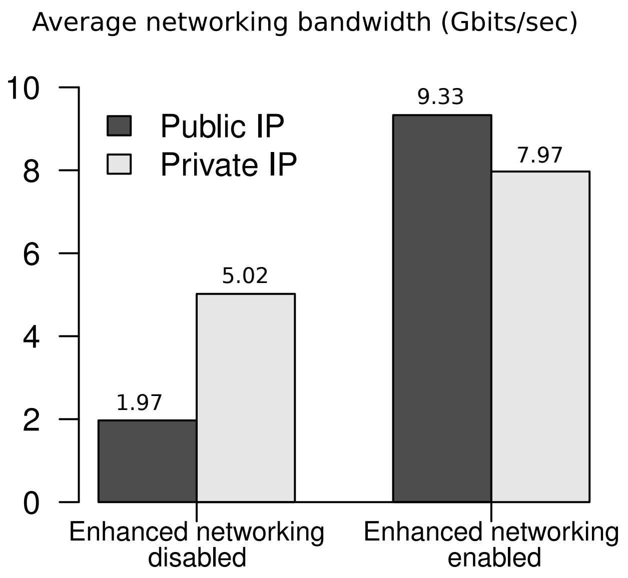


Figure 9.2: Average network bandwidth of the Amazon EC2 instance *c3.8xlarge* when created with the default configuration; and using an internal (private) and an external (public) address for data transfer. The networking bandwidth was measured between two *c3.8xlarge* instances deployed in the region of Virginia (availability zone us-*east-1a*) and running the iperf [168] application during one hour for each configuration scenario

challenge 3: describing a multi-cloud deployment environment. Currently, to deploy an application on multiple clouds the users have to configure each cloud individually. In some cases, they can create a virtual machine image (VMI) with their applications. However, the VMI can only handle software package descriptions leaving for the users the work of selecting a resource and for orchestrating the resources accordingly, i.e., the work of selecting and instantiating the VMI in each cloud. Hence, we need an approach that can describe the whole computing environment, independent of the cloud provider and that also enables automatic resources provisioning on multiple clouds taking into account temporal and functional dependencies between the resources, to leave the environment in a consistent state.

challenge 4: supporting auto-scaling decisions. Auto-scaling systems and resources on-demand are key features of cloud computing. Therefore, they depend on functional and non-functional data about the resources to avoid under or over-provisioning, as well as to consider clouds' capabilities. On the one hand, functional properties like storage size, type of operating system, and software packages can be handled by virtual appliances, but they may not be appropriate in a multi-cloud scenario due to the network traffic. On the other hand, cloud computing lacks an easy way to capture non-functional data such as the amount of time to install or to remove a software package; the average time to boot or to release a resource, and the location of the resources. Therefore, some of these data are only available via API calling, which can lead to vendor lock-in. Furthermore, in case of cloud's failure, it may be difficult for users to re-create the environment in another cloud.

9.3 Multi-Objective Optimization

A multi-objective optimization problem (MOOP) can be defined as a problem of finding a vector of decision variables which satisfies constraints and optimizes a vector function whose elements represent the objective functions. These functions form a mathematical description of performance criteria which are usually in conflict with each other [268]. In other words, multi-objective problems are such problems where the goal is to optimize k, often conflicting, objective functions simultaneously and to find a solution which would give the values of all the objective functions acceptable to the decision maker [246].

Therefore, multi-objective optimization problems can be formally defined as follows [352]:

$$\begin{aligned}
\textbf{optimize} \quad & y = F(\vec{x}) = (f_1(\vec{x}), f_2(\vec{x}), \ldots, f_k(\vec{x})) \\
\textbf{subject to} \quad & g_i(\vec{x}) = (g_1(\vec{x}), g_2(\vec{x}), \ldots, g_m(\vec{x})) \leq 0 \\
\textbf{where} \quad & \vec{x} = (x_1, x_2, \ldots, x_n) \in \Omega \\
& \vec{y} = (y_1, y_2, \ldots, y_k) \in \Lambda
\end{aligned} \tag{9.1}$$

where $\vec{x}$ is an n-dimensional decision variable vector; $\vec{y}$ is an n-dimensional objective vector; Ω is denoted as the decision universe ($\Omega = \{x \in \mathfrak{R}^n\}$); and Λ is called the objective region ($\Lambda = \{y \in \mathfrak{R}^k\}$). The constraints $g_i(\vec{x}) \leq 0$ determine the set of feasible solutions.

A feasible set $X_f \in \Omega$ is defined as the set of decision vectors $\vec{x}$ that satisfy the constraints $g_i(\vec{x}) \leq 0$:

$$X_f = \{x \in \Omega \mid g_i(\vec{x}) \leq 0\} \tag{9.2}$$

Thus, a multi-objective optimization problem (MOOP) consists of n decisions variables, m constraints, and k objectives of which any or all of the objectives functions can be linear or also non-linear. The MOOP's evaluation function maps decision variables ($\vec{x} = x_1, \ldots, x_n$) to objective vectors ($\vec{y} = y_1, \ldots, y_k$). In other words, objective vectors are images of decision vectors as shown in figure 9.3.

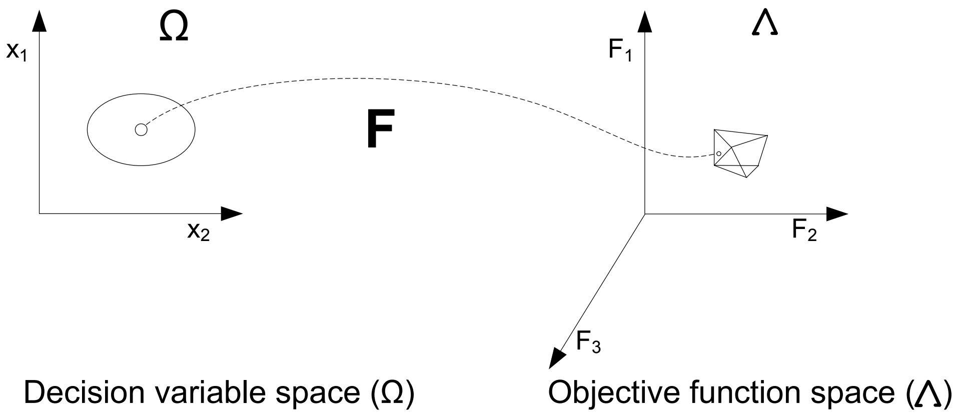


Figure 9.3: MOOP evaluation mapping [352]

Unlike single-objective optimization problems (SOOPs), which may have a unique solution for the objective function, an MOOP often presents an infinite [83] set of solutions that, when evaluated, produce vectors whose components represent the trade-offs in the objective space. For instance, in the design of a computer architecture under reliability constraints, cost and performance conflict. An optimal solution would be a computer system that achieves maximum performance at minimal cost without violating any constraint. Thus, if such solution exists, we only have to solve a SOOP [312]. In this case, the optimal solution for one objective is also optimum for the other objective. However, if the individual optima corresponding to the objective functions are different, there are conflicting objective functions that cannot be optimized simultaneously [394]. This is the case of the previous example, since high-performance systems usually have high cost; and low-cost architectures commonly provide low performance. Furthermore, in a single-objective optimization problem, the feasible set is completely ordered according to an objective function f, whereas it is partially ordered in multi-objective optimization problems.

An MOOP solution can be best, worst, and indifferent according to the objective values (Figure 9.4). Indifferent solutions means solutions neither dominating nor dominated with respect to each trade-off. A best solution, on the other hand, means a solution not worst in any of the objectives and at least better in one objective than the other [3]. A solution is considered optimal if none of its elements can be improved without deteriorating any other objective. Such solution is called *Pareto optimal* and the entire set of optimal trade-off is called *Pareto-optimal set* [84].

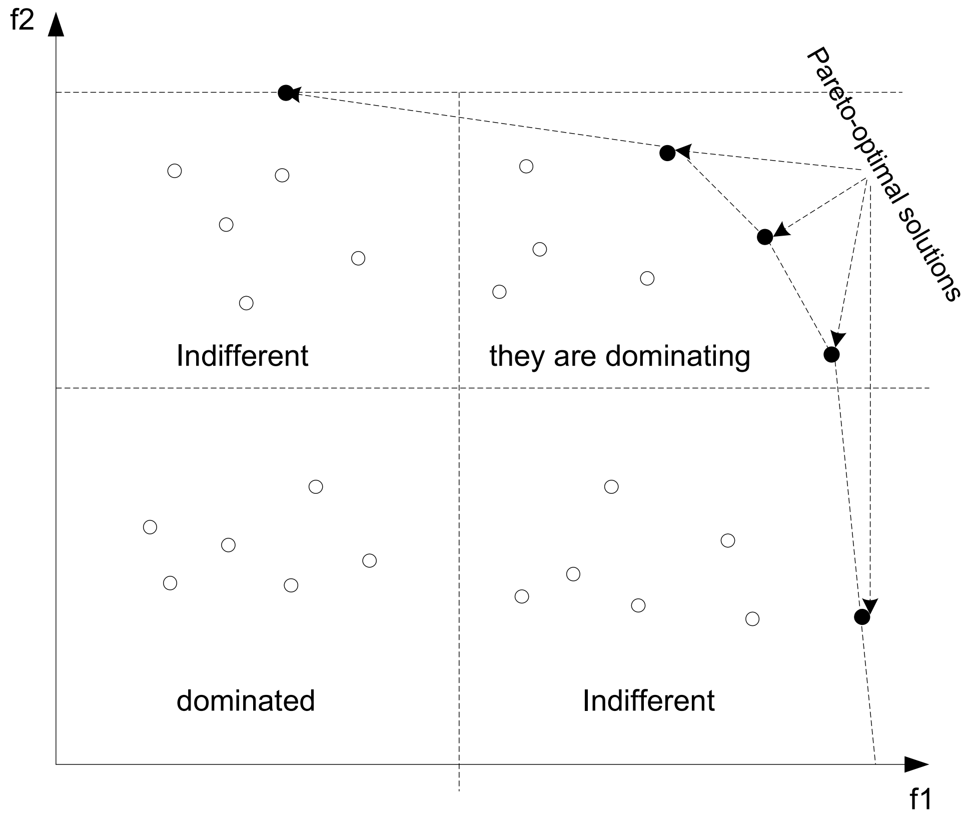


Figure 9.4: Pareto optimality for the objective space $F(\vec{x}) = (f_1(\vec{x}), f_2(\vec{x}))$ [119]

In the following, important Pareto concepts used in multi-objective optimization problems are presented [352][1]:

1. **Pareto Dominance**: For any two decision vectors $\vec{u} = (u_1, \ldots, u_k) \in \Omega$ and $\vec{v} = (v_1, \ldots, v_k) \in \Omega$:

$$
\begin{aligned}
&\vec{u} \prec \vec{v} \text{ ($\vec{u}$ dominates $\vec{v}$)} && \Longleftrightarrow \forall\, i \in \{1, \ldots, k\},\ f_i(\vec{u}) > f_i(\vec{v}) \\
&\vec{u} \preceq \vec{v} \text{ ($\vec{u}$ weakly dominates $\vec{v}$)} && \Longleftrightarrow \forall\, i \in \{1, \ldots, k\},\ f_i(\vec{u}) \geq f_i(\vec{v}) \\
&\vec{u} \sim \vec{v} \text{ ($\vec{u}$ is indifferent to $\vec{v}$)} && \Longleftrightarrow \exists\, i \in \{1, \ldots, k\},\ f_i(\vec{u}) \geq f_i(\vec{v}) \\
& && \quad \wedge \exists\, j \in \{1, \ldots, k\},\ f_j(\vec{u}) < f_j(\vec{v})
\end{aligned}
\tag{9.3}
$$

   In other words, the decision vector $\vec{u}$ dominates $\vec{v}$ if and only if, $\vec{u}$ is as good as $\vec{v}$ considering all objectives, and $\vec{u}$ is strictly better than $\vec{v}$ in at least one objective.

2. **Pareto Optimality**: A solution $x \in \Omega$ is said to be Pareto optimal regarding to Ω if and only if there is no $x' \in \Omega$ for which $\vec{v} = F(x') = (f_1(x'), \ldots, f_k(x'))$ dominates $\vec{u} = F(x) = (f_1(x), \ldots, f_k(x))$. The set of all Pareto-optimal solutions is called the Pareto-optimal set. The corresponding set of objective vectors is known as the non-dominated set, surface or Pareto-optimal front. In practice, it is common for these terms to be used interchangeably to describe solutions of an MOOP [119].

[1]Without loss of generality, a maximization problem is assumed

3. **Pareto Optimal Set**: For a given MOOP function $F(x)$, the Pareto optimal set (P^*) is defined as:

$$P^* = \{x \in \Omega \mid \nexists\, x' \in \Omega : F(x') \preceq F(x)\} \tag{9.4}$$

4. **Pareto Front**: For a given MOOP function $F(x)$ and Pareto optimal set P^*, the Pareto front (PF^*) is defined as:

$$PF^* = \{F(\vec{x} = (f_1(\vec{x}), \ldots, f_k(\vec{x})) \mid x \in P^*\} \tag{9.5}$$

9.4 Feature Modeling

Feature modeling is a software engineering activity used for capturing commonalities and variabilities in a SPL. It was introduced in the early 1990s as a part of the feature-oriented domain analysis (FODA) methodology [182]. A SPL is a set of software systems that share a set of features that satisfy the needs of a particular domain or mission and are developed from a common set of assets in a prescribed way [82]. SPL engineering usually uses feature models to define its assets and their valid combinations [32].

A feature model describes the concepts (i.e., features) of a domain, and details the relationships between them [92]. As an example, consider the variability of a virtual machine, depicted in figure 9.5. Hence, each functional property (i.e., operating system, hardware, purpose usage, and placement group) is represented as a feature. In practice, a feature model is a tree, where each node represents a feature of a product (or solution) and the relation between a parent (or compound) feature and its child features (i.e., subfeatures) are categorized as [182]:

And: all *subfeatures* must be selected. The features *placement group* and *cluster* have an *And* relationship.

Optional (variability): a parent feature does not imply the child feature. For instance, not all virtual machines have a *placement group*.

Mandatory (commonality): whenever a parent feature is selected, the child feature must also be selected. For example, all instances have a *hardware*, an *operating system*, and a *purpose usage*.

Or: at least one child feature must be selected when its parent feature is. In figure 9.5, whenever *storage type* is selected, the features *provisioned*, *ebs* or both must be selected.

Alternative: exactly one child feature must be selected. For example, a virtual machine has only one *operating system* and it can only be either *CentOS*, *Debian* or *Ubuntu*.

Notice that a feature may have multiple child features but only one parent feature, and that a child feature can only appear in a product if its parent does.

Feature models fit into a problem space, as they determine what products are valid in a particular domain. According to figure 9.5, the configuration (or product): {*Debian, Ivy Bridge, dedicated, ten GB, one hundred GB, provisioned*, and *server*} is valid, whereas the one {*Ivy Bridge, dedicated, ten GB, one hundred GB, provisioned*, and *server*} is invalid since it does not specify an operating system.

Besides these relationships, constraints can be also specified using propositional logic to express dependencies among the features. Usually inclusion or exclusion statements describe constraints in the form: *if feature F is selected, then features A and B must also be included or excluded*. For instance, constraint c_1 (Figure 9.5) indicates that selecting the feature *bootstrap* excludes features *cluster, Ivy Bridge, Sandy Bridge, dedicated, ten GB, one hundred GB, and provisioned*. In other words, selecting the feature *bootstrap*, reduces the configuration space to only the operating system.

Moreover, features can be classified as abstract or concrete [345]. Abstract features are used to structure future models, and they do not have any instance[2] as they represent domain decisions [345]. Concrete features, on the other hand, represent instances (i.e., products). For instance, *operating system* is an abstract feature whereas *Ubuntu* is a concrete feature.

Feature models may also have attributes devoted to a feature, which is known as *extended feature model (EFM)*. Such attributes often represent non-functional properties such as cost, power consumption, performance, among others. Although there is no consensus on a notation to define attributes, most of the proposals agree that an attribute should be at least a triple with a *name*, a *domain*, and a *value* [37]. In figure 9.5, the feature *storage* has the attribute *size*.

Finally, feature models may use cardinalities to express the relationships between their features. These relationships are classified as [37]:

feature cardinality: determines the number of instances of a feature that can be part of a product. It is denoted as an interval $[n..m]$, where n is the lower bound and m is the upper bound. This relationship may be used as a generalization of the *Mandatory* ($[1, 1]$) and the *Optional* ($[0, 1]$) relationships.

group cardinality: limits the number of child features that can be included in a product when its parent feature is selected. A group cardinality is denoted by its multiplicity ($\langle n..m \rangle$) comprising a lower and an upper bound value. The multiplicity value defines how many instances of a group must be at least and at most presented in a variant configuration. Table 9.1 shows how the relations *Optional*, *Mandatory*, *Or*, and *Alternative* are expressed using group cardinality.

One important characteristic of feature models is that they help the users on organizing concepts in a structured and hierarchical manner, and to define its assets and their valid combinations [32]. In this context, a valid member of a domain model satisfies all the constraints in the corresponding feature model. Moreover, feature models are usually understood by non-technical users, since they refer to domain concepts.

[2]In this context, an instance means a product or a software *artifact*.

Table 9.1: Using group cardinality in feature model diagrams [293]

Relationship	Group cardinality	Number of features
Optional	0..1	1
Mandatory	1..1	1
Or	1..*	n
Alternative	0..1	1

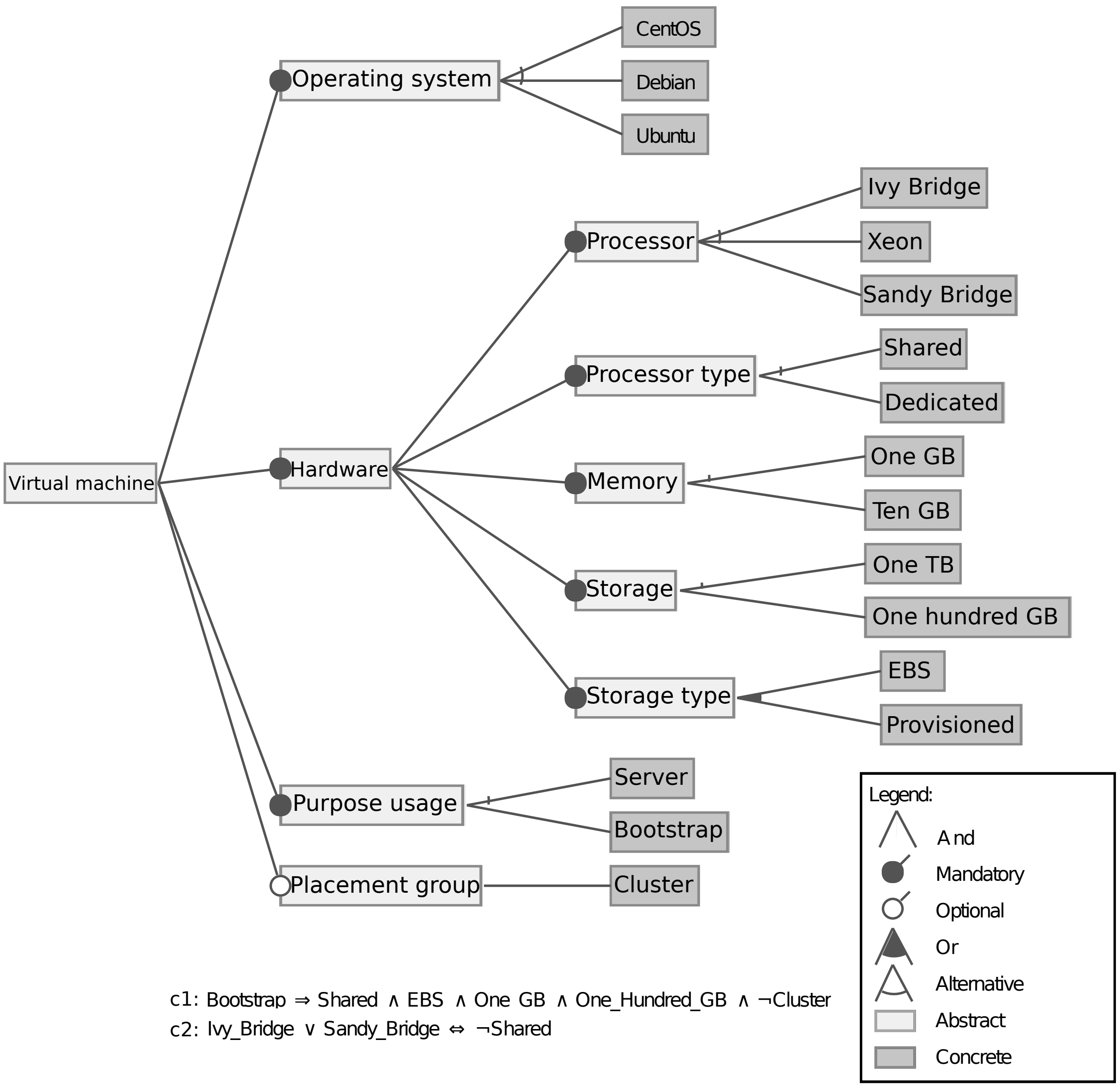


Figure 9.5: Example of a feature model with *And*, *Optional*, *Mandatory*, *Or*, and *Alternative* features

9.5 Proposed Model

This section presents our cloud model to handle the variability points of IaaS clouds. It aims to enable resource selection based on quantitative attributes and on the users' objectives. For a better understanding, table 9.2 presents the concepts/symbols used in this section and their meaning.

Table 9.2: Notation of the model

Name	Meaning
c_j	Cloud j
q	Number of clouds
$r_{i,j}$	Resource i from cloud j
$lc_{i,j}$	Network bandwidth between the resources i and j
$it_{i,j}$	Instance type i of cloud j
$lag_{i,j}$	Acquisition time of instance i in cloud j
$itd_{i,j,k}$	Disk i of instance type j in cloud k
$\Theta_{i,j}$	Set of disks from instance type i in cloud j
$est^{w}_{i,j}$	Estimated execution time of workload k in resource i which belongs to cloud j
$cost_{i,j}$	Financial cost of resource i in cloud j
$\sigma_{i,j}$	Maximum number of virtual machines allowed for an instance type i in cloud j
$vmi_{i,j}$	Virtual machine image i in cloud j
$\eta_{i,j}$	Maximum number of disk allowed for the instance type i in cloud j
$\omega_{i,j}$	Total amount of disk space, in gigabytes, that can be mounted by instance type i in cloud j
tps_j	Maximum aggregate disk space that can be provisioned in cloud j
$dt_{i,j}$	Disk type i in the cloud j
$min_{i,j}$	Minimum size in gigabytes of a disk type i in cloud j
$max_{i,j}$	Maximum size in gigabytes of a disk type i in cloud j
$disk_{i,j,k}$	Disk i of type j in cloud k
ds_i	Size in gigabytes of disk i
dp_i	Performance in IOPS of disk i
dth_i	Technology of disk i
$\rho_{i,j}$	Size of virtual machine image i, in gigabytes, in cloud j
$vt_{i,j}$	Virtualization technique of virtual machine image i in cloud j
$sp_{i,j}$	Software packages of virtual machine image i in cloud j
v_k	Maximum number of virtual machines in cloud k
$\delta_{j,k}$	Number of virtual machines of type j hosted in cloud k
$vm_{i,j,k}$	Virtual machine i of type j hosted in cloud k
$bd_{i,k}$	Boot disk of virtual machine i in cloud k
$nad_{i,k}$	Number of attached disks to a virtual machine i in cloud k
$\Psi_{i,k}$	Total disks size mounted by VM i in cloud k
$H_{i,j,k}(t)$	Matrix of virtual machines allocation ($vm_{i,j}$ hosted in cloud k at the time t)
$price_i$	Real price of virtual machine i
pm_i	Price model of virtual machine i. For example, on-demand, spot, reserved
$az_{i,k}$	Zone (data center) where virtual machine i of cloud k is deployed
$g_{i,k}$	Group of the virtual machine i in cloud k
ic_i	A metric that determines the capacity of virtual machine i (c.f. table 3.2)
ipt_i	Performance trust of virtual machine i (c.f. table 3.2)
ν_i	Internal networking cost of cloud i
λ_i	External networking cost of cloud i
φ_i	Networking cost of cloud i
ϕ_i	Storage cost of cloud i

9.5.1 Cloud Computing Model

We assume a cloud computing system with a set of q clouds denoted as $C = \{c_1, c_2, \ldots, c_q\}$. The clouds are defined as an undirected graph $G(R, L)$ where $R = \{r_{i,j}\}, 1 \leq j \leq q \wedge 1 \leq i \leq n_j$ represents a set of virtual machines (VMs)[3] deployed across the clouds, and L is a set of edges connecting these VMs. Each edge $(r, r') \in L$ has a capacity value $lc_{r,r'}$, representing the network bandwidth between the VMs r and r', with $lc_{r,r'} > 0$. A virtual machine belongs to an instance type and it has a price model, a virtual machine image (VMI), and at least one disk. Additionally, a VM is deployed in a zone (data center) and it may belong to a group. In the remainder of this section j usually refers to the g cloud where the resource stands.

9.5.1.1 Instance Type Model

A cloud $c_j, 1 \leq j \leq q$ offers a set of m_j instance types denoted by $IT_j = \{it_{1,j}, it_{2,j}, \ldots, it_{m,j}\}$. The instance types have different capabilities such as processing power, network throughput, and a cost per use. The cost is defined per unit of time (e.g., per hour) and any partial usage is rounded up to the next time unit. For instance, in a hourly-based scenario, the cost for using an instance during one hour and one minute is the same as using it during two hours. Moreover, an instance type has a family type that determines a target workload (e.g., CPU-, Memory-, I/O-optimized), i.e., a purpose usage. Furthermore, it has a non-negligible and varying acquisition time shortly denoted as $lag_{i,j}, 1 \leq i \leq m \wedge 1 \leq j \leq k$. Additionally, an instance type $it_{i,j}$ may have a set of disks denoted by $\Theta_{i,j} = \{itd_{1,i,j}, itd_{2,i,j}, \ldots, itd_{d,i,j}\}, d \in \mathbb{N}^{+}$. Finally, a cloud may limit the number of instances of each type that can be acquired by a user.

Formally, an instance type $it_{i,j}$ is defined as a tuple with an expected execution time for a given workload ($est^{\mathrm{w}}_{i,j}$), a hourly-based price ($cost_{i,j}$), an acquisition time ($lag_{i,j}$), the maximum number of instances ($\sigma_{i,j} \in \mathbb{Z}^{*}$) allowed for it, and a set of disks ($\Theta_{i,j}$).

$$it_{i,j} = < est^{\mathrm{w}}_{i,j}, cost_{i,j}, lag_{i,j}, \sigma_{i,j}, \Theta_{i,j} > \tag{9.6}$$

Additionally, an instance type may have a restriction on the maximum number of disks and on the total amount of disk space that can be mounted simultaneously. Let $\eta_{i,j} \in \mathbb{N}^{*}$ be the maximum number of disk allowed for an instance type i, $\omega_{i,j}$ be the total amount of disk space, in gigabytes, that can be mounted by a type i, and tps_j the maximum aggregate disk space that can be provisioned by a user in the cloud j.

9.5.1.2 Disk Model

A cloud offers a set of disk types to be used by a virtual machine. Example of disk types are *ephemeral* or *instance disks*, *persistent*, and *object store*. Let $dt_{i,j}$ be a disk type i in the cloud j. In this case, $dt_{i,j}$ is a tuple with a minimum ($min_{i,j}$) and a maximum ($max_{i,j}$) size in gigabytes, and a cost per gigabytes/month ($cost_{i,j}$).

$$dt_{i,j} = < min_{i,j}, max_{i,j}, cost_{i,j} > \tag{9.7}$$

[3]The terms instance, node, and virtual machine are used interchangeably in this chapter

In this context, a disk ($disk_{i,j,k}$) has a type ($dt_{j,k}$), a size in gigabytes (ds_i), a performance (dp_i) defined as the number of input/output operations per second (IOPS), and a disk technology (dth_i). Example of disk technologies are *standard*[4] and *SSD*.

$$disk_{i,j,k} = < dt_{i,k}, ds_i, dp_i, dth_i > \tag{9.8}$$

where, $min_{j,k} \leq ds_i \leq max_{j,k}$ and $\sum_{i=1}^{n} ds_i \leq tps_k$.

9.5.1.3 Virtual Machine Image Model

A virtual machine image (VMI) is a disk that contains an operating system and a root file system required to start a virtual machine. Moreover, it is designed to run on a hypervisor (i.e., virtualization technique), and it may have a set of software packages. Formally, a VMI ($vmi_{i,j}$) is a tuple with a cost per hour ($cost_{i,j}$), a size in gigabytes ($\rho_{i,j}$), a virtualization technique ($vt_{i,j}$), and a set of software packages ($sp_{i,j}$).

$$vmi_{i,j} = < cost_{i,j}, \rho_{i,j}, vt_{i,j}, sp_{i,j} > \tag{9.9}$$

with $\rho_{i,j} \leq tps_j$.

9.5.1.4 Instance Model

An instance is a virtual machine (VM) deployed in a cloud k. A VM_i ($vm_{i,j,k}$) belongs to only one instance type (j) and to only one cloud (k). It cannot be migrated, and a cloud may limit the number of instances that can be acquired.

Let $\delta_{j,k}$ be the number of virtual machines of type j hosted by cloud c_k. Thus, $\delta_{j,k} \leq \sigma_{j,k}$, and the maximum number of virtual machines in the cloud (c_k) is:

$$v_k \leq \sum_{j=1}^{m} \delta_{j,k} \tag{9.10}$$

An instance requires a virtual machine image ($vmi_{i,k}$) and a boot disk ($bd_{i,k}$) with $ds_i \geq \rho_i$, where ds_i is boot disk ($bd_{i,k}$) size. Additionally, it may have a set of attached disks. Let $nad_{i,k}$ be the number of attached disks of a virtual machine i, and $\Psi_{i,k}$ the current total disk size mounted by VM i. Thus, $nad_{i,k} \leq \eta_{j,k}$ and $\omega_{j,k} \leq \Psi_{i,k} \leq tps_k$. In other words, the number and amount of disks attached by a virtual machine must be at most the allowed for its instance type.

We use a binary integer matrix $H_{i,j,k}(t)$ to represent the deployment of a VM in a cloud at the time (t), where:

$$H_{i,j,k}(t) = \begin{cases} 1 & \text{if } vm_{i,j} \text{ is hosted by the cloud } c_k \text{ during the period } t \\ 0 & \text{otherwise} \end{cases} \tag{9.11}$$

[4]Standard disks are often network storages that can be mounted by a virtual machine, which offer the same functions of a regular hard disk attached to a computer.

Moreover, a instance runs in a zone (data center) and it can belong to a group.

An instance $vm_{i,j,k}$ can be defined as a tuple with a price, a pricing model (pm_i), a virtual machine image ($vmi_{i,k}$), a boot disk ($bd_{i,k}$), a zone ($az_{i,k}$), a group ($g_{i,k}$), and the metrics: instance capability (IC) and performance trust (IPT), detailed in section 3.3.2.

$$vm_{i,j,k} = < \mathrm{price}_i, \mathrm{pm}_i, vmi_{i,k}, bd_{i,k}, \mathrm{ic}_i, \mathrm{ipt}_i > \tag{9.12}$$

9.5.2 Cost Model

9.5.2.1 Networking and Storage Cost

Each cloud may have different monetary costs for data transfer (i.e., network pricing) and data provisioning (i.e., storage).

The data transfer cost consists of the *inbound* and *outbound* price per gigabytes. In other words, it is the network cost per amount of data transferred from/to a cloud's resource. It is defined as internal when the resources are located at the same cloud or external when one resource belongs to another cloud (i.e., the Internet traffic). Let ν_i and λ_i be respectively the internal and external networking costs of the cloud i. Thus, the networking cost can be defined as:

$$\varphi_i = \nu_i + \lambda_i \tag{9.13}$$

The data provisioning, on the other hand, is the monetary storage cost. The clouds often consider three storage costs: the provisioned space cost (i.e., disks cost), the snapshot storage cost, and the image storage cost. The provisioned space is the cost of the provisioned disks. The snapshot storage cost is the price to pay for taking a snapshot of an instance and to store it in the cloud. Finally, the image storage cost is the price to store a virtual machine image (VMI) in a cloud. These costs are defined in gigabytes per dollar. Let ϕ_k be the total storage cost in the cloud k. It can be defined as follows:

$$\phi_k = \sum_{i=1}^{n} ds_i * cost_{i,k} + \sum_{j=1}^{n} ds_j * cost_{j,k} + \sum_{l=1}^{n} ds_l * cost_{l,k} \tag{9.14}$$

where ds_i, ds_j, and ds_l are respectively the size of: disk i, snapshot j, and image l in gigabytes.

9.5.2.2 Instance Cost

A virtual machine can be acquired following three pricing models: on-demand, spot, and reserved. The on-demand model has a fixed cost. The spot model, on the other hand, has a varying model, where the users can specify the maximum value that they would like to pay. Finally, in the reserved model, the users pay in advance to use the cloud during a fixed time.

Let $P_{i,j,k}(t)$ be the real price of the instance i of type j deployed in the cloud k during the period t. It can be defined as:

$$P_{i,j,k}(t) = \begin{cases} cost_{j,k} & \text{if } i \text{ is an on-demand instance} \\ sp_{i,j,k} & \text{otherwise} \end{cases} \tag{9.15}$$

where, $cost_{j,k}$ is the price defined by the cloud provider for the instance type, and $sp_{i,j,k}$ is the user's bid for the virtual machine. We assume that these prices are known before the use of an instance.

Hence, the cost of using a cloud c_k at the time (t) can be defined as follows:

$$Cost(t) = (\sum_{i=1}^{n}\sum_{j=1}^{v}\sum_{p=1}^{k} H_{i,j,p}(t) * P_{i,j,p}(t)) + \varphi_p + \phi_p \tag{9.16}$$

In other words, it is the cost of using each instance at the time t plus the storage and network costs. Finally, the total cost of using a cloud during $h \geq 1$ hours is:

$$totalCost(h) = \sum_{t=1}^{h} Cost(t) \tag{9.17}$$

This model should be included in a scheduling algorithm to provision and to schedule the virtual machines taking into account the characteristics of the applications and the cost/performance of the virtual machines.

9.6 Modeling IaaS Clouds Configuration Options with Feature Model

For a reference example, consider two different user groups: one group comprises non-technical users or users who have only high-level knowledge about either system administration or cloud computing, whereas the second group comprises specialized users (i.e., system administrators). While users belonging to the former group might be interested in creating their computing environment based on higher-level descriptions such as CPU, memory size, and operating system, the latter may want to create the environment (or at least to have the option to create it) based on fine-grained options such as hypervisor, virtualization type, storage technologies, among others.

To meet the objectives of these two groups, we can follow a product line engineering (PLE) process (Figure 9.1). The PLE has two major phases: *domain engineering* and *product engineering*. In the domain engineering phase, a variability model is defined for the product line. This model includes the variation points and the commonalities of the products. It can also include the constraints between the variations and the products. The product engineering phase, on the other hand, is responsible for deriving the products from the model established in the domain engineering phase [48].

In both phases, we can use feature models to describe the products. In this case, the output of the domain engineering phase is an abstract feature model. After, a concrete model is created with the products. Figure 9.6 shows our abstract model to handle the configuration options at the infrastructure layer. Its functional properties are guided by the taxonomies available in the literature [158, 284, 320, 353, 383], and it uses attributes to describe the qualitative characteristics of the resources such as network throughput and CPU capacity (i.e., sustainable performance). With this model, the users can see the kind of resources that are available in IaaS clouds, as well as their constraints and variabilities such as disk type, location, among others.

In figure 9.6, a *family type* determines the characteristics and the recommended usage for an *instance type*. This usage is based on the amount of resources available for the instances such as CPU, memory, and accelerators. Moreover, the members of a family type often have the same hardware configuration, i.e., they are homogeneous with regard to the underlying infrastructure. Unlike the family type *shared*, all other instance types offer a fixed amount of resources. Furthermore, instance types that do not have a defined purpose usage are classified as *general*. Although some clouds and models [130] add a new level to classify the size of an instance type using terms such as medium, large, xlarge, among others, our model does not for two reasons. First, these classifications differ only in the amount of resources provided by each instance that can be handled by using attributes. Second, these classifications require additional data to describe an instance type, since instances belonging to different family types may receive the same classification.

As we said in section 3.4, some instance types may have physical disks attached to the host computer of a virtual machine, i.e., *instance disks*. These disks are normally *ephemeral*, which cannot be used as the primary disk of the instance. In this case, a VM can mount the instance disks, but it must have at least one *persistent* disk. It is the virtual machine image which defines the persistent disk. In other words, the VMI defines the root file system of a VM as well as its minimal size. In figure 9.6, *HVM* and *PVM* are respectively hardware assisted and paravirtualization techniques described in section 3.1.1.2.

Additionally, virtual machines can be placed in a group to decrease network latency and/or to increase network throughput, for instance, i.e., they can be organized in a cluster. In this case, the instances must be in the same zone (data center). However, a cloud provider may restrict the instance types that can be placed in a cluster. This feature enables us to reference the cluster instead of each virtual machine individually, and it can also used to model a physical cluster. For example, to represent the Tompouce cluster (Figure 2.2), we can modeled one virtual machine with the total number of cores (i.e.,#vcpu) and GFlops of the cluster or we can model all nodes and assign them the same group.

Finally, as each region have at least one zone, we model this data through a constraint, assuming that there is a function that returns all zones of a region, and that this function is used to validate the zone assigned to a *disk* or to a *group*. Moreover, a virtual machine's zone is always the same of its attached disks.

9.7 Experimental Results

To validate our model, we modeled entirely two different cloud providers: Amazon EC2 and GCE. As in the PLE process, the first step consists of implementing the variability model and the second one of representing the clouds with this model, we implemented the models and a system to manage the models in Java to allow the users to instantiate the clouds, as depicted in figure 9.7. The system works as follows. First, the data about

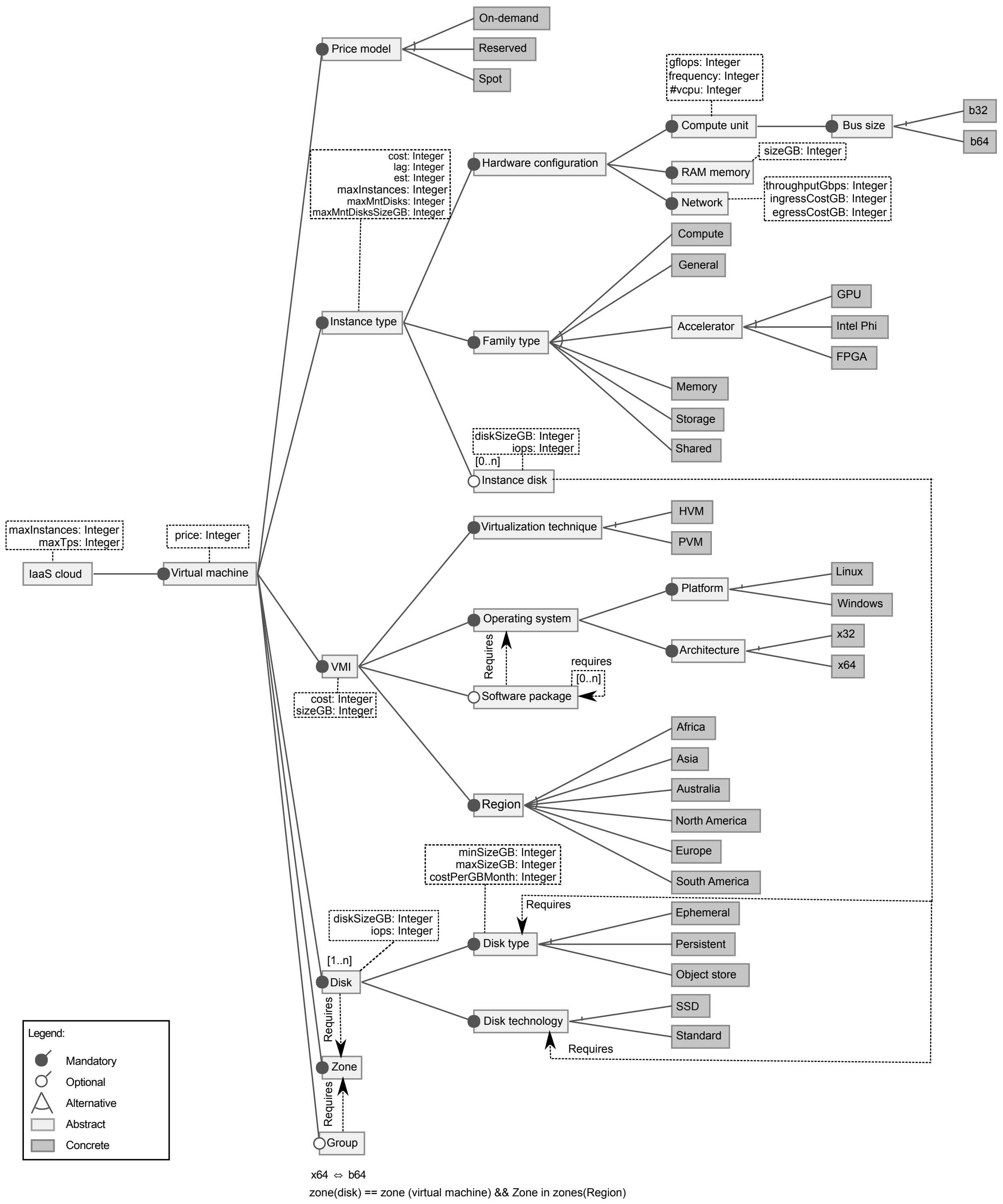


Figure 9.6: Abstract extended feature model of IaaS clouds

the clouds are stored in a database. These data include the costs, the restrictions in each region of the clouds, and the performance of the instance types obtained with benchmarks. Next, the users define their constraints and submit them to a system. Then, the system reads the database to load the data about the resources and uses Choco [178], a constraint satisfaction problem solver, to find the configurations that meet the objectives. Next, the valid configurations are sent for the users, who can see all the solutions, the best solutions, and the solutions in the Pareto front. Finally, the users can select the configurations and request the system to deploy them in the clouds, or they can demand the system to based on the Pareto front suggests one without considering one objective (e.g., cost, CPU, memory).

Tables 9.3 and 9.4 show the instance types available in the clouds EC2 and GCE, respectively, and their cost in one region. Although each table shows the price of only one region, our system considers all regions of the clouds.

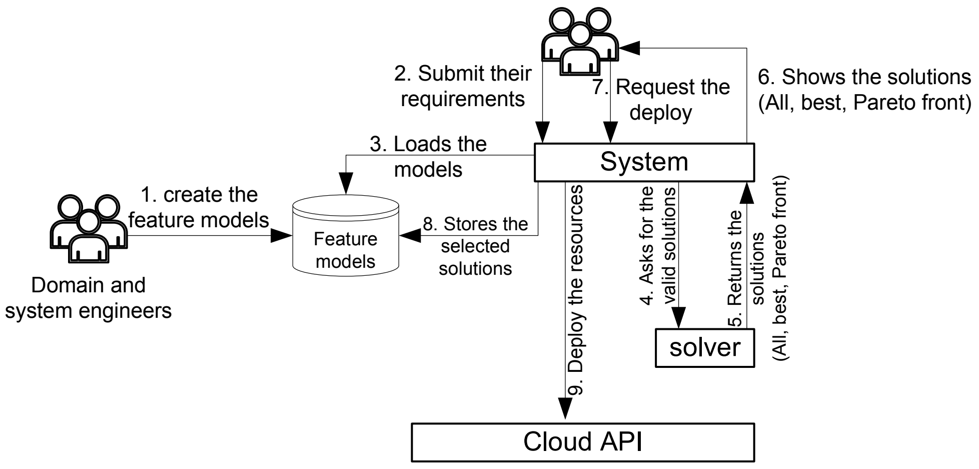

2. Submit their requirements
7. Request the deploy
6. Shows the solutions (All, best, Pareto front)
3. Loads the models
System
1. create the feature models
Feature models
8. Stores the selected solutions
Domain and system engineers
9. Deploy the resources
4. Asks for the valid solutions
5. Returns the solutions (All, best, Pareto front)
solver
Cloud API

Figure 9.7: Our process to select and to deploy the resources in the clouds

Based on the instance types of the clouds, figures 9.8 and 9.9 show the abstract model instantiated to describe two different products (virtual machines) of each cloud. With these models, the users can see that both clouds offer resources in different regions (Europe and North America), and that only EC2's instance types have ephemeral SSD disks with zero cost, for example.

Moreover, using these models, we can simulate resource migration between the clouds. In this case, we use the abstract model to describe the environment and the concrete models to select the resources. For example, a description like: { b64, #vcpu = 2, memorySizeGB = 15, General, HVM, Linux, x64, North America, Persistent, SSD, diskSizeGB = 30 } returns VM1 of GCE (Figure 9.9).

We conducted three benchmarks (Table 9.5) to evaluate the characteristics of the instance types. These benchmarks were required because: (i) the literature only has data about the first generation of EC2 instance types [167, 170, 242, 267, 269, 284], and we were interested in all generations as well as in the GCE's instance types. In addition, only few instance types have been evaluated by these works; and (ii) the literature lacks data about the network throughput of the instances.

In order to obtain the CPU performance, similar to [267, 284], we executed LINPACK [227] with five problem sizes ranging from 13,000 to 55,000. The information

Table 9.3: Amazon EC2 instance types and their cost in the region of Virginia

Instance type	**Virtual cores**	**Memory (GB)**	**Cost (USD) / Hour**	**Family type**
c3.large	2	3.75	0.105	Compute
c1.medium	2	1.7	0.130	Compute
c3.xlarge	4	7.5	0.210	Compute
c3.2xlarge	8	15.0	0.420	Compute
c1.xlarge	8	7.0	0.520	Compute
c3.4xlarge	16	30.0	0.840	Compute
c3.8xlarge	32	60.0	1.680	Compute
cc2.8xlarge	32	60.5	2.000	Compute
g2.2xlarge	8	15.0	0.650	GPU
cg1.4xlarge	16	22.5	2.100	GPU
m3.medium	1	3.75	0.070	General
t2.micro	1	1.0	0.013	General
t2.small	1	2.0	0.026	General
m1.small	1	1.7	0.044	General
m1.medium	1	3.75	0.087	General
m3.large	2	7.5	0.140	General
t2.medium	2	4.0	0.052	General
m1.large	2	7.5	0.175	General
m3.xlarge	4	15.0	0.280	General
m1.xlarge	4	15.0	0.350	General
m3.2xlarge	8	30.0	0.560	General
r3.large	2	15.0	0.175	Memory
m2.xlarge	2	17.1	0.245	Memory
r3.xlarge	4	30.5	0.350	Memory
m2.2xlarge	4	34.2	0.490	Memory
r3.2xlarge	8	61.0	0.700	Memory
m2.4xlarge	8	68.4	0.980	Memory
r3.4xlarge	16	122.0	1.400	Memory
r3.8xlarge	32	244.0	2.800	Memory
cr1.8xlarge	32	244.0	3.500	Memory
t1.micro	1	0.615	0.020	Shared
i2.xlarge	4	30.5	0.853	Storage
i2.2xlarge	8	61.0	1.705	Storage
i2.4xlarge	16	122.0	3.410	Storage
hs1.8xlarge	16	117.0	4.600	Storage
hi1.4xlarge	16	60.5	3.100	Storage
i2.8xlarge	32	244.0	6.820	Storage

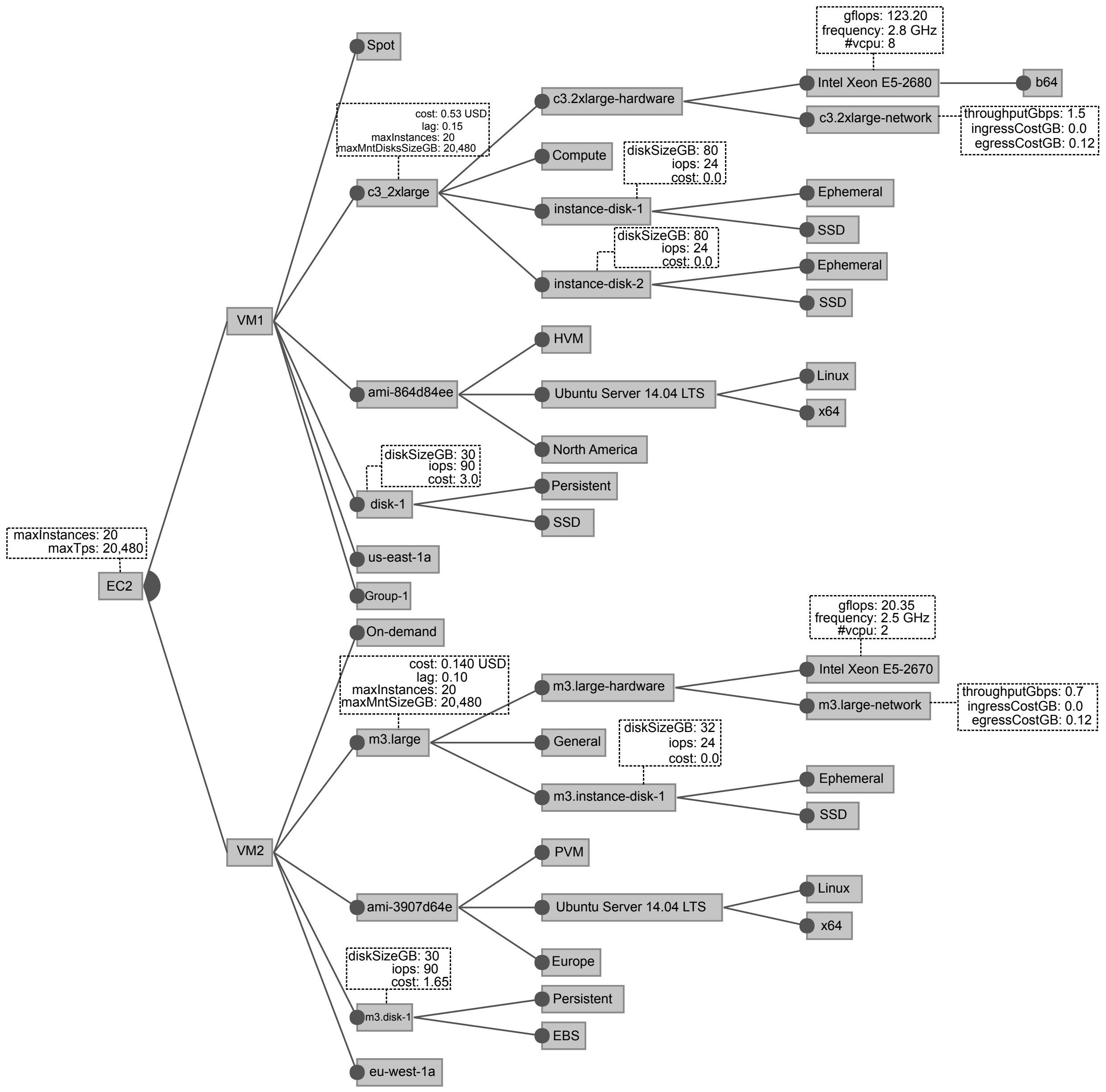


Figure 9.8: Example of the abstract extended feature model (Figure 9.6) instantiated to represent two products of Amazon EC2

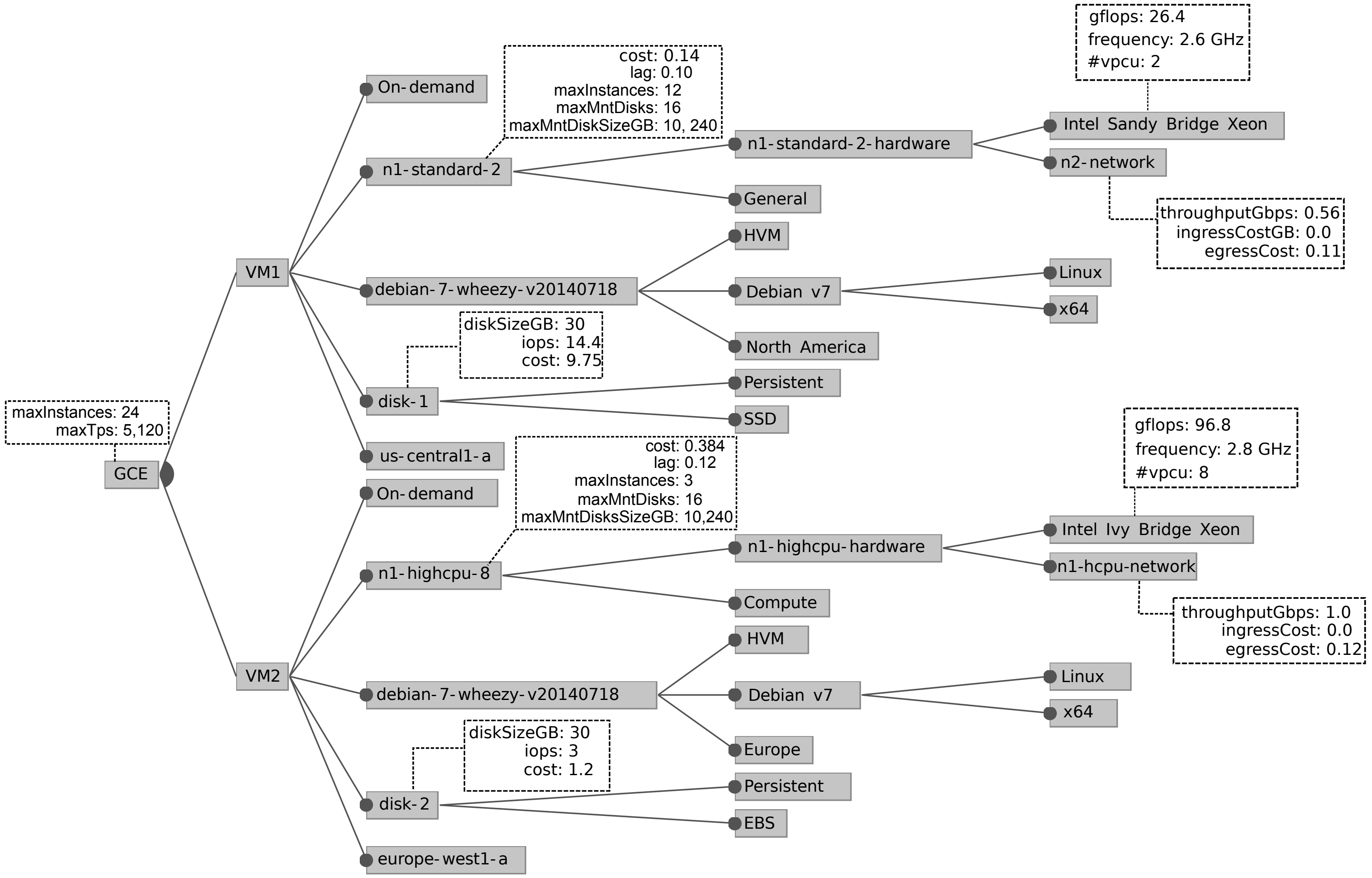


Figure 9.9: Example of the abstract extended feature model (Figure 9.6) instantiated to represent two products of GCE

Table 9.4: Google Compute Engine (GCE) instance types and their cost in the US region

Instance type	Virtual cores	Memory (GB)	Cost (USD) / Hour	Family type
n1-highcpu-2	2	1.8	0.088	Compute
n1-highcpu-4	4	3.6	0.176	Compute
n1-highcpu-8	8	7.2	0.352	Compute
n1-highcpu-16	16	14.4	0.704	Compute
n1-standard-1	1	3.75	0.070	General
n1-standard-2	2	7.5	0.140	General
n1-standard-4	4	15.0	0.280	General
n1-standard-8	8	30.0	0.560	General
n1-standard-16	16	60.0	1.120	General
n1-highmem-2	2	13.0	0.164	Memory
n1-highmem-4	4	26.0	0.328	Memory
n1-highmem-8	8	52.0	0.656	Memory
n1-highmem-16	16	104.0	1.312	Memory
f1-micro	1	0.6	0.013	Shared
g1-small	1	1.7	0.035	Shared

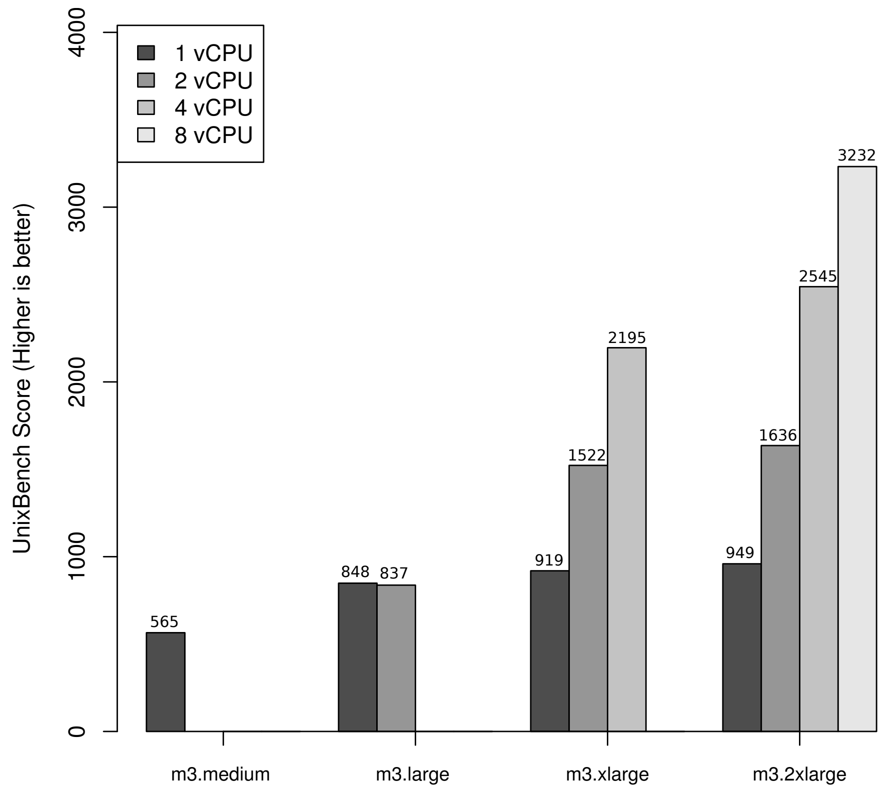
about the underlying hardware was acquired through the non-trapping *cpuid* instruction. The UnixBench [57] was employed to provide another way to compare the performance variations within the same cloud platform. UnixBench is a test suite for Linux systems to analyze their performance with regard to CPU, I/O, system call, among other operations. Based on the result of each test, an index score value is calculated, i.e., the UnixBench computes the index score value of a system using the SPARCstation 20 as baseline. This benchmark allows us to compare the CPU performance of the instances that belong to a family type but that offer a different number of virtual cores. Figure 9.10 shows the CPU performance of the general instance type of the clouds. The general instance type was evaluated since it is not bound to any application's characteristics, which can be used as a baseline to choose other specialized instance types. We observe that in GCE, small instances must be preferred when the applications do not use all the resources (e.g., virtual cores) of the instances, as the performance of a single core type is better than using a virtual core of an eight core instance type (Figure 9.10).

For network performance, we used iperf [168] to measure TCP and UDP throughput. Each measure has a client and a server component located in the same zone belonging to the same instance type. The measurements was done during one day for each type considering both internal and external data transfer.

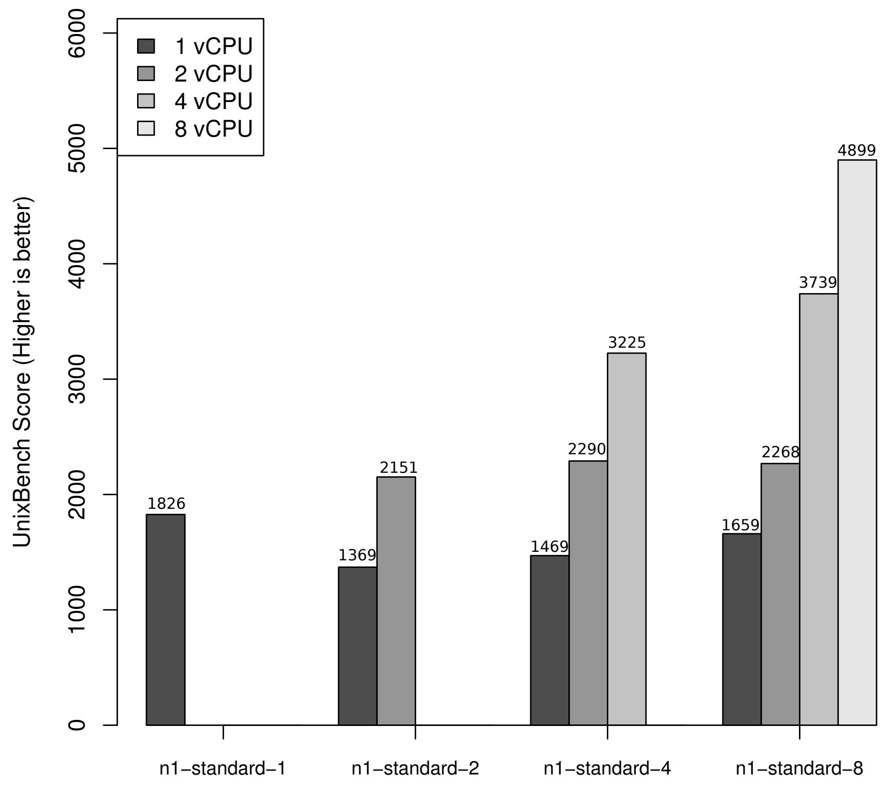
Table 9.5: Benchmark applications

Resource type	Application
CPU	LINPACK [227], UnixBench [57]
Network	iperf [168]

Considering these data, we simulated some users requirements, depicted in table 9.6, considering that they want to select instance types that maximize the amount of memory, the number of CPU cores (vCPU), the CPU performance (GFlops), and the network throughput with a minimal cost. Hence, in table 9.6 the requirements are defined as the



(a) Amazon EC2



(b) Google Compute Engine (GCE)

Figure 9.10: UnixBench score for one, two, four, and eight virtual cores for the general instance types of Amazon EC2 and GCE

minimal number of vCPUs and the amount of memory; and the maximal financial cost per hour, since these data are often known by the users. In this case, each row represents the users' requirements.

Table 9.6: Three users' requirements to select the clouds' resources

# vCPU	Memory (GB)	Cost (USD/hour)
4	4	0.5
16	8	1.0
16	90	2

9.7.1 Scenario 1: simple

The first scenario (First row in table 9.6), represents a small demand of the clouds, as the required resources are relatively small. For this case, there are 55 solutions (Figure 9.11) with the price ranging from 0.21 (*c3.xlarge*) to 0.49 (*m2.2xlarge*) dollars/hour, where 7 of them are in the Pareto front, as depicted in figure 9.12. Although the cheapest one meets the users' requirements, they can pay 34% more and get the double of memory and sustainable performance (m3.xlarge). However, the best solution, taking into account all the objectives, may depend on the characteristics of their applications or on other preferences. For example, fixing the cost, and requesting the system to based on the Pareto front to suggest one instance type, trying to maximize the other variables (vCPU, memory, GFlops, network throughput), the users still have two options (Figure 9.13). However, keeping the cost at the minimal value, after two iterations, they can select the *r3.xlarge* instance type paying 28% less than in the previous one (Figure 9.13) with the same sustainable performance and with a high amount of memory. Finally, if performance is the most important objective at the minimal cost, the *c3.2xlarge* is the instance to select.

9.7.2 Scenario 2: compute

The second scenario (Second row in table 9.6) simulates a CPU-bound demand due to the number of cores requested, which is the maximum offered by the GCE cloud (Table 9.4). In this case, there are 10 valid solutions, as shown in figure 9.14. However, only two different instance types (*n1-highcpu-16* and *c3.4xlarge*) met the objectives in different cloud providers (Figure 9.15). Hence, two of the solutions are in the Pareto front (Table 9.7). The *n1-highcpu-16* instance type can be selected if the network throughput is not important as the *c3.4xlarge* can offer a network throughput of 9.3 Gbps. However, the users must consider that to achieve this throughput, the instance must be restarted, which means that if it is needed in the first hour, a cost of 0.840 USD must be added for its usage. Otherwise the throughput is the same as the delivered by the other solution. Moreover, the *n1-highcpu-16* instance is also the best solution with regard to the minimal cost.

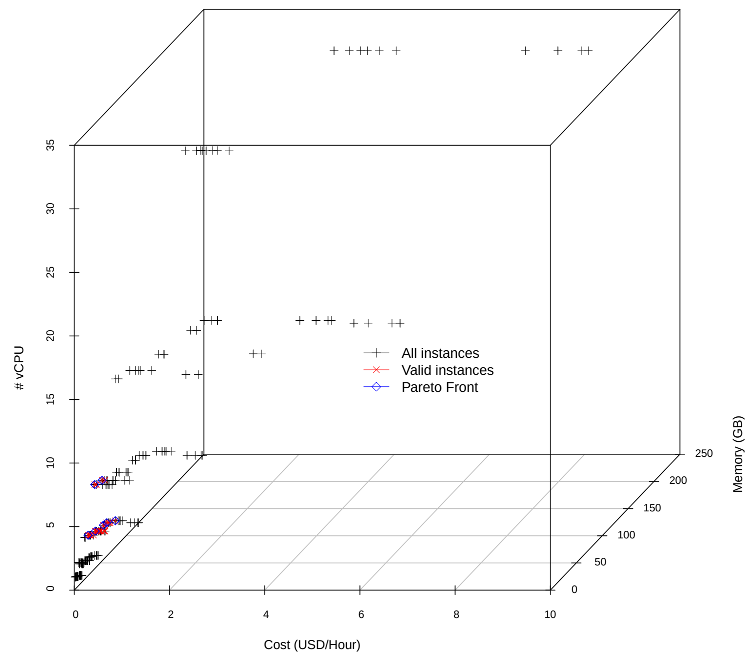
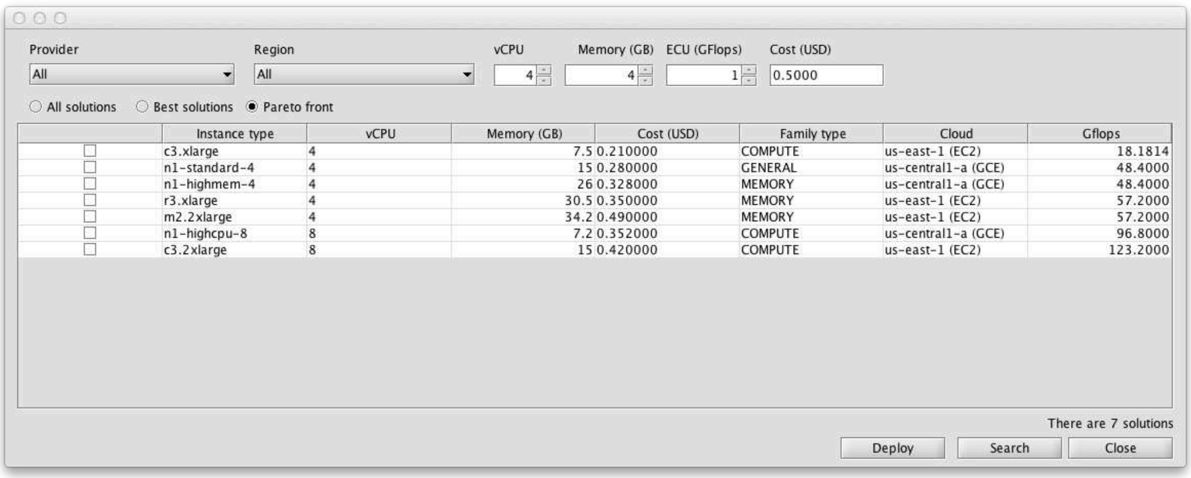


Figure 9.11: Instance types that offer at least 4 vCPU cores and 4GB of RAM memory with a cost of at most 0.5 USD/hour



Provider: All
Region: All
vCPU: 4
Memory (GB): 4
ECU (GFlops): 1
Cost (USD): 0.5000

( ) All solutions ( ) Best solutions [✓ Pareto front]

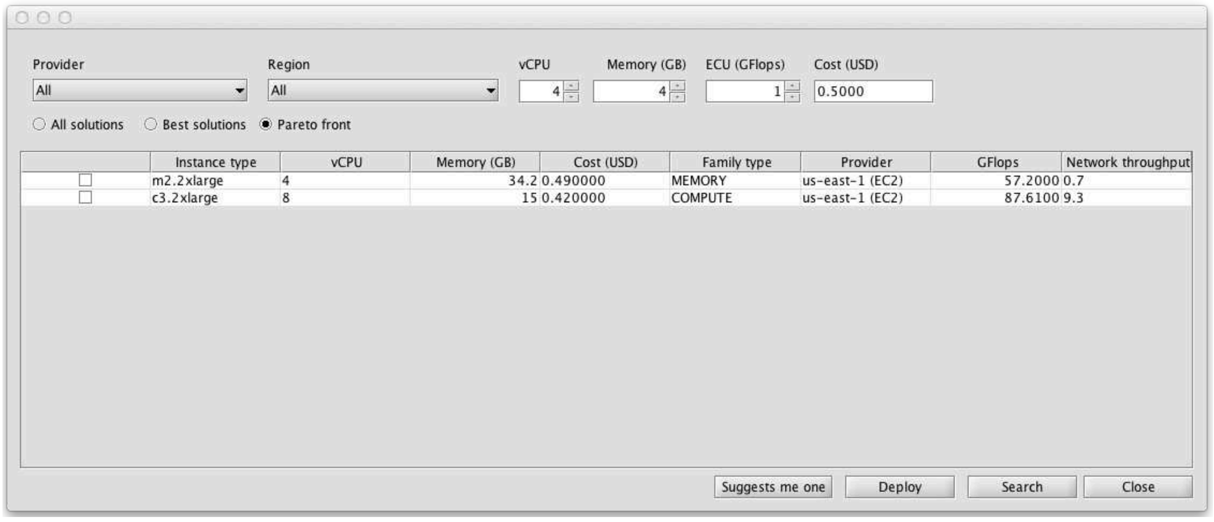
	Instance type	vCPU	Memory (GB)	Cost (USD)	Family type	Cloud	Gflops
☐	c3.xlarge	4	7.5	0.210000	COMPUTE	us-east-1 (EC2)	18.1814
☐	n1-standard-4	4	15	0.280000	GENERAL	us-central1-a (GCE)	48.4000
☐	n1-highmem-4	4	26	0.328000	MEMORY	us-central1-a (GCE)	48.4000
☐	r3.xlarge	4	30.5	0.350000	MEMORY	us-east-1 (EC2)	57.2000
☐	m2.2xlarge	4	34.2	0.490000	MEMORY	us-east-1 (EC2)	57.2000
☐	n1-highcpu-8	8	7.2	0.352000	COMPUTE	us-central1-a (GCE)	96.8000
☐	c3.2xlarge	8	15	0.420000	COMPUTE	us-east-1 (EC2)	123.2000

There are 7 solutions

Figure 9.12: Instance types in the Pareto front that offer at least 4 vCPU cores and 4GB of RAM memory with a cost of at most 0.5 USD/hour

Table 9.7: Instance types in the Pareto front considering a demand for 16 CPU cores and 8GB of RAM memory with a cost of at most 1.0 USD/hour

Instance type	Virtual cores	Memory (GB)	Cost (USD) / Hour	Family type	GFlops	Network throughput (Gbps)
n1-highcpu-16	16	14.4	0.704	Compute	81.6	0.8
c3.4xlarge	16	30	0.840	Compute	72.7	9.3



Provider: All
Region: All
vCPU: 4
Memory (GB): 4
ECU (GFlops): 1
Cost (USD): 0.5000

All solutions Best solutions [✓ Pareto front]

	Instance type	vCPU	Memory (GB)	Cost (USD)	Family type	Provider	GFlops	Network throughput
☐	m2.2xlarge	4	34.2	0.490000	MEMORY	us-east-1 (EC2)	57.2000	0.7
☐	c3.2xlarge	8	15	0.420000	COMPUTE	us-east-1 (EC2)	87.6100	9.3

Figure 9.13: Instance types suggested by the system that offer at least 4 vCPU cores and 4GB of RAM memory with a cost of at most 0.5 USD/hour

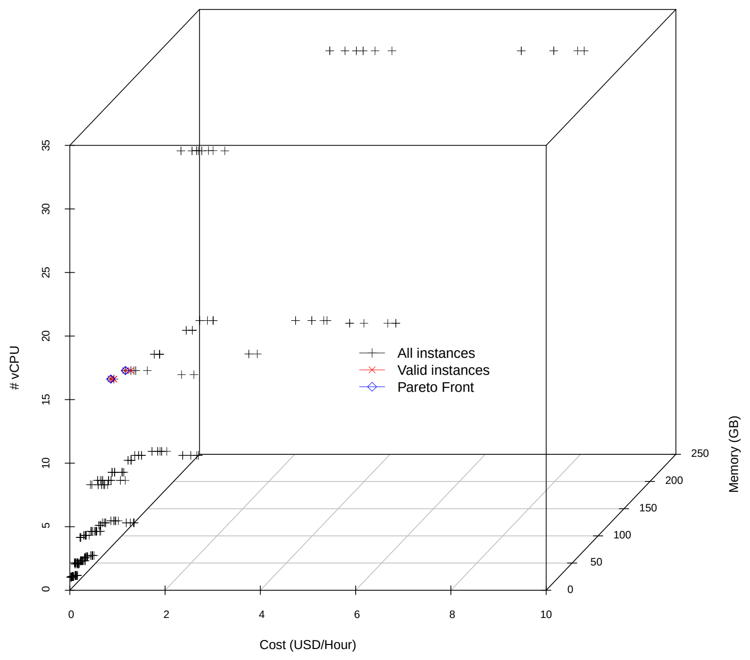
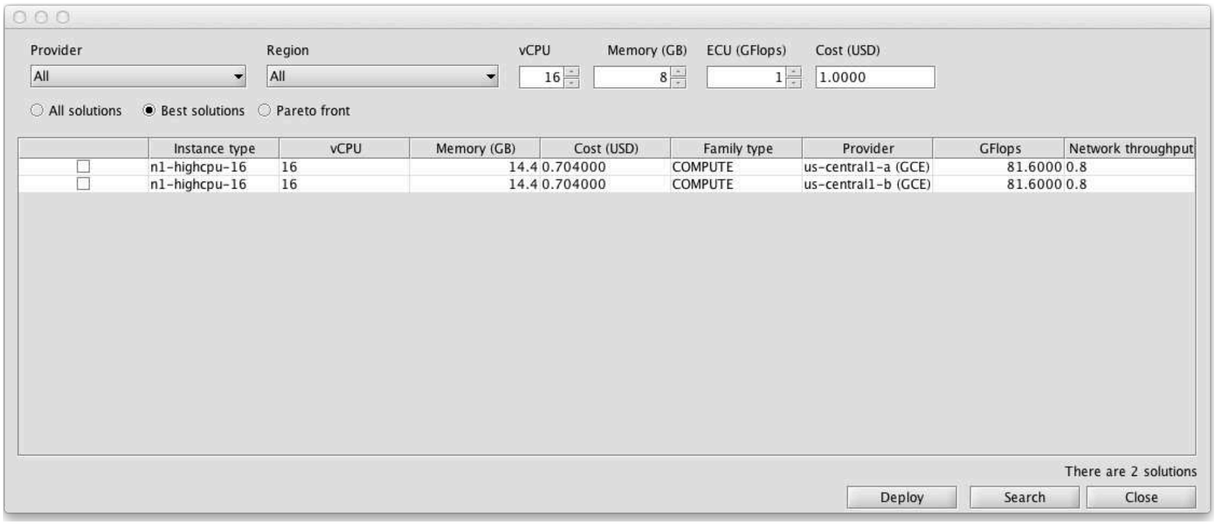


Figure 9.14: Instance types that offer at least 16 CPU cores and 8GB of RAM memory with a cost of at most 1.0 USD/hour



Provider: All
Region: All
vCPU: 16
Memory (GB): 8
ECU (GFlops): 1
Cost (USD): 1.0000

All solutions [✓ Best solutions] Pareto front

	Instance type	vCPU	Memory (GB)	Cost (USD)	Family type	Provider	GFlops	Network throughput
☐	n1-highcpu-16	16	14.4	0.704000	COMPUTE	us-central1-a (GCE)	81.6000	0.8
☐	n1-highcpu-16	16	14.4	0.704000	COMPUTE	us-central1-b (GCE)	81.6000	0.8

There are 2 solutions

Figure 9.15: The solutions with the minimal cost (best solutions) when requested at least 16 CPU cores and 8GB of RAM memory with a cost of at most 1.0 USD/hour

9.7.3 Scenario 3: compute and memory

The last scenario, on the other hand, simulates the need of a memory-bound application but that also requires a high number of CPU cores, i.e., a compute and memory demand. Similar to the previous scenario, there are only two different instance types (*n1-highmem-16* and *r3.4xlarge*) from the 13 valid solutions (Figure 9.16).

In this case, the best choice is the EC2's instance type (*r3.4xlarge*) as it costs only 6% more than the other instance type (*n1-highmem-16*) in the Pareto front (Figure 9.17) but, it has 17% more memory, a sustainable performance (GFlops) higher than the delivered by the other one (Table 9.8). In addition, it also has a network throughput of 9.3 Gbps. Without all this information, it may be difficult for the users deciding to use the *r3.4xlarge*, taking into account that they want low cost, and the description of both instances offers almost the same amount of resources (i.e., vCPU and RAM memory).

Table 9.8: Characteristics of the instance types that offer at least 16 CPU cores and 90GB of RAM memory

Instance type	**Virtual cores**	**Memory (GB)**	**Cost (USD) / Hour**	**Family type**	**GFlops**	**Network throughput (Gbps)**
n1-highmem-16	16	104	1.312	Memory	79.06	0.8
r3.4xlarge	16	122	1.400	Memory	138.86	9.3

9.8 Related Work

This section describes how cloud variabilities have been modeled in the literature and for which objective, as well as which cloud model has been considered. In order to highlight these objectives, the discussed works are organized in five categories: (i) modeling cloud

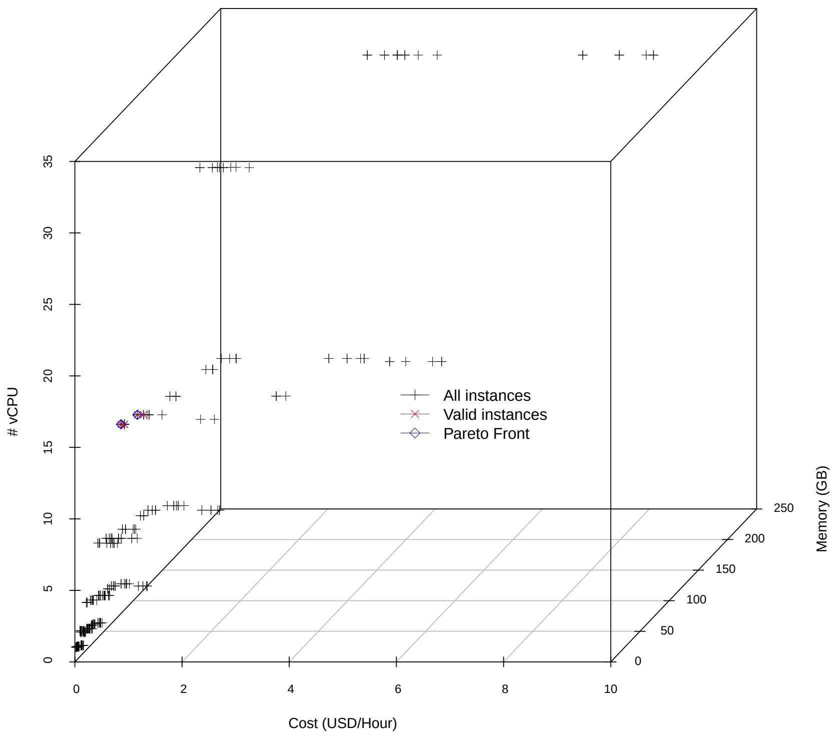
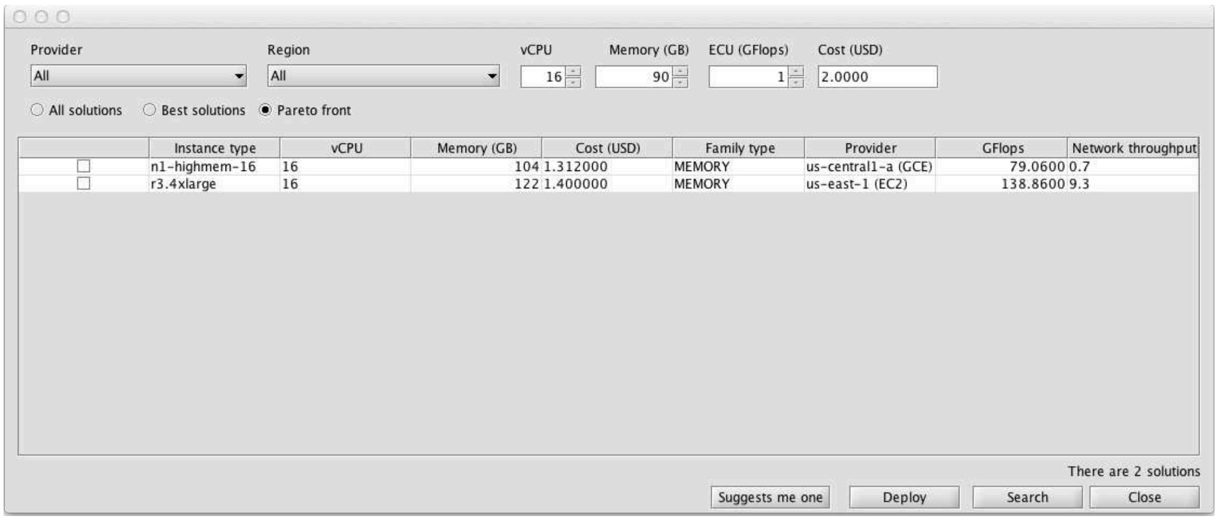


Figure 9.16: Instance types that offer at least 16 CPU cores and 90GB of RAM memory with a cost of at most 2 USD/hour



vCPU 16 | Memory (GB) 90 | ECU (GFlops) 1 | Cost (USD) 2.0000

[✓ Pareto front]

	Instance type	vCPU	Memory (GB)	Cost (USD)	Family type	Provider	GFlops	Network throughput
☐	n1-highmem-16	16	104	1.312000	MEMORY	us-central1-a (GCE)	79.0600	0.7
☐	r3.4xlarge	16	122	1.400000	MEMORY	us-east-1 (EC2)	138.8600	9.3

There are 2 solutions

Figure 9.17: Instance types in the Pareto front that offer at least 16 CPU cores and 90GB of RAM memory, with a cost of at most 2 USD/hour

variability with feature model, (ii) modeling machine image deployment, (iii) modeling PaaS applications, (iv) customization of multi-tenant applications, and (v) modeling infrastructure configurations.

9.8.1 Virtual Machine Image Configuration

9.8.1.1 SCORCH

Smart cloud optimization for resource configuration handling (SCORCH) [102] is a model-driven engineering (MDE) approach to manage virtual machine image (VMI) configurations. It aims to meet applications' performance taking into account the power consumption of the virtual machines. To achieve these goals, SCORCH uses feature modeling to describe the VMI and the applications' requirements. Furthermore, it employs constraint programming (CP) to find out the feasible configurations. In other words, SCORCH uses feature model to select a virtual machine in an auto-scaling queue that matches the applications requirements or that minimizes the time to setup one.

The proposed approach comprises five models (Figure 9.18): (i) cloud configuration model, (ii) configuration demand model, (iii) configuration adaptation model, (iv) energy model, and (v) cost model. The cloud configuration model is a feature model with the VMI configuration options such as CPU, memory, operating system, and application server. This model allows the developers to create a request for an instance configuration, and to check if it is valid. The configuration demand model specifies the instance type required by each application, and its required software packages. The configuration adaptation model, on the other hand, indicates the amount of time to add or to remove a feature from a configuration. The energy and cost models represent respectively the power consumption and the financial cost of each feature.

These models were used to deploy an e-commerce application considering three-family types (general, compute and memory) of Amazon EC2 and three provisioning scenarios: static, non-optimized auto-scaling, and optimized auto-scaling. The static scenario creates the virtual machines in advance and keeps them running all the time, whereas the non-optimized one creates them on-demand but without considering reconfiguration. The optimized scenario, on the other hand, extends the non-optimized to consider image reconfiguration. Using the models, the optimized provisioning strategy could reduce the power consumption and the response time of the applications.

9.8.1.2 VMI Provisioning

In [340], Tam and colleagues use MDE and feature modeling to configure and to deploy VMI. In this case, the proposed feature model, depicted in figure 9.19, describes the software packages required by a VMI and the constraints among them. In this case, the authors employ feature modeling to reduce the time to set up a virtual machine trying to match a user's configuration with one available in a repository.

Experiments realized in a testbed show that this approach could reduce the setup time and the network traffic. Moreover, using an extended feature model, this approach can reduce the power consumption of the virtual machines by removing redundant software [339].

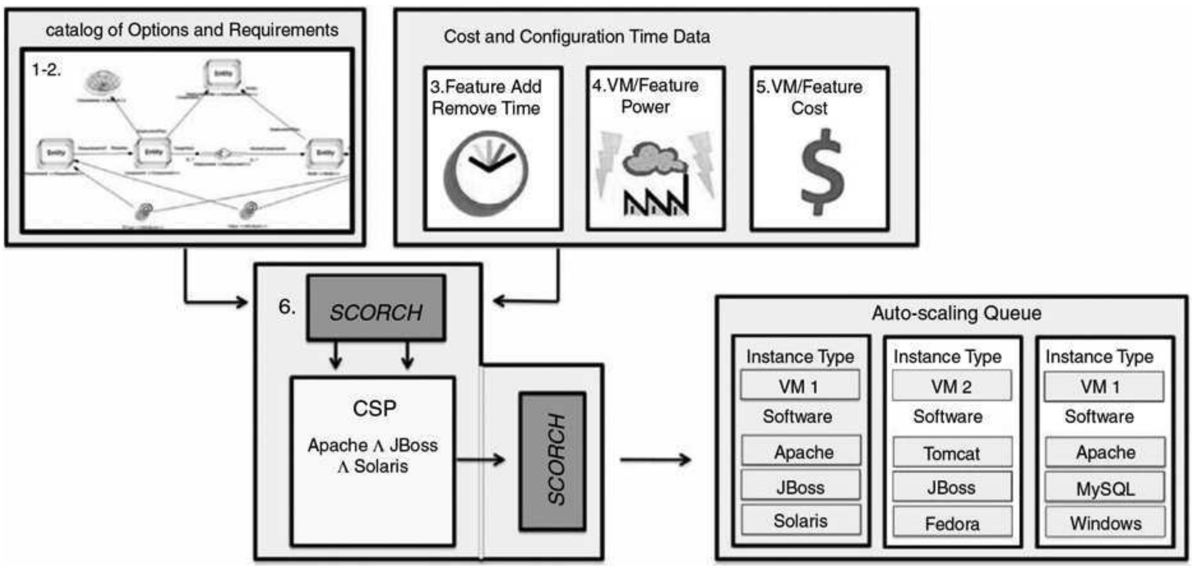
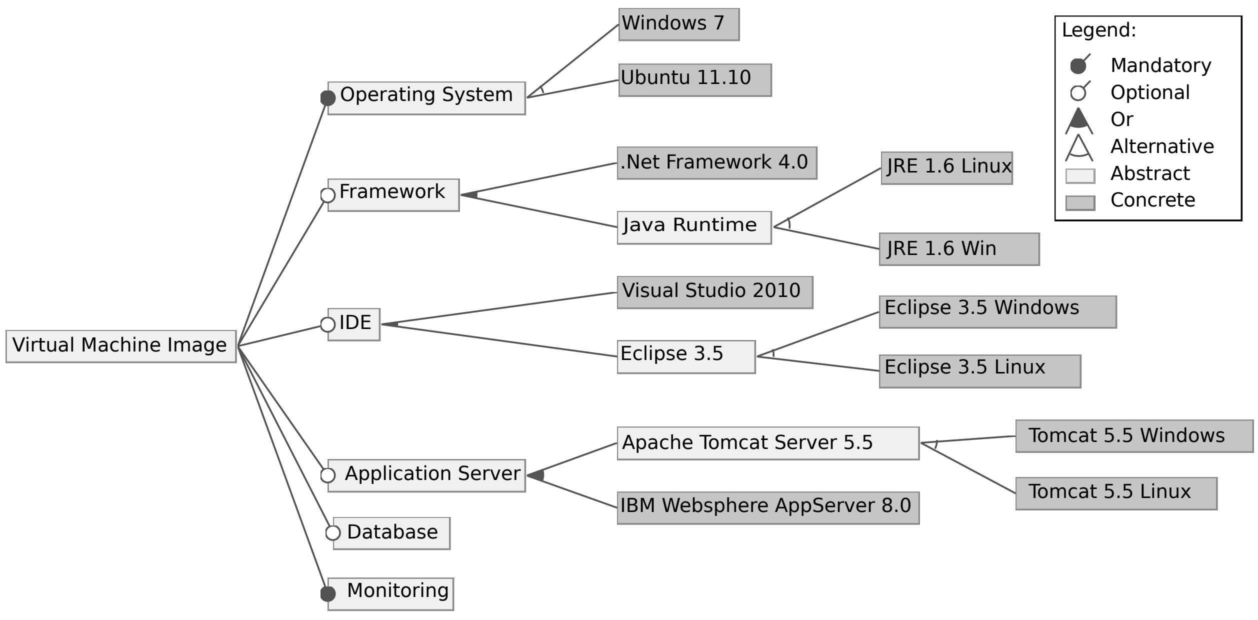


Figure 9.18: SCORCH MDE process [102]



Tomcat 5.5 Linux or Eclipse 3.5 Linux or JRE 1.6 Linux requires Ubuntu 11.10
Tomcat 5.5 Windows or Eclipse 3.5 Windows or JRE 1.6 Win or .Net Framework 4.0 requires Windows 7
Visual Sudio 2010 requires .Net Framework 4.0
Eclipse 3.5 requires Java Runtime

Figure 9.19: MDE approach for VMI configuration. Adapted from [340]

9.8.1.3 Typical Virtual Appliances

Zhang and colleagues [389] present a model called typical virtual appliance (TVA) for provisioning and management of virtual machines. A TVA is a template describing a set of popular softwares, and the frequently used services. The main objective of this approach is to minimize both storage space wasted with pre-built virtual machine images and the time to set up a virtual appliance. In this context, when a user requests a configuration that the system cannot match with one available in a repository, the system tries to select one that requires the minimal time to be converted to the requested configuration.

Simulation results show that this approach can reduce the setup time when there are sufficient data about popular software demands.

9.8.2 Virtual Machine Image Deployment

9.8.2.1 Virtual Appliance Model

In [200], Konstantinou and colleagues present an MDE approach for virtual machine image deployment using a virtual appliance model. A virtual appliance model is an abstract deployment plan that describes the configuration options of virtual machines. These configuration options are pre-built virtual machine images defined by domain experts. This approach, depicted in figure 9.20, works as follows. First, a solution architect creates a virtual solution model (VSM) by selecting virtual appliance models and specifying configuration options such as placement constraints, network zones, and resource consumption requirements. Then, a deployment architect transforms the VSM into a cloud-specific virtual solution deployment model (VSDM), where specific cloud configurations are added. Finally, the deployment model is converted into an executable plan called virtual solution deployment plan (VSDP). Then, the VSDP invokes the cloud's operations to configure the environment and the virtual machines based on the virtual solution model.

According to the authors, the idea of using these different models is first to capture the deployment logic of possible software service topologies and layers. Moreover, they enable a new kind of service called composition-as-a-service.

Experimental results show that using these models, a software architecture could deploy a virtual machine in a cloud.

9.8.2.2 Composite Appliance

Chieu and colleagues [78] introduce the concept of composite appliance to deploy applications in the cloud. A composite appliance is a set of virtual machine images configured to work as a single configuration option. In this case, the proposed model (Figure 9.21) has five classes that describe and capture the constraints of a virtual appliance such as memory size, number of CPUs and required software packages.

The proposed model aims to reduce the time to deploy a VM in the cloud by reusing the images configured for a deployment scenario.

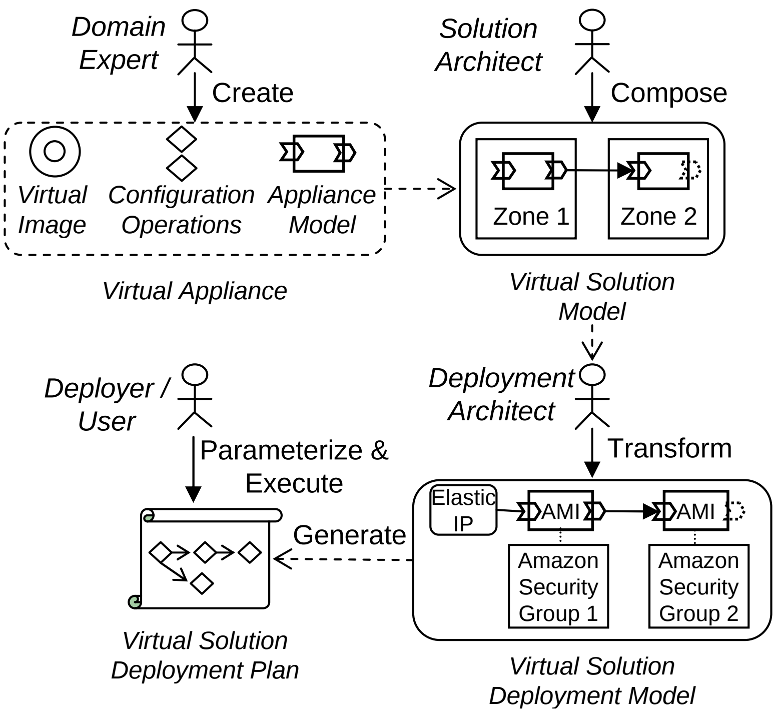


Figure 9.20: Virtual appliance model [200]

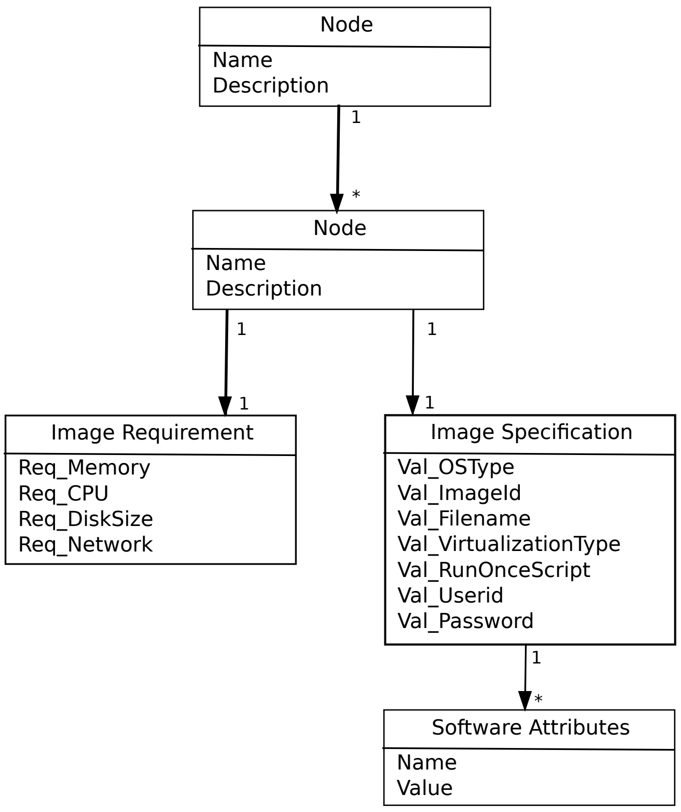


Figure 9.21: Composite appliance model [78]

9.8.3 Deploying PaaS Applications

9.8.3.1 HW-CSPL

Cavalcante and colleagues [9, 67] propose an adaptation of SPL-based development to deploy a Health Watcher application in the cloud (Figure 9.22). This approach works as following. First, the feature model of the application is modified to include cloud features such as storage and database type. Second, the developers change the source code of the application to implement cloud's variability. In such case, feature modeling helps the developers in identifying the clouds that meet the application's requirements, as well as to use the services of the clouds.

They validated this model using conditional compiling techniques and dynamic aspect-oriented programming (DAOP) to implement logging and database services offered by two different clouds.

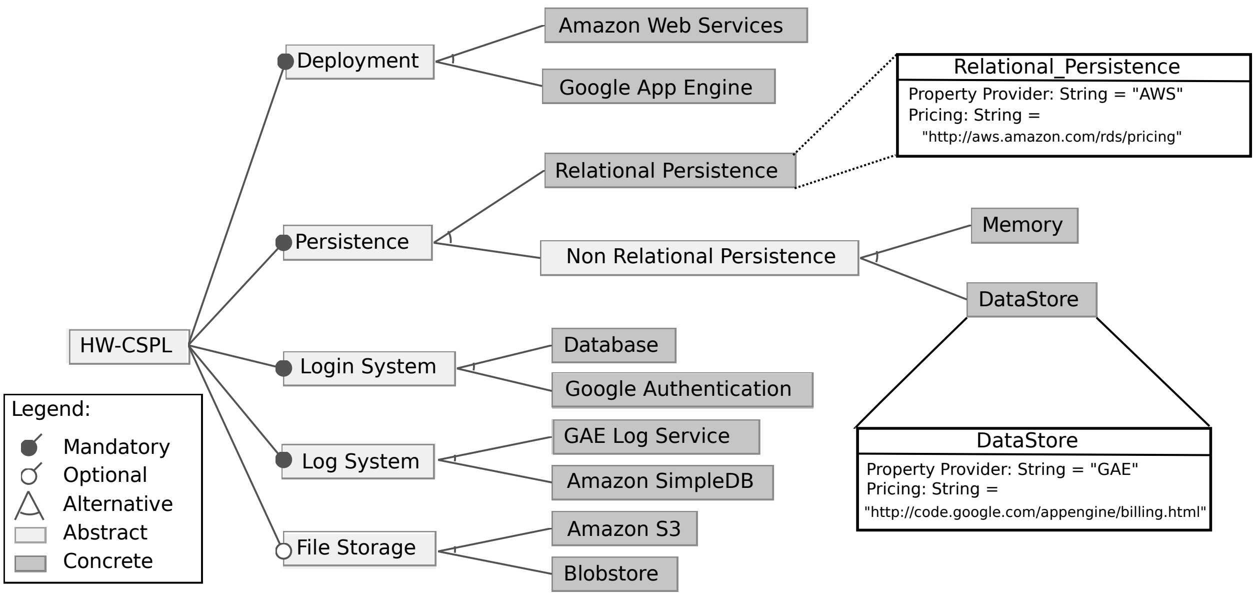
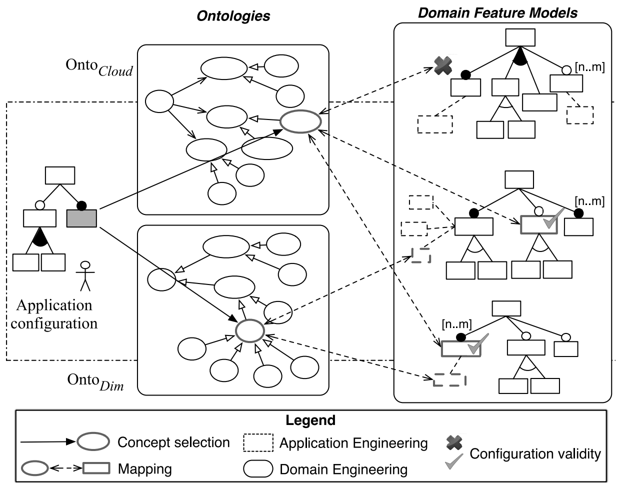


Figure 9.22: HW-CSPL's feature model. Adapted from [67]

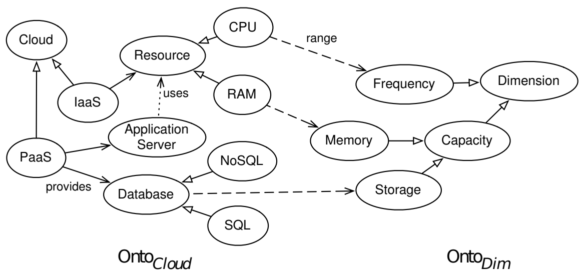
9.8.3.2 SALOON

SALOON [287] is a framework that combines feature models and ontology to support the developers on deploying their applications on multiple PaaS providers. In such case, it uses feature models to capture clouds variability, and ontologies to describe the features (Figure 9.23). SALOON has two ontologies, namely $Onto_{Cloud}$ and $Onto_{Dim}$. The $Onto_{Cloud}$ translates general cloud concepts to the terms used by each cloud provider, whereas the $Onto_{Dim}$ describes the attributes of the feature models. In other words, in this work, feature models handle cloud configurations, and ontologies translate the features and the attributes of each cloud model to general cloud concepts.

Experiments show that SALOON was able to select four different PaaS providers based on two configuration options of an application. Moreover, it could also generate the scripts to deploy the application in the clouds [288].



(a) SALOON approach



(b) SALOON's ontologies

Figure 9.23: SALOON framework [287]

9.8.4 Configuration options of multi-tenant applications

9.8.4.1 Multi-Tenant Deployment

Mietzner and colleagues [247] use variability modeling to support SaaS providers in managing configuration options of their applications. They consider two types of variability options: external and internal. On the one hand, external variability comprises the configuration options that (i) are visible and can be managed by the users (i.e., tenants), and (ii) do not change the application deployment such as changing the logo or the title of an application. On the other hand, internal variability concerns the infrastructure or deployment options that are only managed by the providers such as replication, clustering, and database sharding.

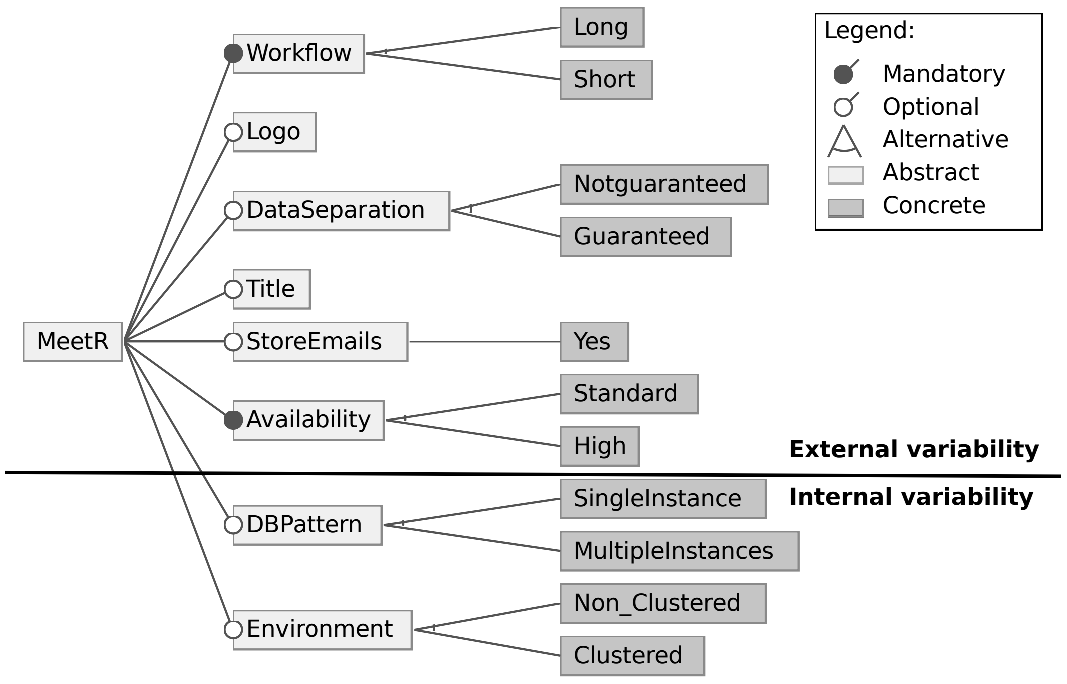
Figure 9.24 shows the feature model with external and internal options of the application modeled by the authors. In this case, feature modeling helped the users on changing the application's requirements (e.g., functional and non-functional) on-the-fly and supported the SaaS providers on generating the deployment scripts for each configuration.

9.8.4.2 Capturing Functional and Deployment Variability

Ruehl and colleagues [304, 305] use feature modeling to allow the users on customizing SaaS applications taking into account four different data-sharing models. Based on configuration templates, an execution engine adjusts a generic application to a configuration requested by a user[5].

In this context, a provider creates a catalog with the variability options supported by the applications, and the users utilize this catalog to deploy their applications (Figure 9.25). In such case, each variability option is implemented as a component that is instantiated according to a deployment model. The deployment model can be private, public, white hybrid, or black hybrid. In the private model, an application runs without sharing its

[5]In the context of this work, a user (i.e., tenant) may be a company and it may have multiple users.



Long ⇒ StoreEmails

Notguaranteed ⇒ DBPattern

Yes ⇒ DBPattern

High ⇒ Clustered

Guaranteed ⇒ MultipleInstances

Figure 9.24: Feature model showing external and internal variability options of a multi-tenant SaaS application. Adapted from [247]

components with other applications, whereas in the public model, it runs following the multi-tenant mode, i.e., multiple tenants use the same component. In the hybrid model, the tenants specify with whom they want to share their application's components.

Using variability modeling the users could select a configuration that met their security constraints and the providers could find a configuration option that reduced the number of single tenant-deployment.

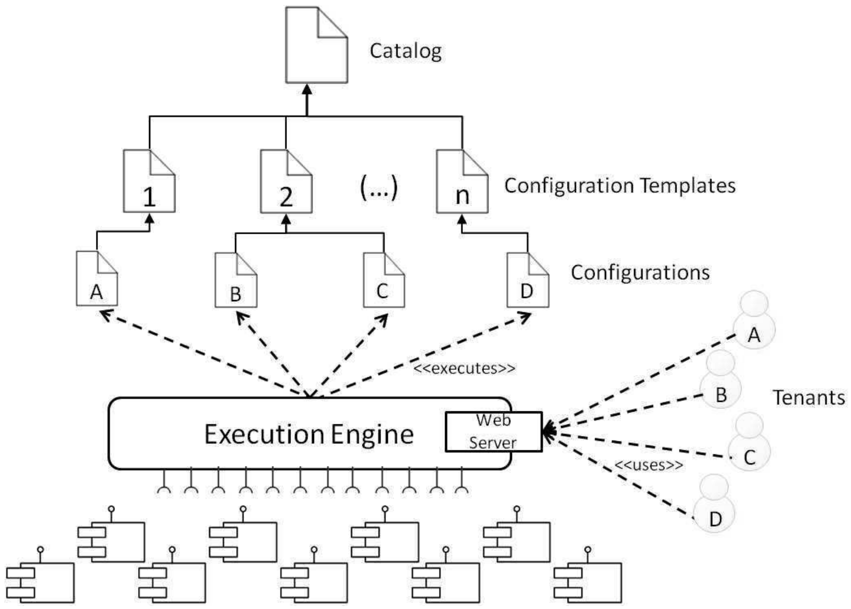


Figure 9.25: A model for managing configuration options of SaaS applications [305]

9.8.4.3 Configuration Management Process

Schroeter and colleagues [313] propose a configuration management process to deal with application requirements, different users views, and dynamic reconfiguration. The configuration process comprises an extended feature model (EFM), a view model, and a configuration process model. The proposed configuration process works as follows. First, extended feature models are used to express functional and non-functional requirements of an application. Then, the variability options are logically grouped according to the users (i.e., tenants) interests. In such case, a view model is created to support the mapping between the configuration decisions and the feature models, as well as to identify the users with similar concerns. Finally, the configuration process model defines the order of each configuration in four different stages as depicted in figure 9.26, where each stage represents a customer view. In the declaration stage, feature models are defined and their variability options are bounded in the specialization stage. Then, these feature models are merged in the integration stage. Moreover, the separation stage splits each feature model in multiple restricted extended feature models.

The objective of this process is to build consistent feature models and to guarantee valid configuration options.

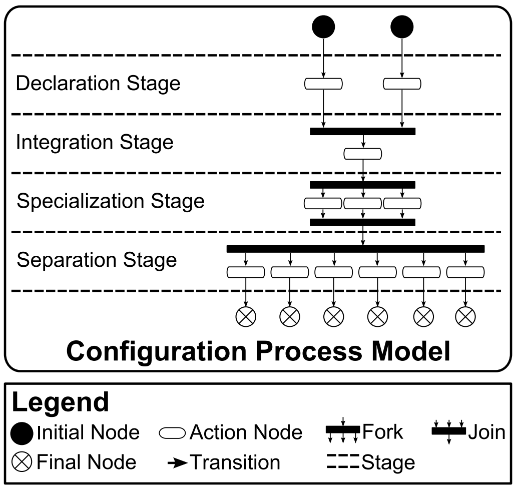


Figure 9.26: The configuration process model and its stages [313]

9.8.4.4 Service Line Engineering Process

In [363], Walraven and colleagues propose a software line engineering method to support configuration and deployment of multi-tenant applications. The method comprises four stages namely service line development, service line configuration, service line composition, and service line deployment (Figure 9.27). Each stage has different activities to support the users on creating the feature models and on developing the applications. This feature-oriented method aims to support the management and development of multi-tenant applications. In other words, this work focus on providing a method to handle the customization of multi-tenant applications in the cloud.

It was validated through the development of a document processing application and the results show that this method could reduce the costs to create and to operate the application.

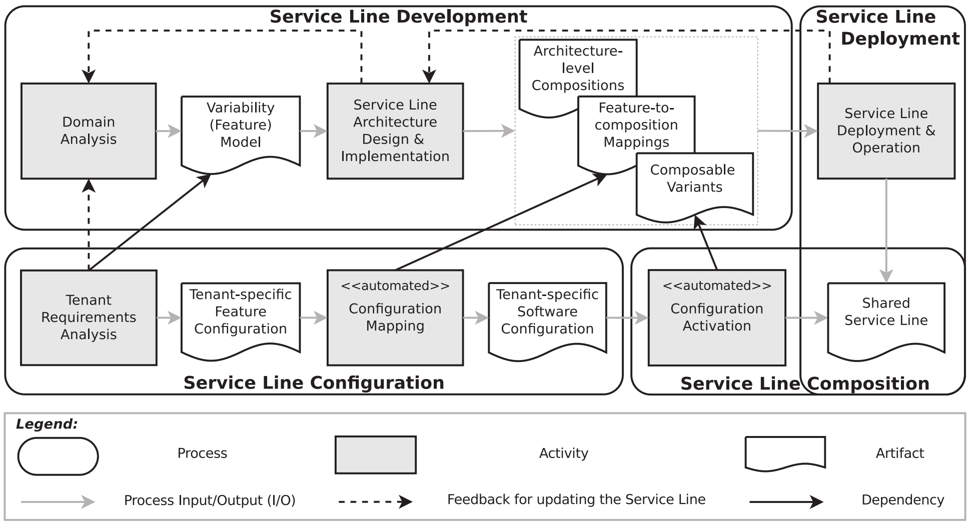

Service Line Development
Service Line Deployment
Domain Analysis
Variability (Feature) Model
Service Line Architecture Design & Implementation
Architecture-level Compositions
Feature-to-composition Mappings
Composable Variants
Service Line Deployment & Operation
Tenant Requirements Analysis
Tenant-specific Feature Configuration
<<automated>> Configuration Mapping
Tenant-specific Software Configuration
<<automated>> Configuration Activation
Shared Service Line
Service Line Configuration
Service Line Composition
Legend:
Process
Activity
Artifact
Process Input/Output (I/O)
Feedback for updating the Service Line
Dependency

Figure 9.27: The service line engineering method [363]

9.8.5 Infrastructure Configuration

9.8.5.1 AWS EC2 Service Provisioning

García-Galán and colleagues [130] uses an extended feature model (EFM) to describe some AWS services. They aim to support the users on selecting valid service configurations basing on high-level constraints such as the number of CPUs, the minimum memory or storage size, and the maximum financial cost (Figure 9.28). Functional requirements (e.g. instance type) were modeled as features and non-functional requirements (e.g., cost and usage) as attributes.

The proposed approach comprises two phases. The first-phase validates if a user's configuration meets the constraints of a model; and the second one selects the configurations that fulfill a given objective. These phases were validated using a customized version of the FaMa tool[6] [348] to show the users which Amazon EC2 configurations fulfill the given objective.

9.8.6 Comparative View

Table 9.9 summarizes the papers reviewed in this section. The first column shows the paper or the name of the solution presented in the paper. Then, the second column presents how feature modeling is used to deal with the clouds' variabilities. The cloud service model considered by each paper is described in the third column. The last column, on the other hand, presents the objectives of each paper.

In the domain of cloud computing, feature modeling has also been used to handle the variabilities of VMI, multi-tenant applications, and services offered by the cloud providers.

Most of the reviewed works [78, 102, 200, 340, 389] handle the variabilities of virtual machine images. In this case, they aim to minimize the time to setup a virtual machine considering performance constraints and in some case, power consumption [102, 339]. Feature modeling has also used to reduce storage space [389] used with pre-built images or to support the users on deploying a virtual machine in the cloud.

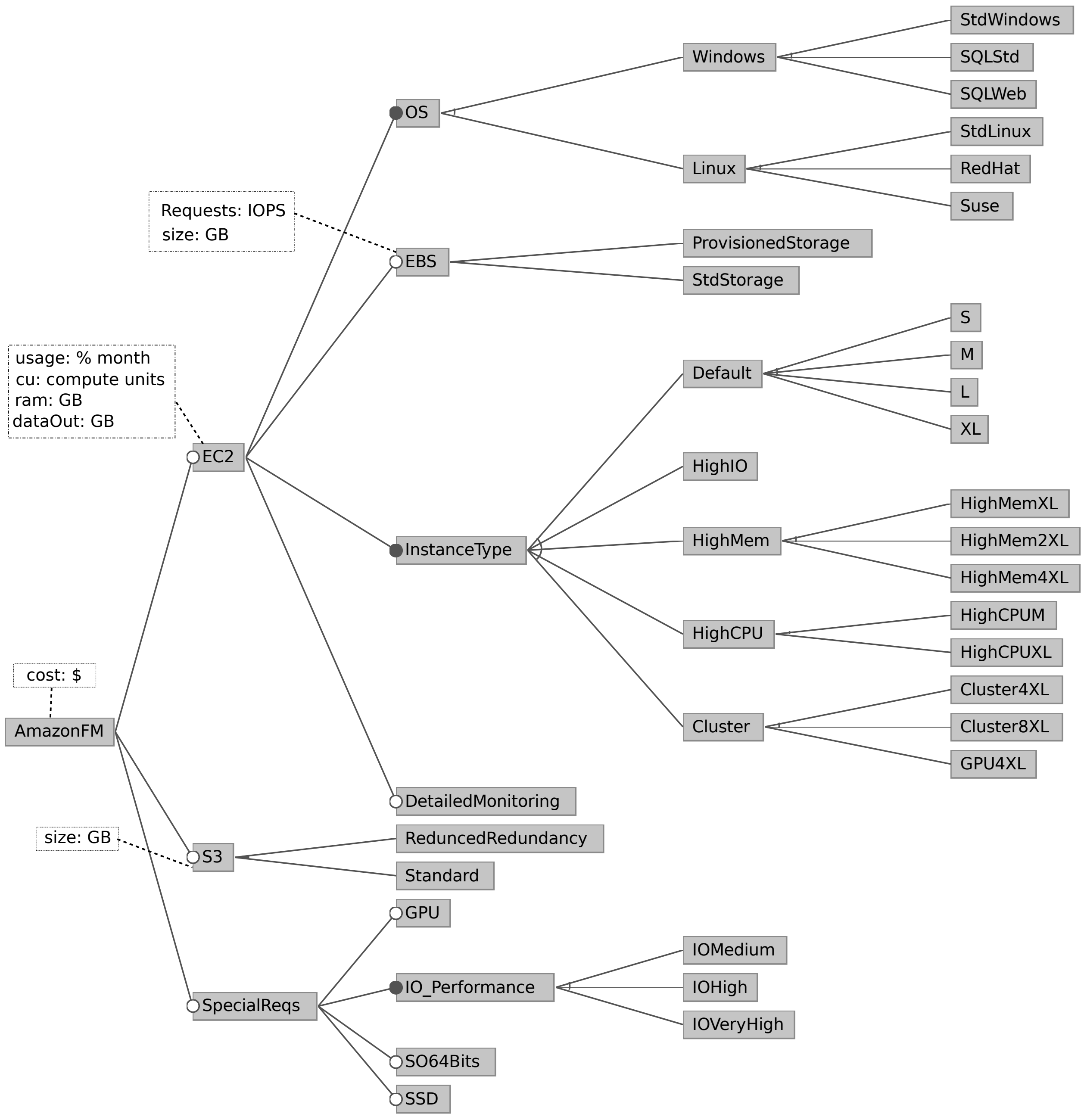
At the PaaS layer, one solution [9, 67] uses feature modeling to help the developers on changing the code of an application to use cloud's services; and another solution [287, 288] uses it to support the developers on deploying their native cloud applications.

The variabilities of SaaS applications have also be considered by some works [247, 304, 305, 313, 363] with different objectives. In such case, they aim to avoid invalid configurations, to minimize the time to configure or to create an application, and to avoid single tenant deploys.

There is one work that considers resource selection [363] in an specific cloud. In this case, based on the users' requirements, a feature model is used to obtain the configurations of the cloud and to check that a user's configuration is valid.

Our work handles the variabilities at the infrastructure layer, and similar to [363], we use feature modeling to select the resources according to the users' requirements. However, we consider multiple users' objectives and multiple clouds, employing an abstract feature model. This abstract model allows us to select different resources like virtual

[6] FaMa is a tool to analyze feature models.



RedHat ⇒ ¬ Cluster

SQLStd ⇒ XL ∨ GPU4XL ∨ HighCPUM

Figure 9.28: Extended feature model for EC2, EBS, and S3 services. Adapted from [130]

machine image, disks, and to describe the whole environment. Moreover, it avoids invalid configurations and it can deploy the configurations in the clouds.

Table 9.9: Comparative view of cloud variability models

Paper	**Usage of feature model**	**Cloud model**	**Multi cloud**	**Objectives**
SCORCH [102]	VMI configuration	IaaS	No	Minimize power consumption taking into account performance constraints
[339, 340]	VMI configuration	IaaS	No	Minimize the time to setup a VM and power consumption
[389]	VMI configuration	IaaS	No	Minimize storage space with pre-built VMI and the time to setup a VM
[200]	VMI deployment	IaaS	No	Minimize the time to deploy a VM
[78]	VMI deployment	IaaS	No	Minimize the time to deploy an application
HW-CSPL [9, 67]	Development of a cloud-aware application	PaaS	Yes	Minimize the time to change an application to use clouds' services
SALOON [287, 288]	Deployment of cloud-aware applications	PaaS	Yes	Minimize the time/effort to deploy native cloud applications
[247]	CMMTA*	SaaS	No	Minimize the time to configure an application
[304, 305]	CMMTA*	SaaS	No	Minimize the number of single-tenant deploys taking into account users' security constraints
[313]	CMMTA*	SaaS	No	Minimize inconsistent feature models
[363]	CMMTA*	SaaS	No	Minimize the time to create a cloud-aware application
[130]	Instance selection	IaaS	No	Minimize invalid configurations considering a single objective
This work	IS and IC$^{\sharp}$	IaaS	Yes	Avoid invalid configurations and select the instances taking into account multiple objectives

* Configuration management of multi-tenant applications (CMMTA)

$^{\sharp}$ Instance selection and infrastructure configuration

9.9 Summary

In this chapter, we presented our solution to help the users on selecting the clouds and the instance types to execute their applications, taking into account multiple objectives. The proposed solution is based on a product line engineering (PLE) process that comprises different phases and roles (Figure 9.1). In the first phase, an abstract feature model (FM) is created to describe the variabilities of the IaaS clouds. Next, this model is instantiated with the features and characteristics (i.e., qualitative and quantitative attributes) of the products available in the clouds. After, the users specify their requirements and objectives, and the configurations that are optimal with regard to the objectives are selected and deployed in the clouds.

We validated our model employing constraint programming (CP) implemented through the Choco [178], a well-established satisfaction problem solver, considering two different cloud providers and multiple objectives. The experimental results show that with our approach, the users can get optimal configurations with regard to their objectives, which without the support of a system it may be difficult for users deciding for the suggested ones.

Our solution deals with the challenges identified in section 9.2 as follows. To face the first challenge (*capturing clouds heterogeneity and constraints*), we proposed a cloud model and an abstract feature model that specify the characteristics and constraints of the clouds. The product line engineering approach deals with the second challenge (*matching application requirements with the resources characteristics*) enabling resource selection based on a declarative strategy and on the users' objectives. In this case, the engineers benchmark the clouds and include the results in the models. We used our abstract model to handle the configuration options of two different clouds (Figures 9.8 and 9.9), helping us to deal with the third challenge (*describing a multi-cloud deployment environment*). These models consider the dependencies between the resources in a multi-cloud environment. Finally, as our models have functional and non-functional data about the clouds, it can support auto-scaling systems without the usage of virtual machine images, dealing with the fourth challenge (*supporting auto-scaling decisions*).

Our solution can be used to implement a control loop, where a system continuously monitors the environment and takes decisions in order to meet the users' objectives. In other words, it can be used to support a system that aims to implement autonomic properties such as self-configuration and self-optimization.

Chapter 10

Dohko: An Autonomic and Goal-Oriented System for Federated Clouds

Contents

In the previous chapter, we described proposed a method based on software product line that allowed automated configuration and deployment of the resources in the cloud, enabling a generative approach. In this chapter, we use this method to build an autonomic and goal-oriented cloud system. This system aims to help the users on executing their applications on federated IaaS clouds. Moreover, our system tackles the issues identified in the second contribution (Chapter 7) of this doctoral thesis, relying on top of a P2P layer. Finally, it meets most of the functional requirements identified for multiple cloud systems such as [277]: (a) it provides a way to describe functional and non-functional requirements through the usage of our software product line engineering method described in the previous chapter; (b) it can aggregate services from different clouds; (c) it provides a homogeneous interface to access services of multiple clouds; (d) it allows the service selection of the clouds; (e) it can deploy its components on multiple clouds; (f) it provides automatic procedures for deployments; (g) it utilizes an overlay network to connect and to organize the resources; (h) it does not impose constraints for the connected clouds.

The reminder of this chapter is organized as follows. Section 10.1 presents the introduction and motivation behind this chapter. Section 10.2 presents the proposed system architecture and its main components, followed by a description of its autonomic properties. Experimental results are presented and discussed in 10.3. Section 10.4 presents a comparative view of some important features of cloud architectures. Finally, section 10.5 concludes this chapter.

10.1 Introduction and Motivation

Deploying and executing an application in an IaaS cloud demand a considerable amount of work. This amount of work is mostly due to the kind of applications considered by these clouds. While the clouds focus on Web applications, the users' applications are many times batch-oriented, performing parameter sweep operations. Furthermore, users applications may require specific configurations and some of them may take days or even weeks to complete their executions.

While the former issue can be dealt with virtual machine image (VMI), the latter requires monitoring and fault-tolerance strategies in order to reduce the chances of losing the work (i.e., the application execution) just before it completes. Nonetheless, creating VMIs is often time-consuming and demands advanced technical skills. Monitoring, on the other hand, is usually realized by employing heartbeats [153]. Heartbeats are presence notification messages sent by each virtual machine to a monitor [88]. In this case, when a heartbeat message fails to reach the monitor within a given threshold, the VM is considered unavailable and a process to recovery it or to create a new one starts.

Since clouds are susceptible to failures, it is interesting to consider a multi-cloud scenario. This scenario increases the difficulties to deploy and to execute an application. These difficulties are mainly due to the clouds' heterogeneity and because of the cost of exporting and importing virtual machine images between the clouds. In addition, the users should be aware of the clouds' internals and the communication pattern of their applications in order to reduce the cost, for instance. For example, in some clouds, the cost of data transfers between virtual machines in the same data center using their external IP addresses is the same as the cost of transferring data between different clouds or regions. Moreover, the network throughput is often lower when using external IP addresses. Hence, changing the applications to consider the location of the nodes or realizing this work manually may be very difficult for the users, especially when there are multiple clouds and the environment can change at runtime. In this context, we advocate the use of autonomic systems (Chapter 4) to do this work automatically.

Therefore, in this chapter we propose and evaluate an autonomic cloud architecture. The proposed architecture implements self-configuration, self-healing, and context-awareness properties to help users on executing their applications in multiple clouds and to tackle the difficulties described previously. Our architecture relies on a hierarchical P2P overlay to organize the virtual machines and to deal with inter-cloud communication.

The reminder of this chapter is organized as follows. Section 10.2 presents the proposed architecture and its main components, followed by a description of its autonomic properties. Then, experimental results are discussed in 10.3. Next, a comparative view of some important features of cloud architectures is presented in section 10.4. Finally, section 10.5 presents final considerations.

10.2 System Architecture

In our proposal, we assume a multi-cloud environment, where each cloud has a public API that implements the operations to: (a) manage the virtual machines (e.g., to create, to start, to shutdown) and their disks; (b) get the list of the available VMs and their state (e.g., running, stopped, terminated, among others); (c) complement the VMs' descriptions through the usage of metadata. We also consider that the virtual machines fail following the fail-stop model. In other words, when a VM fails, its state changes to stopped and the system can either restart it or replace it by a new one.

As the proposed architecture organizes the nodes using a P2P overlay, we assume that there is a set of nodes (e.g., super-peers) responsible for executing some specialized functions such as system bootstrapping and communication management. It is important to notice that the presence of such nodes does not mean a centralized control, as described in section 2.4. In addition, the number of these specialized nodes may vary according to some objective, e.g., the number of clouds.

The architecture uses a message queue system (MQS) to implement a publish/subscribe policy. The publish/subscribe interaction pattern is employed due to its properties such as [108]: space decoupling, time decoupling, and synchronization decoupling. These properties help the system to increase scalability and to make its communication infrastructure ready to work on distributed environments. Moreover, message queue systems normally do not impose any constraint for the infrastructure. In this case, all messages are persisted to support both node's and services' failures; and a message continues in the queue until a subscriber consumes it or until a defined timeout is achieved.

The proposed architecture comprises three layers: *client*, *core*, and *infrastructure*, as depicted in figure 10.1. There is also a cross-layer called *monitoring*. In the context of this work, the rationale for using a layered architecture is that it enables a loosely coupled design and a service-oriented architecture (SOA).

10.2.1 Client Layer

The *client* layer provides the *job submission* module. This module takes an *application descriptor* as input, and submits it to a node in the cloud. This node is called *application*

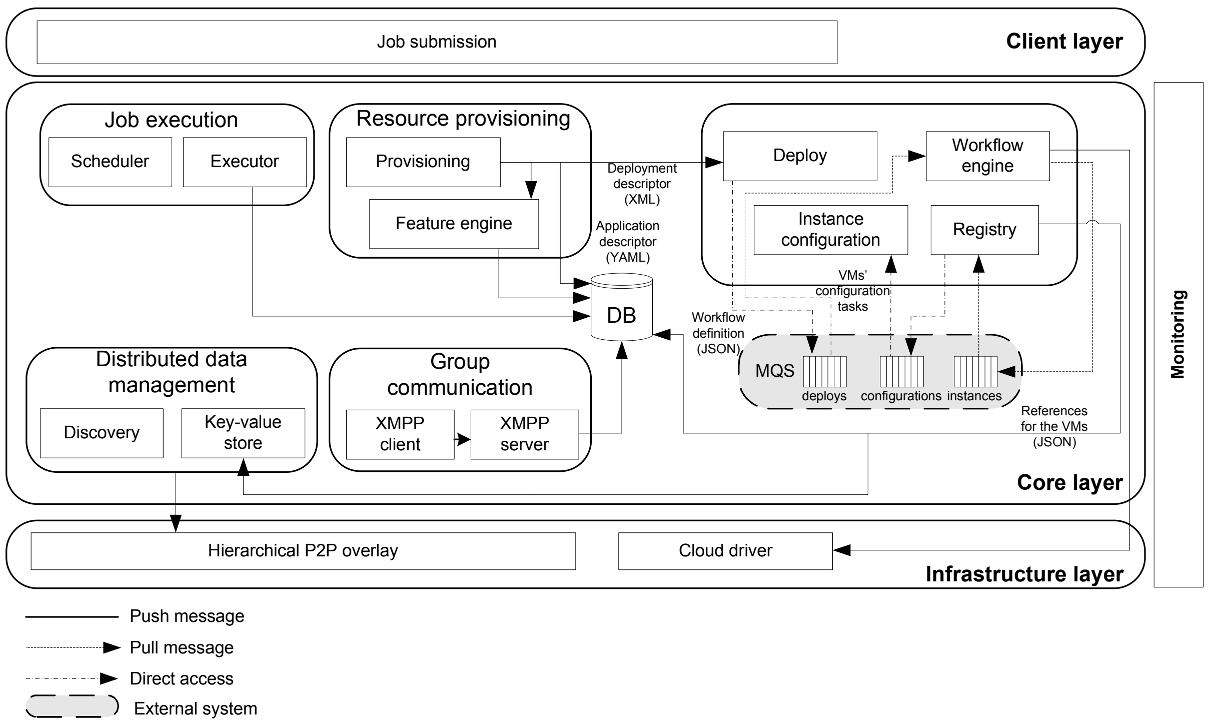


Figure 10.1: Design and main modules of an autonomic architecture for multiple clouds

manager, and it will coordinate the executions in the cloud.

An *application descriptor* has five parts (Listing 10.1): user, requirements, clouds, applications, and on-finish action. The user section contains the user's information such as user name and his/her SSH keys. If the user does not have the SSH keys, they will be generated by the system. These keys are used to create and to connect to the virtual machines. The requirements section, on the other hand, includes: (a) the maximal cost to pay for a VM per hour; (b) the minimal number of CPU cores and RAM memory size; (c) the platform (i.e., operating system); and (d) the number of virtual machines to be created in each cloud. These requirements are used to select the instance types based on the product line engineering (PLE) approach described in chapter 9. The clouds section comprises the data of the user in each cloud provider. These data consist of the access and the secret key required to invoke the clouds' operations, and they are given to users by the cloud providers. This section also contains informations such as region and instance types. However, these parameters target advanced users (e.g., system administrators) who may already have the regions and/or the instance types to be used. The applications section describes the tasks to be executed including their inputs and outputs. Finally, the on-finish (Line 42 in listing 10.1) part instructs the architecture about what to do after the applications have finished. The options are: NONE, FINISH, and TERMINATE. The FINISH option shuts down the virtual machines, which means that their persistent disks continue in the clouds; whereas the TERMINATE option shuts down and deletes the virtual machines.

```
---
name:

user:
  username:
  keys:
    key: []

requirements:
  cpu:
  memory:
  platform:
  cost:
  number-of-instances-per-cloud:

clouds:
  cloud:
  - name:
    provider:
      name:
    access-key:
      access-key:
      secret-key:
    region:
    - name:
      zone:
      - name:
    instance-types:
      instance-type:
      - name:
    number-of-instances:

applications:
  application:
    name:
    command-line:
    file:
    - name:
      path:
      generated:

on-finish:
```

Listing 10.1: Structure of an application descriptor

10.2.2 Core Layer

The *core* layer of the architecture (Figure 10.1) comprises the modules to provision, to create, to configure, to implement group communication, and to manage data distribution, as well as to execute the applications.

The *provisioning* module is responsible for receiving an *application descriptor* and for generating a *deployment descriptor*. It utilizes the *feature engine* to obtain the instance types that meet the users' requirements. The *feature engine* implements the feature model

(FM) described in chapter 9. The *deployment descriptor* comprises all data required to create a virtual machine. These data are: (a) the cloud provider; (b) the instance type; (c) the zone (data center); (d) the virtual machine image; (e) the network security group and its inbound and outbound traffic rules; (f) the disks; and (g) the metadata. Listing 10.2 shows an example of an *deployment descriptor*. In this example, one virtual machine should be created with the name *e8e4b9711d36f4fb9814fa49b74f1b724-1* in the zone *us-east-1a* of region *us-east-1*. The cloud provider should be Amazon, using the instance type *t2.micro* and the virtual machine image *ami-864d84ee*. Furthermore, two metadata (tags) are added to the virtual machine (app-deployment-id and manager).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<deployment user="username">
    <uuid>f4070433-a551-4a60-ba57-3e771ceb145f</uuid>
    <node name="e8e4b9711d36f4fb9814fa49b74f1b724-1" count="1" region="us-east-1"
          zone="us-east-1a">
        <provider name="amazon">
            <image>ami-864d84ee</image>
            <instance-type>t2.micro</instance-type>
        </provider>
        <tags>
            <tag>
                <name>app-deployment-id</name>
                <value>e8e4b9711d36f4fb9814fa49b74f1b724</value>
            </tag>
            <tag>
                <name>manager</name>
                <value>i-a1f8798c</value>
            </tag>
        </tags>
    </node>
</deployment>
```

Listing 10.2: An example of a deployment descriptor generated by the provisioning module

The *deployment descriptor* is sent to the *deploy* module. Then, the *deploy* module creates a workflow (*deployment workflow*) with the tasks to instantiate the VMs. A workflow is used since there are some precedent steps that must be performed in order to guarantee the creation of the VMs. For example, (a) the SSH keys must be generated and imported into the clouds; (b) if the instance types support the *group* feature (Figure 9.6), the system must check if one exists in the given zone and if not, create it. Furthermore, using workflow enables the architecture to support partial failures of the deployment process. In addition, it decouples the architecture from the clouds' drivers. In this case, based on the provider's name, the *deploy* module selects in the database its correspondent driver to be instantiated at runtime by the *workflow engine*. This *deployment workflow* is enqueued by the *deploy* module and dequeued by the *workflow engine*. The *workflow engine* executes the workflow and enqueues the references to the created virtual machines in another queue.

The *registry* module is responsible for: (a) storing the instances informations in a local database and in the distributed key-value store; and (b) selecting the configurations (i.e., software packages) to be applied in each instance. These configurations are usually defined by the product engineers (Chapter 9), and they represent a set of scripts. Each script may have some variability points. The variability points are informations only known at runtime such as the location (i.e., zone), the name, the addresses (e.g., internal and external), among others, that must be set by the system. Listing 10.3 illustrates one script with three variability points. A variability point is defined between ${[and]}. In this example, the script is exporting tp the system the region's name of a node, its endpoint, and its zone's name.

```
...
 export NODE_REGION_NAME=${[location.region.name]}
   export NODE_REGION_ENDPOINT=${[location.region.endpoint]}
   export NODE_ZONE_NAME=${[location.name]}
...
```

Listing 10.3: Example of one script with three variability points

The configuration tasks are put in a queue to be executed by the *instance configuration* module. The *instance configuration* module connects to the virtual machine via SSH and executes all the scripts, installing the software packages. This process guarantees that all instances have the required software packages.

The *group communication* module uses the Extensible Messaging and Presence Protocol (XMPP) (xmpp.org) for instant communication among the nodes in a cloud. It has an *XMPP server* and an *XMPP client*. The architecture uses the XMPP since some other group communication techniques such as broadcast or multicast are often disabled by the cloud provider. In addition, it supports the *monitoring* layer through its presence states (e.g., available, off-line, busy, among others).

The *job execution* module comprises a *scheduler* and an *executor*. They are responsible for implementing a task allocation policy and for executing the tasks. The task allocation policy determines in which node a task will be executed, considering that there are no temporal precedence relations among the tasks [63]. The goal of a task allocation/scheduling problem may be: (a) minimize the execution time of each task; (b) minimize the execution time of the whole application; (c) maximize the throughput (number of tasks completed per period of time). On its generic formulation, this problem has been proved NP-complete [141]. For this reason, several heuristics have been proposed to solve it. By default, this architecture provides an implementation of a simple task allocation policy (i.e., self-scheduling (SS) [342]). However, other task allocation policies can be implemented and added to our architecture, i.e., this architecture does not assume any specific task allocation policy.

The self-scheduling (SS) policy assumes a master/slave organization, where there is a master node responsible for allocating the tasks, and several slaves nodes that execute the tasks. Moreover, it considers that very few information is available about the execution time of the tasks and the computing power of the nodes. In this case, it distributes the tasks, one by one, as they are required by the slave nodes. Thus, each node always receives one task, executes it, and, when the execution finishes, asks for more task [342].

In our architecture, the node that receives the *application descriptor* becomes the application master node, and it is the responsible for creating and coordinating the configuration of the other nodes (e.g., slave nodes).

The *distributed data management* module provides a *resource discovery* and a distributed *key-value store* service. These services are based on the *hierarchical P2P* (Section 2.4.5) overlay available at the *infrastructure* layer (Section 10.2.3). Table 10.1 shows the operations implemented by the *key-value store*. Basically, the key-value store providers three operations: (i) insert, (ii) retrieve, and (iii) remove. The insert operation takes a key and a value, and stores the value under the given key. In case there exists a value with the same key, all of them they are stored. In other words, the value of a key may be a set of objects. The retrieve method receives a key and returns its values. Finally, the remove method can remove all values of a given key or only one specific value.

Table 10.1: Main operations implemented by the key-value store

Operation	Description
void **insert** (key, value)	inserts the value into the network with the given key. If two or more values exist with the same key, all of them are stored
Set<Value> **retrieve** (key)	returns all the values with the given key
remove (key)	removes all the values stored under the given key
void **remove** (key, value)	removes the value stored under the given key

10.2.3 Infrastructure Layer

The *infrastructure* layer consists of the *hierarchical P2P* overlay and the *cloud drivers*. The *hierarchical P2P* overlay is used to connect the clouds and their nodes. In this case, in a cloud with n nodes, where $n > 0$, $n - 1$ nodes (i.e., leaf-nodes) join an internal overlay network and one node (super-peer) joins the external overlay network. In other words, there is one overlay network connecting the clouds and another overlay network in each cloud connecting its nodes. The super-peer and its leaf-nodes communicate through a HTTP service, and the leaf-nodes monitor the super-peer via XMPP. Both overlays are implemented using the Chord [332] protocol presented in section 2.4.3. This solution is an extension of a previous work that was published in [216]. In [216], there is a *P2P* module that implements P2P functions such as placement, search, event, among others, decoupled from both system and protocol used. Figure 10.2 illustrates the hierarchical P2P overlays connecting two clouds, each one with four nodes.

When a node n running in cloud c starts, it asks the bootstrapping node through a HTTP service, the super-peers of cloud c. If node n is the first peer of cloud c, it joins the super-peers overlay by the bootstrapping node. Otherwise, it demands the super-peer, its leaf-nodes and joins the leaf-nodes overlay network or creates a overlay network. After has joined one overlay network, the node stores in the key-value store its information under the keys: $/c/n$, if it is the super-peer or $/c/ <$ super-peer's id $>$ /members, otherwise.

Leaf-nodes of different clouds communicate through their super-peers. In this case, when a leaf-node wants to communicate with a node outside its cloud, it sends a message for its super-peer that first connects to the super-peer of such node, and next forwards the message for it.

When a node leaves one cloud, it notifies its super-peer that removes the information about the node from the system.

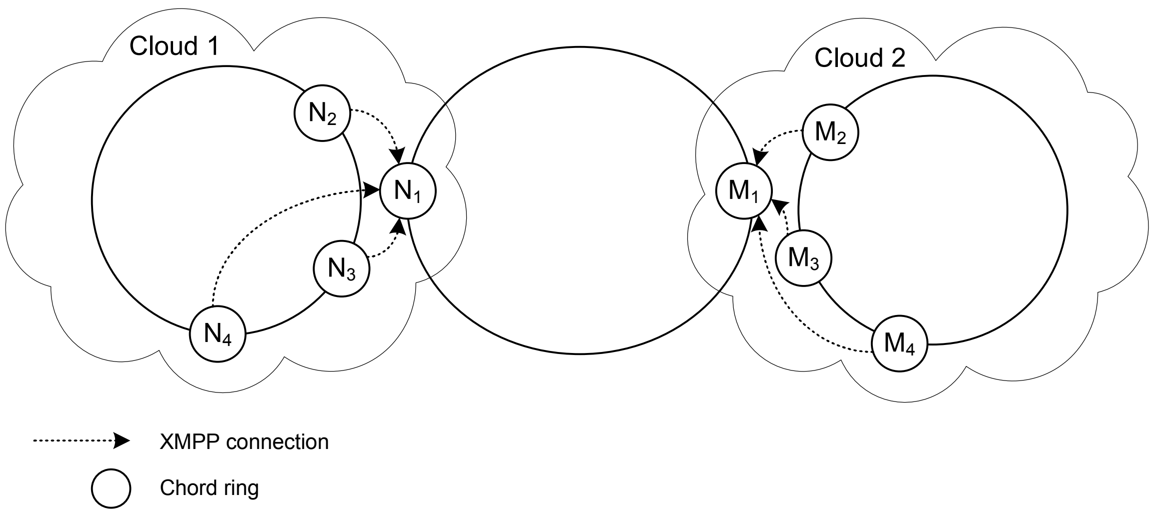


Figure 10.2: Structure of the hierarchical P2P overlay connecting two clouds

10.2.4 Monitoring Cross-Layer

The *monitoring* layer is responsible for (a) checking if there are virtual machines that were not configured; (b) detecting and restarting failure nodes; (c) keeping up-to-date the information about the super-peers and the leaf-nodes in the system.

10.2.5 Autonomic Properties

This section presents how the proposed architecture implements the following autonomic properties: self-configuration, self-healing, and context-awareness. These autonomic properties were presented in section 4.2. Although there exists other autonomic properties, as described in chapter 4, our architecture implements only these properties since our focus is to help the users to tackle the difficulties of deploying and executing an application in the clouds, taking into account different objectives and without requiring cloud and system administration skills from the users. Figure 10.3 shows the autonomic control loop implemented by this architecture.

The process follows a declarative strategy. A declarative strategy allows the users to concentrate on their objectives rather than on dealing with cloud or system administration issues. In this case, the process starts with the users describing their applications and constraints. Then, using a self-configuration process, the system (a) creates and configures the whole computing environment taking into account the characteristics and the state of the environment, i.e., the availability of other virtual machines; (b) monitors the availability and state of the nodes through the self-healing; (c) connects the nodes taking into account their location. In other words, using a hierarchical organization, nodes in the same cloud joins an internal overlay network and one of them joins an external overlay network, connecting the clouds (Figure 10.2). Nodes in the internal overlay network use internal IP addresses for communication, which often has zero cost and a network throughput higher than if they were using external IP addresses; finally (d) executes the applications.

In the next sections, we will explain each one of these properties in detail.

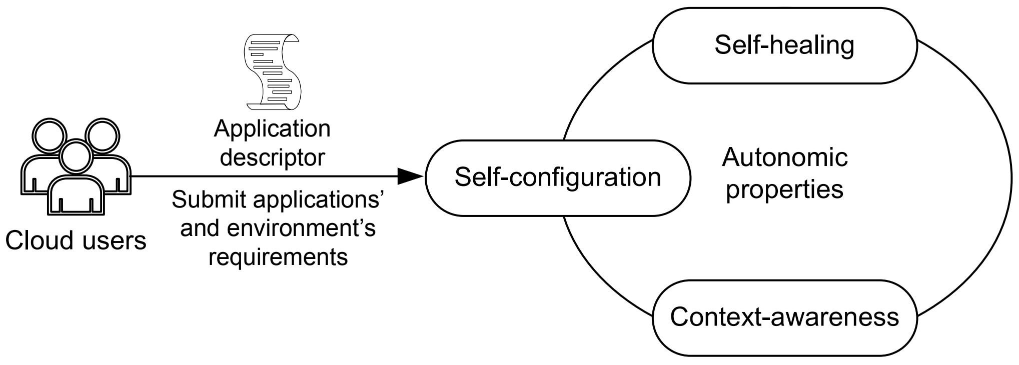


Figure 10.3: The autonomic properties implemented by our architecture

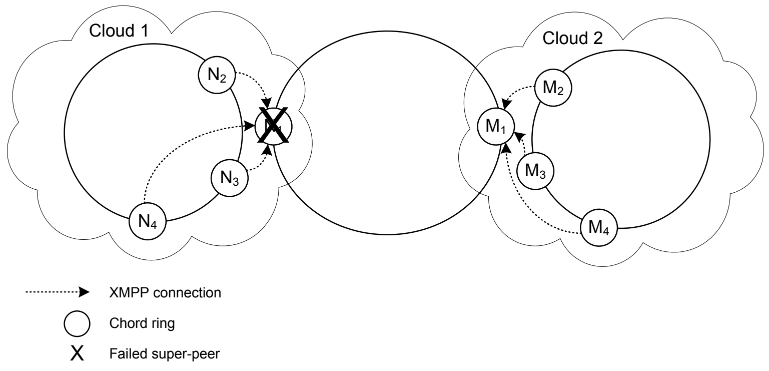
10.2.5.1 Self-Configuration

Our architecture automatically creates and configures the virtual machines in the clouds. The configuration process is based on the concrete feature model defined by the product engineers (Section 9.6) and on the users requirements. When the architecture receives an *application descriptor*, it (a) creates the SSH keys (i.e., private and public keys) to be used by the virtual machines, and imports them into the clouds; (b) creates a security group with the inbound and outbound rules; (c) uses the clouds' metadata support to describe the VMs, allowing the users to trace the origin of each resource, and to access them without using the architecture, if necessary. In addition, these metadata provide support for the self-healing process; (d) selects one data center (i.e., availability zone) to deploy the virtual machines according to the availability of VMs running in the same zone; (e) selects the instance types; (f) configures the instances with all the software packages; (g) starts all the required services considering the instance state. For example, if a service requires one information (e.g., its region's name, up-to-date IP addresses) about the environment where it is running, it is automatically assigned by the architecture before it starts. For instance, when an instance fails or needs to be restarted, the architecture detects the new values of its ephemeral properties such as the IP addresses (i.e., private and public IPs) and starts its services to use up-to-date informations.

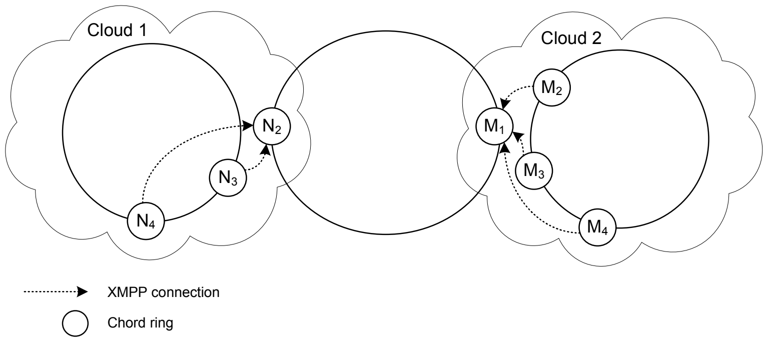
The metadata of an instance include: (i) the user name to access the VM, (ii) the value of the keys, (iii) the name of its virtual machine image (VMI), (iv) the owner (i.e., the user), (v) the name of its application manager, and (vi) the name of the feature model used to configure it. These data help the system to support failures of both the manager and the architecture. For example, suppose that just after having created the virtual machines, the application manager node fails. Without these metadata, another node could not access the instances to configure them, leaving for the system only the option to terminate the instances, which implies a financial cost as the users will pay for the instances without having used them.

10.2.5.2 Self-Healing

The cross-layer *monitor* module uses the XMPP to monitor the availability of the virtual machines. In this case, every virtual machine runs an XMPP server, which periodically, requests the *discovery* service to list the VMs running in the cloud. The returned VMs are included as the monitor contact. In this context, each VM utilizes the XMPP's status to report to other one its status and also to know their status. When a VM's status changes to off-line, its application manager waits a determined time and requests the cloud provider to restart it. The restarting avoid unresponsive node due to its workload state (i.e., overloaded VM) or some network issues. If the node's status does not change to on-line, the manager terminates it, and creates a new one. In case of the super-peer's failures (Figure 10.4(a)), the leaf-node with the higher uptime tries to recover the failed super-peer. If not possible, this leaf-node leaves its network (Figure 10.4(b)) and joins the super-peers network, becoming the new super-peer.



(a) The cloud's 1 super-peer (N_1) failed disconnecting the clouds 1 and 2



(b) The leaf-node N_2 leaves its overlay network and joins the new super-peer overlay network, connecting the clouds 1 and 2

Figure 10.4: Example of super-peer failure and definition of a new super-peer

Since the application manager may completely fail during the configuration of the virtual machines, the *monitor* checks if there are instances created by the architecture without have been completely configured. If such instances do exist, it contacts their application manager or sends them for other node to continue its configuration.

10.2.5.3 Context-Awareness

Similar to [222] and [17], by context we refer to the topology of the overlay network or peers association, which may impact the overall system's performance (e.g., throughput and latency), as well as in the financial cost (e.g., network communication cost). In other words, in the context of this work, context-awareness means the capacity of the P2P communication protocol be aware of the peers' locations and of adapting the system's behavior based on the situation changes [222, 380].

As described in section 10.2, the nodes are organized into two overlay networks. This avoids unnecessary data transfers between the clouds and often increases the network throughput between the nodes in the internal overlay. Furthermore, it helps to decrease the cost of the application execution, avoiding inter-cloud data transferring due to the overlay stabilization activities (Section 2.4.3). If all nodes were organized in the same P2P overlay network, they would have to communicate using external IP addresses (i.e., public IP), which implies in Internet traffic cost, even if the nodes are located in the same data center. In addition, in a high churn rate scenario (Section 2.4.2), it decreases the cost of maintaining the distributed hash tables (DHTs) (Section 2.4.3) up-to-date since it does not require inter-cloud communication.

10.2.6 Executing an Application in the Architecture

In order to illustrate the usage of our architecture, consider that one user is interested in executing his/her application in the cloud (Figure 10.5). In this example, the application manager is the node that receives the user's demand (*application descriptor*), and the worker is a node that receives a task to execute.

The process starts when the user defines an *application descriptor* depicted in listing 10.4, with the requirements and applications. In this example, one virtual machine with at least 1 CPU core and 1 GB of RAM memory is requested, with the Linux operating system, and a cost of at most 0.02 USD/hour. Moreover, this instance should be created on Amazon using the given access and secret key. Finally, the task consists of getting the information about the CPU (Line 25 in listing 10.4).

The *application descriptor* is submitted to the system through the *job submission* module (Figure 10.5 (1)). Then, the *job submission* module looks for an appropriate node through the *discovery* service (Figure 10.5 (2)), and sends the *application descriptor* to it (Figure 10.5 (3)).

After, the *provisioning* module in the application manager takes the *application descriptor* and persists into its database (Figure 10.5 (4)). Next, the *provisioning* module demands the *feature engine* module: (a) an instance type that meets the user's constraints, and (b) a virtual machine image (VMI). The *feature engine* returns the *t2.micro* (Table 9.3) instance type in the *us-east-1* region, and the virtual machine image *ami-864d84ee*. With these data, the *provisioning* module: (a) selects a zone to host the virtual machine, (b) generates the *deployment descriptor* (Listing 10.2), and (c) submits it to the *deployment* module (Figure 10.5 (5)).

The *deployment* module creates a workflow (Figure 10.6) with the steps to instantiate the VM, and enqueues it through the MQS (Figure 10.5 (6), queue: deploys). The *workflow engine* executes the deployment workflow, i.e., it connects to the cloud and creates the virtual machine (Figure 10.5 (7 and 8)). The data about the VM are enqueued in *instances* queue (Figure 10.5 (9)). Next, the *registry* module dequeues the instance from the queue *instances* and inserts its information into the database and into the *key-value store* (Figure 10.5 (10)). After, it creates the configuration tasks to be executed in the virtual machine (Figure 10.5 (11)). Each task comprises a host, a user name, the SSH keys, and the scripts to be executed. After, the *instance configuration* module: (a) connects to the node via SSH; (b) executes the scripts; and (c) starts an instance of this architecture (Figure 10.5 (12)). Then, the virtual machine, which is now executing the architecture, notifies the application manager using the *group communication* module (Figure 10.5 (13)). Finally, the manager uses the *scheduler* (Figure 10.5 (14)) to distribute the tasks, according to the self-scheduling (SS) task allocation policy (i.e., self-scheduling (SS) [342]). In other words, the *scheduler* sends a task to the worker, that should execute it and return its result to the application manager. The whole process is monitored by the *monitor* module.

```yaml
---
name: "example"
user:
  username: "user"
requirements:
  cpu: 1
  memory: 1
  platform: "LINUX"
  cost: 0.02
  number-of-instances-per-cloud: 1
clouds:
  cloud:
  - name: "ec2"
    provider:
      name: "amazon"
    access-key:
      access-key: "65AA31A0E92741A2"
      secret-key: "619770ECE1D5492886D80B44E3AA2970"
    region: []
    instance-types:
      instance-type: []
applications:
  application:
    name: "cpuinfo"
    command-line: "cat /proc/cpuinfo"
on-finished: "NONE"
```

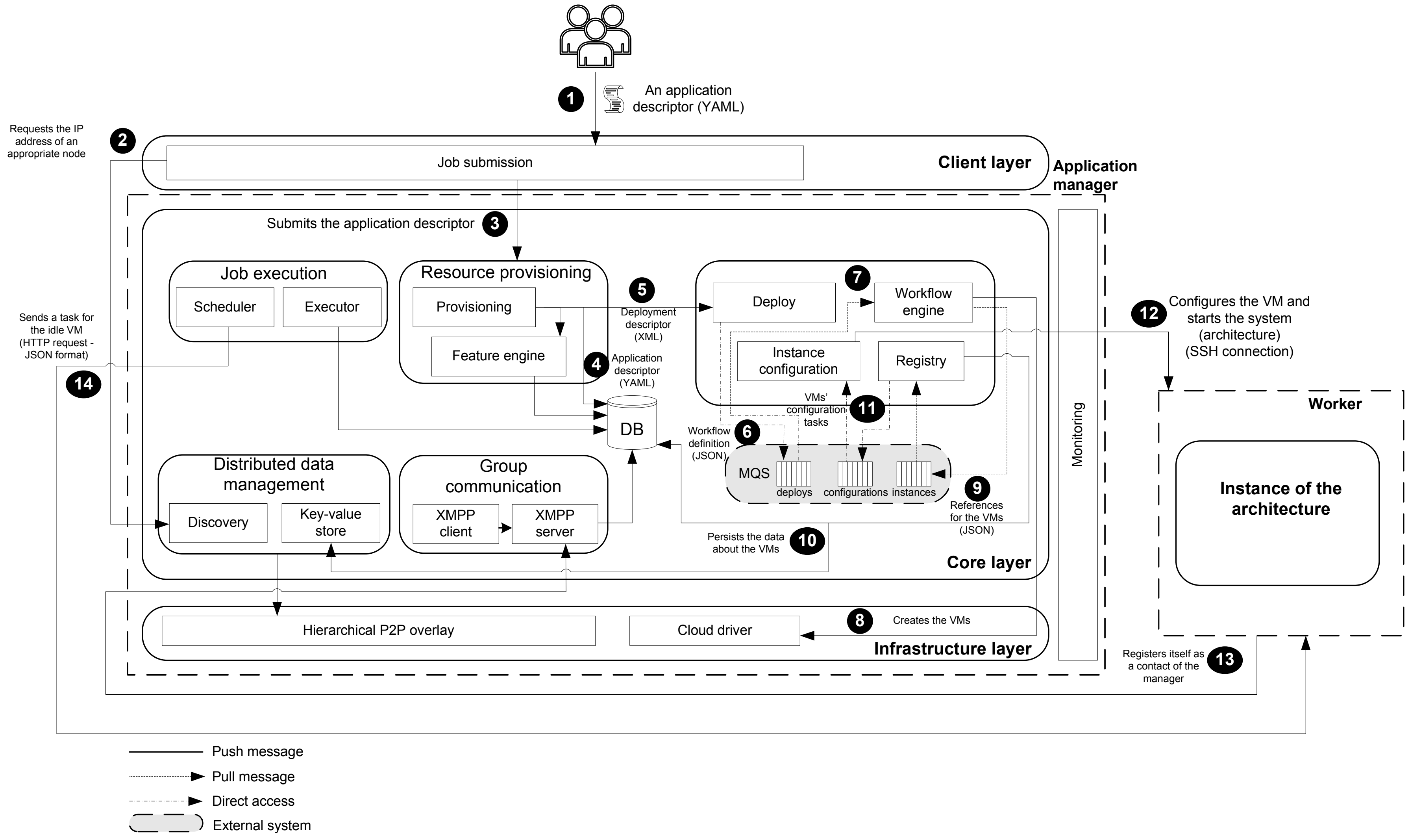
Listing 10.4: Application descriptor with the requirements and one application to be executed in one cloud

10.3 Experimental Results

10.3.1 Experimental Setup

The architecture was implemented in Java 7 and it used the RabbitMQ as the MQS. The Linux distributions: Debian and Ubuntu were used by the nodes in the clouds. In this case, the Debian was used in the experiments executed in GCE and Ubuntu in the ones executed in EC2. Table 10.2 presents the setup of the application.

In order to evaluate our architecture, we used the self-scheduling (SS) task allocation policy [342] to execute parameter sweep applications. A parameter sweep application is defined as a set $T = \{t_1, t_2, \ldots, t_m\}$ of m independent tasks. In this context, independence means that there is neither communication nor temporal precedence relations among the



1
An application descriptor (YAML)
2
Requests the IP address of an appropriate node
Job submission
Client layer
Application manager
3
Submits the application descriptor
Job execution
Scheduler
Executor
Resource provisioning
Provisioning
Feature engine
5
Deployment descriptor (XML)
4
Application descriptor (YAML)
Deploy
7
Workflow engine
Instance configuration
Registry
12
Configures the VM and starts the system (architecture) (SSH connection)
Sends a task for the idle VM (HTTP request - JSON format)
14
DB
VMs' configuration tasks
11
Workflow definition (JSON)
6
MQS
deploys
configurations
instances
9
References for the VMs (JSON)
Distributed data management
Discovery
Key-value store
Group communication
XMPP client
XMPP server
Persists the data about the VMs
10
Core layer
Monitoring
Worker
Instance of the architecture
Hierarchical P2P overlay
Cloud driver
8
Creates the VMs
Infrastructure layer
Registers itself as a contact of the manager
13
Push message
Pull message
Direct access
External system

Figure 10.5: Interaction between the architecture's module when submitted an application to execute

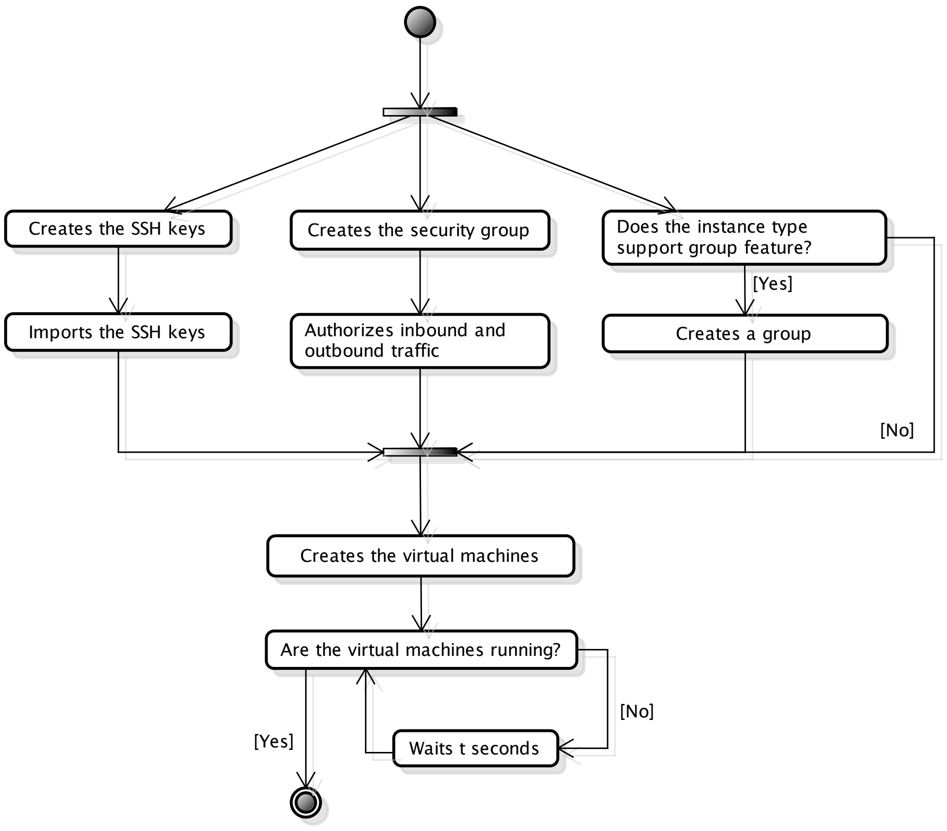

Creates the SSH keys
Creates the security group
Does the instance type support group feature?
[Yes]
Imports the SSH keys
Authorizes inbound and outbound traffic
Creates a group
[No]
Creates the virtual machines
Are the virtual machines running?
[No]
[Yes]
Waits t seconds

Figure 10.6: Workflow to create one virtual machine in the cloud

Table 10.2: Setup of the application

Software	Version
GNU/Linux x86_64	3.2.0-4-amd64
GNU/Linux x86_64	3.13.0-29-generic
OpenJDK	1.7.0_65
RabbitMQ	3.3.5
SSEARCH	36.3.6

tasks. Besides that, all the m tasks execute exactly the same program changing only the input of each task [62].

We used the SSEARCH program retrieved from www.ebi.ac.uk/Tools/sss as the parameter sweep application. In our tests, the SSEARCH application compared 24 sequences with the database UniProtKB/Swiss-Prot (September 2014), available at uniprot.org/downloads, composed of 546, 238 sequences. The query sequences are the same presented in table 7.2. For each sequence, we defined an entry in the *application descriptor*, i.e., each sequence represents a task. Listing 10.5 illustrates the definition (i.e., command line) of one task.

In this execution, we requested at least 2 cores and 6 GB of RAM memory, the Linux operating system, and a cost of at most 0.2 USD/hour. We asked to execute the task *ssearch36* 24 times. Each *ssearch36* task receives as input one query sequence ($HOME-/sequences/O60341.fasta in listing 10.5) and one database ($HOME/uniprot_sprot.fasta). Its output should be stored in $HOME/scores/O60341_scores.txt.

We evaluated our architecture considering a multiple and a single cloud scenarios, and different users' requirements. The cloud providers were Google Compute Engine (GCE) and Elastic Compute Cloud (EC2). Table 10.3 presents the users' constraints and table 10.3 the instance types that were selected based on these requirements.

Each experiment was repeated three times, and the mean was taken.

```yaml
---
name: "ssearch-app"
user:
  key: []
  username: "user"
requirements:
  cpu: 2
  memory: 6
  platform: "LINUX"
  cost: 0.2
clouds:
  cloud: []
applications:
  application:
    name: "ssearch36"
    command-line: "ssearch36 -d 0 ${query} ${database} >> ${score_table}"
    file:
    - name: "query"
      path: "$HOME/sequences/O60341.fasta"
      generated: "N"
    - name: "database"
      path: "$HOME/uniprot_sprot.fasta"
      generated: "N"
    - name: "score_table"
      path: "$HOME/scores/O60341_scores.txt"
      generated: "Y"
  ...
on-finished: "TERMINATE"
```

Listing 10.5: An application descriptor with one SSEARCH description to be executed in the cloud

Table 10.3: Users' requirements to execute the SSEARCH in the cloud

Req. #	# vCPU	Memory (GB)	Cost ($/hour)	# VM	Cloud provider
1	2	6	0.2	5	EC2
	2	6	0.2	10	
2	2	6	0.2	5	GCE
	2	6	0.2	10	
3	4	6	1.0	5	EC2
	4	6	1.0	10	
4	4	6	1.0	5	GCE
	4	6	1.0	10	
5	4	6	1.0	5	EC2 and GCE
	4	6	1.0	10	

Table 10.4: Instance types that met the users' requirements to execute the SSEARCH

Req. #	Instance type	# vCPU	Memory (GB)	Cost (USD/hour)	Family type	Cloud provider
1	m3.large	2	7.5	0.14	General	EC2
2	n1-standard-2	2	7.5	0.14	Memory	GCE
3	c3.xlarge	4	7.5	0.21	Compute	EC2
4	n1-standard-4	4	15	0.28	General	GCE
5	c3.xlarge	4	15	0.21*	General	EC2
	n1-standard-4	4	15	0.28	General	GCE

3 c3.xlarge and 2 n1-standard-4 virtual machines

10.3.2 Scenario 1: application deployment

This experiment aims to measure the deployment time, i.e., the time to create and to configure the virtual machines. The wallclock time was measured including: (a) the time to instantiate the VMs in the cloud provider; (b) the time to download and to configure all the software packages (e.g., Java, RabbitMQ, SSEARCH, among others); and (c) the time to start the architecture. By default, the architecture performs the configurations in parallel, with the number of parallel processes defined in a parameter of the architecture. In this case, the maximum of 10 configurations were done in parallel.

Figure 10.7 presents the deployment time for the instance types listed in table 10.4. As can be seen, in Amazon, increasing the number of virtual machines to deploy from 5 to 10 decreased the deployment time. This occurs because the virtual machines of each experiment are homogeneous, which enables us to request multiple instances at the same time. Similar behavior has already been observed by other works in the literature [167, 267]. On the other hand, in Google, the deployment time is proportional to the number of virtual machines.

In our experiment, the deployment time of 5/10 VMs took at most 10 minutes. With this, the applications can start across multiple VMs and multiple clouds, without requiring from users cloud and system administration skills, and also without needing the use of virtual machine image (VMI).

The 10 virtual machines of the instance type: *n1-standard-4* were deployed in a multiple cloud scenario, since GCE imposes a limit of 6 virtual machines of this type in each region.

In this case, the system allocated 5 VMs in the U.S. and 5 in Europe.

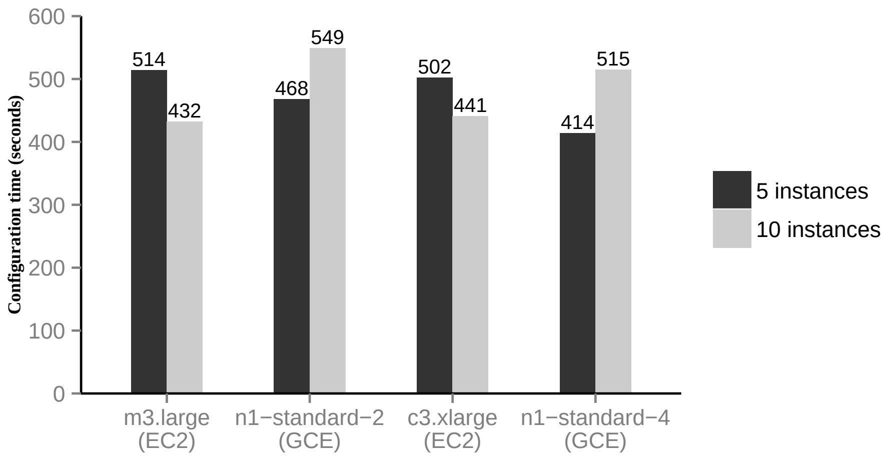


Figure 10.7: Configuration time of the virtual machines on the clouds

10.3.3 Scenario 2: application execution

Figure 10.8 presents the wallclock execution time for the four standalone experiments (i.e., requirements 1 to 4 of table 10.3), and table 10.5 their total financial cost. We can see that the instances that belong to the same family type have almost the same performance. The lower execution time (89 seconds) was achieved by the instance type *c3.xlarge* (40 vCPU cores) with a cost of 2.10 USD. Considering that the application does not take a long time to finish neither it demands many computing resources, this represents a high cost. If the users wait 33% more (43 seconds), they can pay 50% less (1.05 USD).

In figure 10.9, we present the execution time for a multi-cloud scenario. In this case, for the requirement of 5 instances (i.e., requirement 5 of table 10.3), 3 virtual machines were used from EC2 and 2 instances from GCE. If we compare figure 10.8 with figure 10.9, we can see that the execution time increased almost 34%. One reason for this difference is probably the network throughput between the clouds of different providers, since we do not observe such overhead in the scenario with ten *n1-standard-4* virtual machines distributed across two GCE's clouds.

Figure 10.10 presents the total time (i.e., deployment + execution time) for the experiments. In this figure, we can observe the impact of deploying 10 virtual machines has in the total execution time. In this case, the deploy time of 5 virtual machines was better than the deploy time of 10 virtual machines.

10.3.4 Scenario 3: application deployment and execution with failures

To evaluate the self-healing property of our architecture, we simulated two types of failures. In the first case, we stopped the application manager just after it had received the

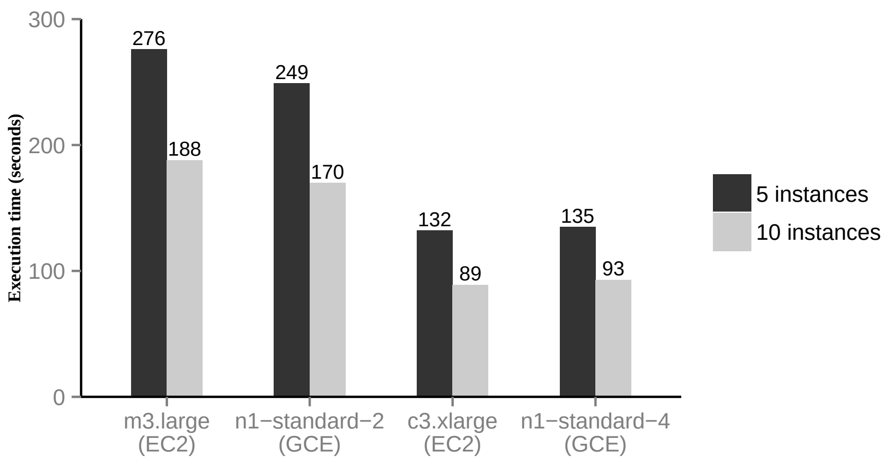


Figure 10.8: SSEARCH's execution time on the clouds to compare 24 genomics query sequences with the UniProtKB/Swiss-Prot database

Table 10.5: Financial cost for executing the application in the cloud considering different requirements (Table 10.3)

Instance type	# Instances	Wallclock time (seconds)	Total cost (USD)
m3.large	5	276	0.7
m3.large	10	188	1.4
n1-standard-2	5	249	0.7
n1-standard-2	10	170	1.4
c3.large	5	132	1.05
c3.large	10	89	2.10
n1-standard-4	5	135	1.4
n1-standard-4	10	93	2.94*
c3.large	3	131	0.63
n1-standard-4	2		0.53$^{\sharp}$
c3.large	5	119	1.05
n1-standard-4	5		1.40$^{\flat}$

**total cost: 2.96 (USD) (1.4 (U.S.) + 1.54 (Europe))*

$^{\sharp}$total cost: 1.16 USD

$^{\flat}$total cost: 2.45 USD

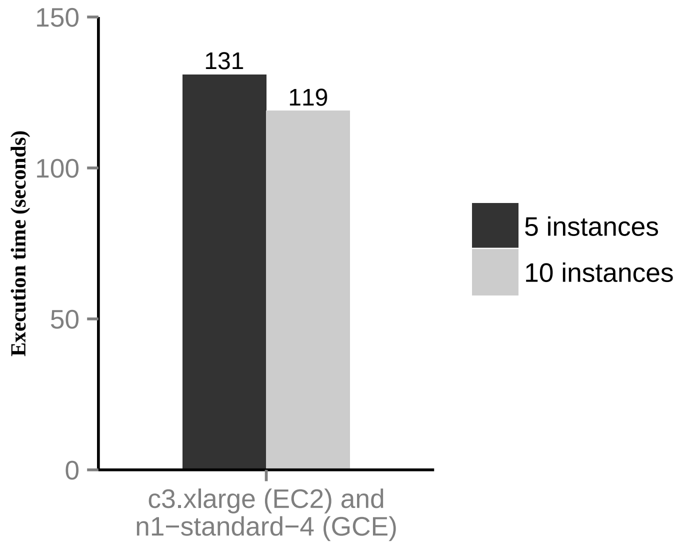


Figure 10.9: Execution time of the SSEARCH application to compare 24 genomics query sequences with the UniProtKB/Swiss-Prot database in a multi-cloud scenario

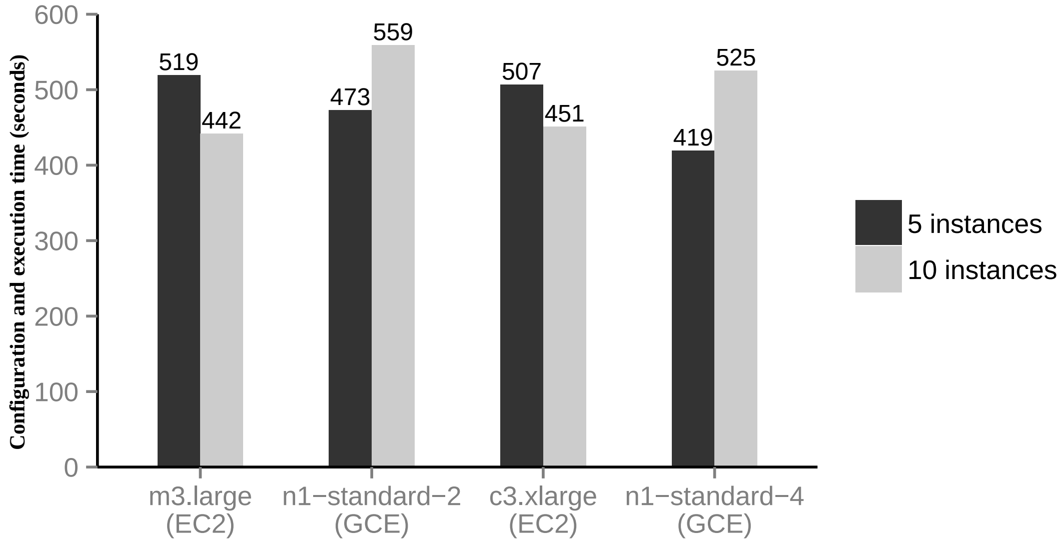


Figure 10.10: Deployment and execution time of the experiments

tasks to execute. In the second case, we executed a process that, at each minute, selected and stopped a worker. These failures were evaluated in a multi-cloud scenario (10 virtual machines – 5 *c3.xlarge* and 5 *n1-standard-4*). Figure 10.11 presents the execution time for each failure scenario.

The failure of the application manager has a high impact in the total execution time, however, it is low than of the workers. This is mostly occurs because worker's failures, demands a time from the application manager to detect it and to reassign the task of the failed worker to another one. Moreover, since multiple workers failed, this highly impacted the total execution time compared with both the failure-free scenario and the failure of the application manager. Moreover, when the application manager fails, the workers can still continue the execution of their tasks.

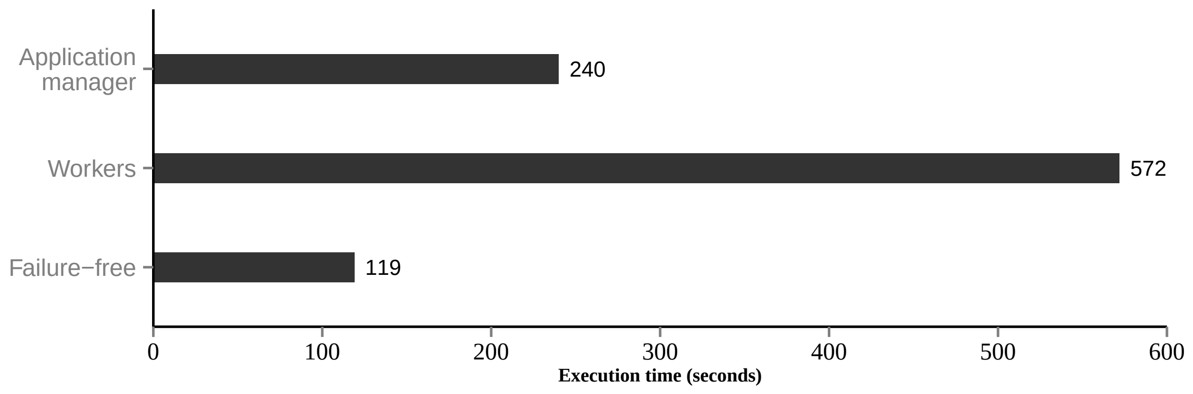


Figure 10.11: Execution time of the application on the clouds with three different type of failures

10.4 Related Work

Over the years, different cloud architectures have been proposed in the literature, with different objectives. We discussed some of these proposals in section 3.5 and chapter 4. In table 10.6, we present a comparative view of important features of these architectures. The architecture is presented in the first column. The second column presents if the architecture implements self-configuration, which means that it can select and configure the resources automatically. The third column presents if the system implements failure recovery policy. The fourth and the fifth columns show if the architectures are context-aware and if they implement self-optimization. Context-aware in this case, means if nodes are organized considering their location. The cloud model of the architecture is presented in the sixth column. Finally, the last column presents if the system can work in a multi-cloud environment.

As can be seen, half of the works implement self-configuration properties. In this case, the developers specify their needs in a service manifest, and the architecture automatically selects a cloud provider and deploys the applications. Moreover, self-healing property is implemented by some of the works. For example, Snooze [111] employs a hierarchical resource management, and uses multicast to locate the different nodes (e.g., group managers, local controllers). Similar to us, CometCloud [194] organizes the nodes using a P2P overlay to connect and to detect failures.

Only two architectures [112, 194] deploy the applications taking into account the location of the resources (i.e., nodes) where the application will run. For example, OPTMIS [112] deploys the applications taking into account the energy consumption of the cloud as well its carbon footprint. Another example is CometCloud [194]. CometCloud distributes the tasks considering three security domain level. In this case, there is a master that tries to send the tasks to trusted workers, i.e., workers that are running in a private cloud. Tasks are sent to untrusted workers (i.e., workers that are running in a public cloud) only when the SLA violation ratio exceeds a threshold.

Moreover, the majority of the architectures, implements self-optimization property to help the developers on meeting the QoS constraints of their native cloud applications [218, 278, 296, 300] or to minimize power consumption [111].

Finally, only two architectures target IaaS cloud, and the other ones platform-as-a-service (PaaS) cloud.

CometCloud [194] is the closest work to ours. Our work differs from it in the following ways. First, we consider that self-configuration is an important feature to support the users on running their applications in the clouds, since most of the cloud potential users do not have cloud and/or system administration skills. Our self-configuration relies on feature models (FMs), described in chapter 9, which enables a declarative strategy. In CometCloud, there is not self-configuration. Second, we use a hierarchical P2P overlay to organize the resources, which helps us to reduce the cost of communication between the clouds. In CometCloud, the resources are organized in the same P2P overlay, and it uses different security domains to distribute the tasks.

Table 10.6: Comparison of the cloud architectures considering their autonomic properties

Architecture	**SC**	**SH**	**CA**	**SO**	**Cloud Model**	**Multiple clouds**
Cloud-TM [300]	No	Yes	No	Yes	PaaS	Yes
JSTaaS [218]	Yes	No	No	Yes	PaaS	Yes
mOSAIC [278]	Yes	No	No	Yes	PaaS	Yes
Reservoir [296]	Yes	Yes	No	Yes	PaaS	Yes
OPTIMIS [112]	Yes	No	Yes	Yes	PaaS	Yes
FraSCaTi [317]	Yes	No	No	No	PaaS	Yes
TClouds [355]	No	Yes	No	No	PaaS	Yes
COS [165]	No	No	No	Yes	PaaS	No
Snooze [111]	No	Yes	No	Yes	IaaS	No
CometCloud [194]	No	Yes	Yes	Yes	IaaS	Yes
This work	Yes	Yes	Yes	No	IaaS	Yes

self-configuration (SC), self-healing (SH), context-awareness (CA), self-optimization (SO)

10.5 Summary

In this chapter, we presented and evaluated an IaaS cloud architecture. Our architecture enables a declarative approach, where the users describe their applications and submit it to the system. Then, the system automatically creates and configures the virtual machines in one or multiple clouds, taking into account the users' constraints; and executes the applications. In addition, the architecture implements self-healing strategies to support failures of the resources.

In our experiments, we could execute the SSEARCH to compare up to 24 query sequences with the database UniProtKB/Swiss-Prot (September 2014) in two different cloud providers and five different scenarios.

Chapter 11

Conclusion

Contents

11.1 Overview

In the 1960s, time-sharing pushed up the development of computer networks [152]. Nowadays, cloud computing is being seen as the new time-sharing [71] due to characteristics such as on-demand, pay-per-usage, and elasticity.

In addition, it is pushing down the cost of the users pay for having their own large-scale computing infrastructures, and removing the need for up-front investments to establish these computing infrastructures. In other words, cloud computing has enabled a utility computing model, offering computing, storage, and software as a service.

The utility model yields the notion of resource democratization and provides the capability for a pool of resources accessible to anyone on the Internet in nearly real-time. This notion is the main difference between cloud computing and other paradigms (e.g., grid computing systems) that have tried to deliver computing resources over the Internet before cloud [388]. Besides that, clouds offer services without knowing who the users are and unaware of the technology they use to access the services. Finally, cloud computing has also gained popularity since it can help data centers to reduce monetary costs and carbon footprint.

However, considering the cloud users and the cloud data center viewpoints, we identified four major concerns related to the cloud: (a) the high amount of energy consumed by cloud data centers; (b) the difficulty to select an appropriate cloud to execute applications at reduced cost; (c) the difficulty to configure cloud resources in an appropriate way; and (d) the difficulty to model different clouds in a uniform way.

In this context, this thesis has focused on federated clouds. Particularly, we have set out one general goal:“ *to investigate the use of federated clouds to achieve different objectives taking into account multiple users’ profiles, i.e., providers, developers, and both experienced (e.g., system administrators) and inexperienced cloud users*”. This main goal was broken down into four objectives delineated in chapter 1. In order to achieve these objectives, we, firstly, presented the state-of-the-art in large-scale distributed systems (Chapter 2), which included cluster, grid, P2P, and cloud. The descriptions of these systems were followed by some important and related concepts such as virtualization (Section 3.1.1), MapReduce (Section 3.1.3), autonomic computing (Chapter 4), and green computing (Chapter 5). Furthermore, a review of IaaS cloud architectures (Section 3.5) and autonomic computing systems (Section 4.4) were also presented.

We observed a convergence between grid, P2P, and cloud computing systems. On the one hand, many grid environments have employed virtualization techniques to provide customized execution environments and to increase resource control [4, 310, 368]. Moreover, some grid systems have relied on P2P techniques to deal with large-scale resource sharing [123]. On the other hand, clouds have started to make use of grid concepts such as resource federation [56] to increase cloud’s capacity on-demand, to increase resource availability, and to help the users (i.e., cloud providers and cloud users) on decreasing financial costs. As a result, the complexity of managing, using, and developing for these environments increased. In such context, autonomic computing has taken place. Autonomic computing systems can autonomously optimize themselves according to the users’ objectives [162], adapting to changes in the environment.

In the domain of large-scale distributed systems, autonomic computing systems have be used to decrease power consumption [239, 298], to meet performance guarantees [105, 272], and to deal with failures [111]. Nevertheless, the majority of the autonomic systems available in the literature has addressed only one autonomic property of the eight properties (Section 4.2) proposed by Horn [162]. Furthermore, it has relied on a centralized architecture, which limits the systems’ scalability.

After the state-of-the-art, we presented our contributions, which are summarized in the next section.

11.2 Summary of the Contributions

This work has succeeded in achieving the objectives pointed out in the Introduction and in addressing the issues already presented all over this document. In this section, we describe the contributions of this doctoral thesis.

A power-aware server consolidation for federated clouds

In chapter 6, we presented the first contribution of this thesis: a server consolidation strategy to reduce power consumption on cloud federations, which tackles concern (a) cited in section 11.1. Our server consolidation strategy aims to reduce power consumption on cloud federations while trying to meet QoS requirements. We assumed that clouds have a limited power consumption defined by a third party agent. In this case, we addressed applications' workloads, considering the costs to turn servers on/off and to migrate the virtual machines in the same data center and between different data centers [215]. Simulation results showed that our strategy could reduce up to 46% of the power consumption, with a slowdown of 22% in the execution time. Similar to other works [319, 375, 387], the experiments were realized through the CloudSim [58] simulator with two clouds and 400 simultaneous virtual machines. Altogether, the results demonstrated that cloud federation can provide an interesting solution to deal with power consumption, by using the computing infrastructure of other clouds when a cloud runs out of resources or when other clouds have power-efficient resources. Even though we achieved very good results with our strategy, we noticed that other variables should also be considered such as the workload type, the data center characteristics (i.e., location, power source), and the network bandwidth as these variables may impact the whole power consumption of a data center. In addition, since the CPU no longer dominates the nodes' power consumption [249], the power consumption of other components (e.g., memory, disks) must be taken into account. Moreover, resource heterogeneity should also be considered, as data centers usually comprise heterogeneous resources that can have different power consumption and curves. This requires energy and performance-aware load distribution strategies. We leave these extensions for future work.

An architecture to execute native cloud applications on a vertical cloud federation

In chapter 7, we proposed and evaluated an architecture to execute a native cloud application on a vertical cloud federation (Section 3.2.3.2) at zero-cost [214], which tackles concern (b) presented in section 11.1. The architecture used a hierarchical and distributed management strategy to allow the developers to execute their applications using services offered by two different types of cloud providers, i.e., PaaS and IaaS clouds (Section 3.2.1). In our architecture, there is a cloud coordinator, several cloud masters, and slaves. The users submit their applications to the cloud coordinator, which sends them to the cloud masters. Then, the cloud masters distribute the tasks to the slaves following a publish/subscribe model. The application was a MapReduce version of a biological sequence comparison application, which was also implemented in this work. Experimental results showed that (i) using five public clouds, the proposed architecture could outperform up to 22.55% the execution time of the best stand-alone cloud execution; (ii) the execution time was reduced from 5 hours and 44 minutes (SSEARCH sequential tool) to 13 minutes (our cloud execution). This result is comparable to the ones achieved with biological sequence comparison executions in multi-core clusters and Cell/BEs (Table 7.3 on page 127); and (iii) the federation could enable the execution of huge cloud-aware application at no expense and without being tied to any cloud provider. In this experiment, we executed biological sequence comparisons. Although we could execute a huge application in a vertical cloud federation, our architecture had some issues such as: (i) the usage of a centralized coordinator to distribute the tasks, and (ii) the lack of fault-tolerance strategies for both cloud coordinator and cloud masters. The first issue limited the scalability of the architecture and its usage in a dynamic environment. In addition, the failure of the coordinator required the re-execution of the whole application, as the architecture did not provide a way to discover the tasks distributed to each cloud master. In this case, the masters continued the execution of their tasks, but the results are inaccessible for the users.

Besides that, even though MapReduce applications are computing infrastructure agnostic, it became clear for us that designing and developing applications for different clouds represent a difficult task even for experienced cloud developers, due to the various constraints and resource types offered by the clouds, i.e., clouds' heterogeneity. Hence, at this point, we decided to investigate: (a) the feasibility of the use of a cloud environment by ordinary users to execute cloud-unaware applications, without needing to re-engineering the applications; and (b) models that may help the users on dealing with cloud heterogeneity, offering a level of abstraction that can be understood by different user profiles and that can also promote reuse.

Excalibur

In chapter 8, we proposed and evaluated a cloud architecture that aims to: (i) help unskilled cloud users on executing their applications on the cloud; (ii) scale the applications requiring minimal users' intervention; and (iii) try to meet the users objectives such as performance and reduced financial cost [217]. To achieve such goals and also to tackle the concern (c) cited in section 11.1, our architecture sets up the whole cloud environment, and it tries to speed up the execution of the applications by implementing an auto-scaling strategy. Our auto-scaling targets applications that comprise independent tasks, although the applications were developed to execute sequentially. The independent tasks are identified based on high-level descriptions provided by the user. We evaluated the architecture executing a genomics workflow on Amazon EC2, considering two scenarios. In the first scenario, the users selected the resources, whereas in the second one, our architecture, based on historical data, automatically selected the resources. The experiments showed that our auto-scaling strategy could outperform the execution time of the resource chosen by the users,dynamically scaling the applications up to 11 virtual machines (VMs). Moreover, it also helped on reducing the monetary cost in 84%. However, this architecture has the following issues: (i) the users had to select the resources in the first execution of the applications; (ii) it assumed an environment where the constraints associated to a resource were the same between all the clouds belonging to a cloud provider. Nevertheless, the clouds of a provider are commonly heterogeneous.

A software product line engineering method for resource selection and configuration on federated clouds

Motivated by the difficulties which we have observed when developing for multiple clouds and by the complexity tasks required to configure a real cloud environment, as explained in our two previous contributions, we decided to investigate models that would tackle these problems. Particularly, we were interested in models that would (i) support the description of clouds' services independent of cloud providers; (ii) enable automatic resources provisioning on multiple clouds, taking into account temporal and functional dependencies between the resources, thus leaving the environments in a consistent state;

(iii) help on achieving the goals (i.e., functional and non-functional requirements) of different user profiles, i.e., cloud experts, system administrators, developers, and novice users; and (iv) provide a level of abstraction suitable for these different kinds of users. To the best of our knowledge, the literature lacked such models. In order to tackle these problems, in chapter 9, we used a software product line (SPL) engineering method. Using SPL, we defined a feature-based variability model for the cloud configurations. Using this model, we were able to define an architecture and to derive appropriate products. By employing an SPL engineering method, we could capture the knowledge of creating and configuring cloud computing environments in the form of reusable assets. Moreover, non-technical users understand feature models, as they refer to domain concepts. In the context of this thesis, a valid product is a cloud computing environment (i.e., virtual machine (VM) and applications) that meets the users' requirements, where the requirements can be either based on high or low-level descriptions.

While high-level descriptions include the number of CPU cores, the operating system, the minimal amount of RAM memory, and the maximum financial cost per hour, low-level descriptions include the virtualization type, the sustainable performance, the disk technology, among others. Besides that, a cloud computing environment must also match cloud's configuration constraints and applications dependencies. From the users' viewpoint, using an SPL engineering method led to customizable cloud environments at lower financial costs. In addition, as various assets were reused, it increased reliability and correctness [92]. Thus, in particular, our contributions in this part are the following: (i) the use of extended feature model (EFM) (Section 9.4) with attributes to describe IaaS cloud environments. The feature model (FM) handles the commonalities and variabilities at the IaaS layer, enabling the description of the whole computing environment (i.e., hardware and software) independent of cloud provider and without requiring the usage of virtual machine image; (ii) a declarative and automated strategy that allows resource selection and configuration on a multi-cloud scenario; (iii) a domain configuration knowledge mapping the feature models into reusable assets; (iv) the automated deployment of the computing environment on the clouds.

We also developed a prototype to evaluate our method considering two different cloud providers. In this prototype, we used Choco [178], an off-the-shelf constraint satisfaction problem (CSP) solver, to implement our feature-based model and to select the configurations that meet the users' objectives. Experimental results showed that using the proposed method the users could get an optimal configuration with regard to their objectives without needing to know the constraints and variabilities of each cloud. Moreover, our model enabled application deployment and reconfiguration at runtime in a federated cloud scenario, without requiring the usage of virtual machine image (VMI). The proposed SPL engineering method was used to implement an autonomic system, which tackles the concerns (c) and (d) pointed out in section 11.1.

Dohko

In chapter 10, we proposed and evaluated our last contribution: an autonomic and goal-oriented system for federated clouds. Our autonomic system aims to tackle the issues identified in our second contribution (Chapter 7) and to enable a declarative strategy to execute the users' applications on multiple clouds, following the SPL engineering method described in the previous section and in chapter 9. Our system implements the following autonomic properties: self-configuration, self-healing, and context-awareness. Moreover, it relies on a hierarchical P2P overlay [216] to deal with failures and to reduce inter-cloud communication. By employing a declarative strategy, the system could execute a biological sequence comparison application in a single and in a federated cloud scenario, requiring minimal users' intervention to select, to deploy, and to configure the whole cloud environment. In particular, our system tackled the lack of middleware prototypes that can support different scenarios when using services from multiple IaaS clouds. Moreover, it met the various functional requirements identified for multiple cloud-unaware systems [277] such as: (i) it provides a way to describe functional and non-functional requirements through the usage of a product line engineering (PLE) method (Chapter 9); (ii) it can aggregate services from different clouds; (iii) it provides a homogeneous interface to access services of multiple clouds; (iv) it allows the service selection of the clouds; (v) it can deploy its components on multiple clouds; (vi) it provides automatic procedures for deployments; (vii) it utilizes an overlay network to connect and to organize the resources; (viii) it does not impose constraint for the connected clouds.

One issue of our system is the lack of a self-optimization strategy, since it does not focus on task scheduling. We leave this extension as our future work. The software prototype is available at dohko.io.

11.3 Threat to Validity

There are some concerns related to the validity of our contributions described in the second part of this doctoral thesis. First, the systems and feature models illustrated in this work were built based on our experience in developing, deploying, and configuring IaaS cloud services, as well as in administrating computing environments. Seconds, these models were limited to the features released by the cloud providers. Third, since our method relied on benchmarks to gather the qualitative attributed of the resources, small variations in their result values might occur mostly for two reasons: (i) we cannot control the allocation of the physical resources to the virtual ones and (ii) clouds' workload may change over time, as well as the cloud providers' objectives. Fourth, due to the evolution of cloud computing, some resources, clouds or even providers may appear and/or disappear. Thus, the experiments and concrete feature models presented in this thesis might not be valid anymore over the long term. However, this does not invalidate our abstract feature model (Figure 9.6 on page 163), it is independent of cloud provider. Moreover, our architecture (Figures 10.1 and 10.5) is not invalidated too. Finally, although the examples deal with public clouds such as EC2, GCE, and Heroku, our contributions are not limited to these clouds, as the cloud platforms follow the same principles employed by these cloud providers. Examples of cloud platforms include OpenStack (openstack.org), OpenNebula [248], CloudStack (cloudstack.apache.org), Eucalyptus [263], and Nimbus [186]

11.4 Perspectives

In this section, we present some future research directions for this work. They can be categorized as follows: (a) energy management models for multiple clouds; (b) cloud computing management and opportunistic computing; (c) supporting other autonomic properties; and (d) improved mechanisms to support the developers on writing self-adaptive applications.

Energy management models for multiple clouds: it should be interesting to consider a dynamic multi-cloud scenario, where the objectives and incentives of the providers change over time. In this case, we intend to investigate strategies based on game theory to model the different interactions among cloud entities (e.g., users, providers), taking into account selfish behaviors and conflicts. Moreover, we want to extend our power-aware strategy to take into consideration other variables as the type of power source, data center location, and carbon neutrality. For example, carbon neutrality can be achieved by purchasing carbon credits from the providers on the federation, distributing the workload across the clouds with both low electricity cost and carbon footprint, and/or using workload consolidation to reduce the number of underutilized resources.

Cloud computing management and opportunistic computing: we also intend to combine the computational power of mobile devices with the cloud in such a way that applications autonomously decide which code can be performed locally and which one should be offloaded to the cloud. This can be used to reduce financial cost or to reduce power consumption of both portable devices and data center, as well as to maximize the usage of mobile devices. A difficulty may be the coordination and distribution of the work for the devices. In this case, a P2P coordination model can be employed together with the usage of a product line engineering method, as the one described in the chapter 9, to deal with functional and non-functional dependencies. Moreover, distributed resources might be aggregated to create micro-clouds[1] in order to provide enough resource to an application.

Supporting other autonomic properties: we plan to implement self-optimization and to extend our context-awareness implementation to consider other reconfiguration scenarios at runtime, according to the characteristics of the applications and the appearance of new clouds or resources. In addition, we also intend to investigate the implementation of task scheduling strategies such as work stealing [177, 303] in cloud federation environments.

Improved mechanisms to support the developers on writing self-adaptive applications: although software product line helps on improving maintainability of an application, there is still a needing for tools and frameworks to support the development of self-adaptive applications for large-scale environments. In this case, domain-specific languages might be developed. Today, we have Class feature reference (Clafer) [25] that allows the domain engineers to validate the domain models and to generate the representation of the models to be reasoned by some solvers such as Alloy [169] and Choco [178].

[1] A micro-cloud represents the usage of commodity computing infrastructures to aggregate computational capacity at a low financial cost. In this scenario, the clouds are similar to a peer on a P2P system.

Thus, similar strategy might be created to validate the development of product lines, and to support autonomic applications' decisions at runtime.

11.5 Summary

Cloud represents a computing model for providing high performance computing as utility, promoting the cooperation among different organizations and also to push down the costs. We expect that studies, such as the ones presented in this thesis, contribute for the development of cloud's systems, i.e., to the development of the metacomputer, approximating to the utility computing concept envisioned by McCarthy [132], Smarr and Catlett [326], and Dertouzos [99].

Bibliography

[1] Moustafa AbdelBaky, Manish Parashar, Hyunjoo Kim, Kirk E. Jordan, Vipin Sachdeva, James Sexton, Hani Jamjoom, Zon-Yin Shae, Gergina Pencheva, Reza Tavakoli, and Mary F. Wheeler. Enabling high-performance computing as a service. *Computer*, 45(10):72–80, 2012. (Cited on pages 129 and 141)

[2] Karl Aberer. P-Grid: A self-organizing access structure for P2P information systems. In *9th International Conference on Cooperative Information Systems*, pages 179–194, 2001. (Cited on page 25)

[3] Ajith Abraham and Lakhmi Jain. Evolutionary multiobjective optimization. In Ajith Abraham, Lakhmi Jain, and Robert Goldberg, editors, *Evolutionary Multiobjective Optimization*, pages 1–6. Springer, 2005. (Cited on page 152)

[4] Sumalatha Adabala, Vineet Chadha, Puneet Chawla, Renato Figueiredo, José Fortes, Ivan Krsul, Andrea Matsunaga, Mauricio Tsugawa, Jian Zhang, Ming Zhao, Liping Zhu, and Xiaomin Zhu. From virtualized resources to virtual computing grids: The in-vigo system. *Future Generation Computer Systems*, 21(6):896–909, 2005. (Cited on page 213)

[5] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, 2006. (Cited on page 39)

[6] G. Agosta, M. Bessi, E. Capra, and C. Francalanci. Dynamic memoization for energy efficiency in financial applications. In *International Green Computing Conference and Workshops*, pages 1–8, 2011. (Cited on pages 91 and 109)

[7] A.M. Aji and Wu chun Feng. Optimizing performance, cost, and sensitivity in pairwise sequence search on a cluster of PlayStations. In *IEEE International Conference on BioInformatics and BioEngineering*, pages 1–6, 2008. (Cited on pages 115 and 127)

[8] M. Albano, L. Ricci, and L. Genovali. Hierarchical P2P overlays for DVE: An additively weighted Voronoi based approach. In *International Conference on Ultra Modern Telecommunications Workshops*, pages 1–8, 2009. (Cited on page 27)

[9] André Almeida, Everton Cavalcante, Thais Batista, Nélio Cacho, Frederico Lopes, Flavia Delicato, and Paulo Pires. Dynamic adaptation of cloud computing applications. In *25th International. Conference on Software Engineering and Knowledge Engineering*, pages 67–72, 2013. (Cited on pages 179, 184, and 187)

[10] Amazon. Summary of the Amazon EC2 and Amazon RDS service disruption in the US east region. aws.amazon.com/message/65648, 2011. Last accessed in July 2014. (Cited on page 48)

[11] Amazon. Summary of the AWS service event in the US east region. aws.amazon.com/message/67457, 2012. Last accessed in July 2014. (Cited on page 48)

[12] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@Home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002. (Cited on pages 15 and 34)

[13] Sergio Andreozzi, Natascia De Bortoli, Sergio Fantinel, Antonia Ghiselli, Gian Luca Rubini, Gennaro Tortone, and Maria Cristina Vistoli. GridICE: A monitoring service for grid systems. *Future Generation Computer Systems*, 21(4):559–571, 2005. (Cited on page 53)

[14] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371, 2004. (Cited on pages 15, 21, 23, 24, 25, 28, and 34)

[15] Samuel V Angiuoli, Malcolm Matalka, Aaron Gussman, Kevin Galens, Mahesh Vangala, David R Riley, Cesar Arze, James R White, and W Florian Fricke. CloVR: A virtual machine for automated and portable sequence analysis from the desktop using cloud computing. *BMC Bioinformatics*, 12(1):1–15, 2011. (Cited on pages 132, 141, and 143)

[16] Tomonori Aoyama and Hiroshi Sakai. Inter-cloud computing. *Business & Information Systems Engineering*, 3(3):173–177, 2011. (Cited on pages 52 and 53)

[17] Knarig Arabshian and Henning Schulzrinne. Distributed context-aware agent architecture for global service discovery. In *2nd International Workshop on Semantic Web Technology For Ubiquitous and Mobile Applications*, 2006. (Cited on page 199)

[18] Danilo Ardagna, Elisabetta Di Nitto, Giuliano Casale, Dana Petcu, Parastoo Mohagheghi, Sébastien Mosser, Peter Matthews, Anke Gericke, Cyril Ballagny, Francesco D'Andria, Cosmin-Septimiu Nechifor, and Craig Sheridan. MODAClouds: A model-driven approach for the design and execution of applications on multiple clouds. In *ICSE Workshop on Modeling in Software Engineering*, pages 50–56, 2012. (Cited on page 60)

[19] Danilo Ardagna, Barbara Panicucci, Marco Trubian, and Li Zhang. Energy-aware autonomic resource allocation in multitier virtualized environments. *IEEE Transactions on Services Computing*, 5(1):2–19, 2012. (Cited on pages 86, 88, and 113)

[20] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, February 2009. URL www.eecs.berkeley.edu/Pubs/TechRpts/2009/

EECS-2009-28.html. Last accessed in January 2014. (Cited on pages 16, 31, 35, and 54)

[21] James Aspnes and Gauri Shah. Skip Graphs. In *14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, 2003. (Cited on page 25)

[22] Arutyun I. Avetisyan, Roy Campbell, Indranil Gupta, Michael T. Heath, Steven Y. Ko, Gregory R. Ganger, Michael A. Kozuch, David O'Hallaron, Marcel Kunze, Thomas T. Kwan, Kevin Lai, Martha Lyons, Dejan S. Milojicic, Hing Yan Lee, Yeng Chai Soh, Ng Kwang Ming, Jing-Yuan Luke, and Han Namgoong. Open Cirrus: A global cloud computing testbed. *Computer*, 43(4):35–43, 2010. (Cited on pages 69, 70, and 74)

[23] Dan Azevedo and Andy Rawson. Measuring data center productivity. Technical report, AMD Metrics and Measurements Work Group, 2008. URL bit.ly/1ynfWzp. Last accessed in April 2014. (Cited on page 99)

[24] Lee Badger, Tim Grance, Robert Patt-Corner, and Jeff Voas. Cloud computing synopsis and recommendations. Technical Report NIST Special Publication 800-146, National Institute of Standards and Technology, May 2012. URL csrc.nist.gov/publications/nistpubs/800-146/sp800-146.pdf. Last accessed in July 2014. (Cited on pages 30 and 47)

[25] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wasowski. Feature and meta-models in Clafer: Mixed, specialized, and coupled. In *3rd International Conference on Software Language Engineering*, pages 102–122, 2011. (Cited on page 218)

[26] H.M.N. Dilum Bandara and Anura P. Jayasumana. Collaborative applications over peer-to-peer systems-challenges and solutions. *Peer-to-Peer Networking and Applications*, 6(3):257–276, 2013. (Cited on page 28)

[27] D.F. Bantz, C. Bisdikian, D. Challener, J.P. Karidis, S. Mastrianni, A. Mohindra, D.G. Shea, and M. Vanover. Autonomic personal computing. *IBM Systems Journal*, 42(1):165–176, 2003. (Cited on page 77)

[28] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003. (Cited on page 38)

[29] Luiz André Barroso. The price of performance. *ACM Queue*, 3(7):48–53, 2005. (Cited on pages 90 and 99)

[30] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40:33–37, 2007. (Cited on page 89)

[31] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2nd edition, 2013. (Cited on pages 2, 3, and 101)

[32] Don Batory. Feature models, grammars, and propositional formulas. In *9th International Conference on Software Product Lines*, pages 7–20, 2005. (Cited on pages 154 and 155)

[33] J.-P. Baud, J. Casey, S. Lemaitre, and C. Nicholson. Performance analysis of a file catalog for the LHC computing grid. In *14th IEEE International Symposium High Performance Distributed Computing*, pages 91–99, 2005. (Cited on page 11)

[34] Christian L. Belady. In the data center, power and cooling costs more than the it equipment it supports. *Electronics Cooling*, 13(1), 2007. (Cited on page 99)

[35] Anton Beloglazov and Rajkumar 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*, 24(13), 2012. (Cited on page 112)

[36] Anton Beloglazov, Rajkumar Buyya, Young Choon Lee, and Albert Zomaya. A taxonomy and survey of energy-efficient data centers and cloud computing systems. In *Advances in Computers*, volume 82, pages 47–111. Elsevier, 2011. (Cited on pages 41, 90, 91, 101, and 102)

[37] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6): 615–636, 2010. (Cited on page 155)

[38] Andreas Berl, Erol Gelenbe, Marco Di Girolamo, Giovanni Giuliani, Hermann De Meer, Minh Quan Dang, and Kostas Pentikousis. Energy-efficient cloud computing. *Computer Journal*, 53:1045–1051, 2010. (Cited on page 90)

[39] Tim Berners-Lee. Information management: A proposal. www.w3.org/History/1989/proposal.html, 1989. Last accessed in January 2014. (Cited on page 13)

[40] Tim Berners-Lee. WWW: past, present, and future. *Computer*, 29(10):69–77, 1996. (Cited on page 13)

[41] David Bernstein, Erik Ludvigson, Krishna Sankar, Steve Diamond, and Monique Morrow. Blueprint for the intercloud - protocols and formats for cloud computing interoperability. In *4th International Conference on Internet and Web Applications and Services*, pages 328–336, 2009. (Cited on page 51)

[42] Alysson Bessani, Rüdiger Kapitza, Dana Petcu, Paolo Romano, Spyridon V. Gogouvitis, Dimosthenis Kyriazis, and Roberto G. Cascella. A look to the old-world*sky: Eu-funded dependability cloud computing research. *ACM SIGOPS Operating Systems Review*, 46(2):43–56, 2012. (Cited on page 62)

[43] Cor-Paul Bezemer and Andy Zaidman. Multi-tenant saas applications: Maintenance dream or nightmare? In *International Workshop on Principles of Software Evolution*, pages 88–92, 2010. (Cited on page 31)

[44] Ricardo Bianchini and Ram Rajamony. Power and energy management for server systems. *IEEE Computer*, 37(11):68–74, 2004. (Cited on page 89)

[45] Philip Bianco, Grace A. Lewis, and Paulo Merson. Service level agreements in service-oriented architecture environments. Technical Report CMU/SEI-2008-TN-021, Software Engineering Institute, September 2008. URL `sei.cmu.edu/reports/08tn021.pdf`. Last accessed in February 2014. (Cited on page 41)

[46] Ken Birman, Gregory Chockler, and Robbert van Renesse. Toward a cloud computing research agenda. *ACM SIGACT News*, 40(2):68–80, 2009. (Cited on page 32)

[47] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. In *9th ACM Symposium on Operating Systems Principles*, page 3, 1983. (Cited on page 13)

[48] Günter Böckle, Klaus Pohl, and Frank van der Linden. A framework for software product line engineering. In *Software Product Line Engineering*, pages 19–38. Springer, 2005. (Cited on page 161)

[49] Robert B. Bohn, John Messina, Fang Liu, Jin Tong, and Jian Mao. NIST cloud computing reference architecture. In *IEEE World Congress on Services*, pages 594–596, 2011. (Cited on page 51)

[50] Pat Bohrer, Elmootazbellah N. Elnozahy, Tom Keller, Michael Kistler, Charles Lefurgy, Chandler McDowell, and Ram Rajamony. *The case for power management in web servers*, pages 261–289. Kluwer Academic Publishers, 2002. (Cited on page 89)

[51] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web Services Architecture. `www.w3.org/TR/2004/NOTE-ws-arch-20040211`, 2004. Last accessed in January 2014. (Cited on page 15)

[52] Damien Borgetto, Michael Maurer, Georges Da-Costa, Jean-Marc Pierson, and Ivona Brandic. Energy-efficient and SLA-aware management of IaaS clouds. In *3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet*, pages 25:1–25:10, 2012. (Cited on page 41)

[53] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *International Symposium on Computer architecture*, pages 83–94, 2000. (Cited on page 91)

[54] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing virtualization to the x86 architecture with the original VMware workstation. *ACM Transactions on Computer Systems*, 30(4):12:1–12:51, 2012. (Cited on page 15)

[55] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall PTR, 1999. (Cited on page 17)

[56] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25: 599–616, 2009. (Cited on pages 29, 48, 90, and 213)

[57] byte-unixbench. A Unix benchmark suite. code.google.com/p/byte-unixbench, 2014. Last accessed in July 2014. (Cited on page 168)

[58] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011. (Cited on pages 32, 61, 103, 106, 107, 112, 113, and 214)

[59] Zhibo Cao and Shoubin Dong. An energy-aware heuristic framework for virtual machine consolidation in cloud computing. *The Journal of Supercomputing*, 69(1): 429–451, 2014. (Cited on pages 112 and 113)

[60] Eugenio Capra, Chiara Francalanci, and Sandra A. Slaughter. Is software "green"? application development environments and energy efficiency in open source applications. *Information and Software Technology*, 54(1):60–71, 2012. (Cited on page 90)

[61] Emanuele Carlini, Massimo Coppola, Patrizio Dazzi, Laura Ricci, and Giacomo Righetti. Cloud federations in Contrail. In *International Conference on Parallel Processing*, pages 159–168, 2011. (Cited on pages 71, 72, and 74)

[62] Henri Casanova, Dmitrii Zagorodnov, Francine Berman, and Arnaud Legrand. Heuristics for scheduling parameter sweep applications in grid environments. In *9th Heterogeneous Computing Workshop*, pages 349–363, 2000. (Cited on page 204)

[63] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2): 141–154, 1988. (Cited on page 195)

[64] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony I. T. Rowstron. One ring to rule them all: service discovery and binding in structured peer-to-peer overlay networks. In *ACM SIGOPS European Workshop*, pages 140–145, 2002. (Cited on page 27)

[65] Miguel Castro, Manuel Costa, and Antony Rowstron. Debunking some myths about structured and unstructured overlays. In *2nd Conference on Symposium on Networked Systems Design & Implementation*, pages 85–98. USENIX, 2005. (Cited on page 27)

[66] Danielle Catteddu and Gilles Hogben. Cloud computing: Benefits, risks and recommendations for information security. Technical report, European Network and Information Security Agency (ENISA), February 2009. URL www.enisa.europa.eu/act/rm/files/deliverables/cloud-computing-risk-assessment/at_download/fullReport. Last accessed in February 2014. (Cited on page 41)

[67] Everton Cavalcante, André Almeida, Thais Batista, Nélio Cacho, Frederico Lopes, Flavia C. Delicato, Thiago Sena, and Paulo F. Pires. Exploiting software product

lines to develop cloud computing applications. In *16th International Software Product Line Conference*, pages 179–187, 2012. (Cited on pages 148, 179, 184, and 187)

[68] Antonio Celesti, Francesco Tusa, Massimo Villari, and Antonio Puliafito. How to enhance cloud architectures to enable cross-federation. In *3rd IEEE International Conference on Cloud Computing*, pages 337–345, 2010. (Cited on page 49)

[69] Antonio Celesti, Francesco Tusa, Massimo Villari, and Antonio Puliafito. Three-phase cross-cloud federation model: The cloud SSO authentication. In *2nd International Conference on Advances in Future Internet*, pages 94–101, 2010. (Cited on page 51)

[70] Antonio Celesti, Francesco Tusa, and Massimo Villari. Toward Cloud Federation: Concepts and Challenges. In Ivona Brandic, Massimo Villari, and Francesco Tusa, editors, *Achieving Federated and Self-Manageable Cloud Infrastructures: Theory and Practice*, pages 1–17. IGI Global, 2012. (Cited on page 74)

[71] Vinton G. Cerf. ACM and the professional programmer. *Queue*, 12(7):10:10–10:11, 2014. (Cited on page 212)

[72] Vinton G. Cerf and Robert E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, Com-22(5):627–641, 1974. (Cited on page 12)

[73] Kelly Chard, Michael Russell, Yves A. Lussier, Eneida A Mendonça, and Jonathan C. Silverstein. Scalability and cost of a cloud-based approach to medical NLP. In *24th International Symposium on Computer-Based Medical Systems*, pages 1–6, 2011. (Cited on page 147)

[74] Ramnath Chellappa. Intermediaries in cloud-computing: A new computing paradigm. *INFORMS*, 1997. (Cited on page 15)

[75] Yee Ming Chen and Hsin-Mei Yeh. An implementation of the multiagent system for market-based cloud resource allocation. *Journal of Computing*, 2(11):27–33, 2010. (Cited on pages 112 and 113)

[76] Dazhao Cheng, Changjun Jiang, and Xiaobo Zhou. Heterogeneity-aware workload placement and migration in distributed sustainable datacenters. In *28th IEEE Internal Parallel and Distributed Processing Symposium*, pages 307–316, 2014. (Cited on page 92)

[77] Ann Chervenak and Shishir Bharathi. Peer-to-peer approaches to grid resource discovery. In *Making Grids Work*. Springer, 2008. (Cited on page 24)

[78] Trieu C. Chieu, Ajay Mohindra, Alexei Karve, and Alla Segal. Solution-based deployment of complex application services on a cloud. In *IEEE International Conference on Service Operations and Logistics and Informatics*, pages 282–287, 2010. (Cited on pages 177, 178, 184, and 187)

[79] Wu chun Feng, Xizhou Feng, and Rong Ge. Green supercomputing comes of age. *IT Professional*, 10:17–23, 2008. (Cited on page 93)

[80] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *2nd Conference on Symposium on Networked Systems Design & Implementation*, pages 273–286. USENIX Association, 2005. (Cited on page 39)

[81] Stuart Clayman, Alex Galis, Clovis Chapman, Giovanni Toffetti, Luis Rodero-Merino, Luis M. Vaquero, Kenneth Nagin, and Benny Rochwerger. Monitoring service clouds in the future internet. In Georgios Tselentis, John Domingue, Alex Galis, Anastasius Gavras, David Hausheer, Srdjan Krco, Volkmar Lotz, and Theodore Zahariadis, editors, *Towards the Future Internet: Emerging Trends from European Research*, pages 115–126. IOS Press, 2010. (Cited on page 53)

[82] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. (Cited on pages 147 and 154)

[83] Carlos A. Coello Coello, Gary B. Lamont, and David A. Van Veldhuizen. *Evolutionary algorithms for solving multi-objective problems*, volume 242. Springer, 2 edition, 2002. (Cited on page 152)

[84] Jared L. Cohon. *Multiobjective programming and planning*. Academic Press, 1978. (Cited on page 152)

[85] Gary Cook, Tom Dowdall, David Pomerantz, and Yifei Wang. Clicking Clean: How companies are creating the green Internet 2014. Technical report, Greenpeace, April 2014. URL `greenpeace.org/usa/Global/usa/planet3/PDFs/clickingclean.pdf`. Last accessed in May 2014. (Cited on pages 90 and 92)

[86] Stefania Costache, Nikos Parlavantzas, Christine Morin, and Samuel Kortas. An economic approach for application QoS management in clouds. In *International Conference on Parallel Processing*, pages 426–435, 2012. (Cited on pages 81 and 88)

[87] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luís Rodrigues. D2STM: Dependable distributed software transactional memory. In *15th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 307–313, 2009. (Cited on page 62)

[88] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley, 5th edition, 2011. (Cited on pages 11, 12, 13, 14, and 190)

[89] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *7th International Workshop on the Web and Databases*, pages 25–30, 2004. (Cited on page 25)

[90] Adina Crainiceanu, Prakash Linga, Ashwin Machanavajjhala, Johannes Gehrke, and Jayavel Shanmugasundaram. P-ring: An efficient and robust P2P range index structure. In *ACM SIGMOD International Conference on Management of Data*, pages 223–234, 2007. (Cited on page 25)

[91] Cycle Computing. New cyclecloud HPC cluster is a triple threat: 30000 cores, $1279/hour, & grill monitoring GUI for Chef. bit.ly/cyclecomputing, 2011. Last accessed in February 2014. (Cited on page 16)

[92] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley, 2000. (Cited on pages 154 and 216)

[93] Rajarshi Das, Jeffrey O. Kephart, Charles Lefurgy, Gerald Tesauro, David W. Levine, and Hoi Chan. Autonomic multi-agent management of power and performance in data centers. In *AAMAS*, pages 107–114, 2008. (Cited on pages 112 and 113)

[94] Doug Davis and Gilbert Pilz. Cloud Infrastructure Management Interface (CIMI) Model and RESTful HTTP-based Protocol: An interface for managing cloud infrastructure. Technical Report DSP0263, Distributed Management Task Force, May 2012. URL dmtf.org/sites/default/files/standards/documents/DSP0263_1.0.0.pdf. Last accessed in July 2014. (Cited on page 53)

[95] Daniel de Oliveira, Eduardo Ogasawara, Fernanda Baião, and Marta Mattoso. SciCumulus: A lightweight cloud middleware to explore many task computing paradigm in scientific workflows. In *3rd IEEE International Conference on Cloud Computing*, pages 378–385, 2010. (Cited on pages 60, 62, and 74)

[96] Frederico Alvares de Oliveira, Jr. and Thomas Ledoux. Self-management of cloud applications and infrastructure for energy optimization. *ACM SIGOPS Operating Systems Review*, 46(2):10–18, 2012. (Cited on pages 82 and 88)

[97] Edans Flavius de Oliveira Sandes and Alba Cristina Magalhães Alves de Melo. Retrieving Smith-Waterman Alignments with Optimizations for Megabase Biological Sequences Using GPU. *IEEE Transactions on Parallel and Distributed Systems*, 24 (5):1009–1021, 2013. (Cited on pages 115 and 127)

[98] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Conference on Symposium on Opearting Systems Design & Implementation*, pages 137–149. USENIX Association, 2004. (Cited on pages xxii, 16, 17, 43, and 44)

[99] Michael L. Dertouzos. *The Unfinished Revolution: Human-Centered Computers and What They Can Do for Us*. HarperInformation, 2001. (Cited on page 219)

[100] Scott W Devine, Edouard Bugnion, and Mendel Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture, 1998. US Patent 6397242. (Cited on page 38)

[101] Roger Dingledine, Michael J. Freedman, and David Molnar. The free haven project: Distributed anonymous storage service. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 67–95, 2000. (Cited on page 24)

[102] Brian Dougherty, Jules White, and Douglas C. Schmidt. Model-driven auto-scaling of green cloud computing infrastructure. *Future Generation Computer Systems*, 28 (2):371–378, 2012. (Cited on pages 146, 148, 175, 176, 184, and 187)

[103] Corentin Dupont, Thomas Schulze, Giovanni Giuliani, Andrey Somov, and Fabien Hermenier. An energy aware framework for virtual machine placement in cloud federated data centres. In *3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet*, pages 4:1–4:10, 2012. (Cited on pages 111 and 113)

[104] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis*. Cambridge, UK:Cambridge University Press, 1998. (Cited on page 116)

[105] Sourav Dutta, Sankalp Gera, Akshat Verma, and Balaji Viswanathan. SmartScale: Automatic application scaling in enterprise clouds. In *5th International Conference on Cloud Computing (CLOUD)*, pages 221–228, 2012. (Cited on pages 85, 86, 88, and 213)

[106] Rubin Ellen. Cloud federation and the intercloud. `ellenrubin.sys-con.com/node/1249746`, 2010. Last accessed in May 2014. (Cited on page 51)

[107] Google App Engine. Google App Engine post-mortem for February 24th, 2010 outage. `goo.gl/uYnQFf`, 2010. Last accessed in July 2014. (Cited on page 48)

[108] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003. (Cited on page 191)

[109] Faiza Fakhar, Barkha Javed, Raihan ur Rasool, Owais Malik, and Khurram Zulfiqar. Software level green computing for large scale systems. *Journal of Cloud Computing*, 1(1):1–17, 2012. (Cited on page 91)

[110] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz André Barroso. Power provisioning for a warehouse-sized computer. In *34th annual international symposium on Computer architecture*, pages 13–23, 2007. (Cited on page 89)

[111] Eugen Feller, Louis Rilling, and Christine Morin. Snooze: A scalable and autonomic virtual machine management framework for private clouds. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 482–489, 2012. (Cited on pages 82, 83, 87, 88, 209, 210, and 213)

[112] Ana Juan Ferrer, Francisco Hernández, Johan Tordsson, Erik Elmroth, Ahmed Ali-Eldin, Csilla Zsigri, Raül Sirvent, Jordi Guitart, Rosa M. Badia, Karim Djemame, Wolfgang Ziegler, Theo Dimitrakos, Srijith K. Nair, George Kousiouris, Kleopatra Konstanteli, Theodora Varvarigou, Benoit Hudzia, Alexander Kipp, Stefan Wesner, Marcelo Corrales, Nikolaus Forgó, Tabassum Sharif, and Craig Sheridan. OPTIMIS: A holistic approach to cloud service provisioning. *Future Generation Computer Systems*, 28(1):66–77, 2012. (Cited on pages 50, 72, 73, 74, and 210)

[113] Tiago C. Ferreto, Marco A. S. Netto, Rodrigo N. Calheiros, and César A. F. De Rose. Server consolidation with migration control for virtualized data centers. *Future Generation Computer Systems*, 27(8):1027–1034, 2011. (Cited on pages 40 and 102)

[114] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, 2002. (Cited on page 14)

[115] Roy T Fielding, J. Gettys, H. Frystyk, L. Masinter, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. `www.ietf.org/rfc/rfc2616.txt`, 1999. Last accessed in January 2014. (Cited on page 15)

[116] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000. (Cited on page 119)

[117] Brian W. Fitzpatrick and JJ Lueck. The case against data lock-in. *Communications of the ACM*, 53(11):42–46, 2010. (Cited on page 33)

[118] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972. (Cited on page 12)

[119] Carlos M. Fonseca and Peter J. Fleming. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary Computation*, 3(1):1–16, 1995. (Cited on page 153)

[120] Global Inter-Cloud Technology Forum. Use cases and functional requirements for inter-cloud computing. Technical report, Global Inter-Cloud Technology Forum, August 2010. URL `gictf.jp/doc/GICTF_Whitepaper_20100809.pdf`. Last accessed in January 2014. (Cited on page 48)

[121] I. Foster, Yong Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop*, pages 1–10, 2008. (Cited on pages 29, 31, 32, 45, and 46)

[122] Ian Foster. Globus Toolkit version 4: Software for service-oriented systems. In *International Conference on Network and Parallel Computing*, pages 2–13. Springer-Verlag, 2005. (Cited on pages 21, 22, and 35)

[123] Ian Foster and Adriana Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In *2nd International Workshop on Peer-to-Peer Systems*, pages 118–128, 2003. (Cited on pages 34 and 213)

[124] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996. (Cited on page 14)

[125] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., 1999. (Cited on page 14)

[126] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001. (Cited on pages 19, 20, and 34)

[127] Geoffrey Fox, Xiaohong Qiu, Scott Beason, Jong Choi, Jaliya Ekanayake, Thilina Gunarathne, Mina Rho, Haixu Tang, Neil Devadasan, and Gilbert Liu. Biomedical Case Studies in Data Intensive Computing. In *IEEE International Conference on Cloud Computing*, pages 2–18, 2009. (Cited on pages 125 and 127)

[128] Guilherme Galante and Luis Carlos Erpen Bona. Constructing elastic scientific applications using elasticity primitives. *13th International Conference on Computational Science and Applications*, 5:281–294, 2013. (Cited on page 60)

[129] Peter Xiang Gao, Andrew R. Curtis, Bernard Wong, and Srinivasan Keshav. It's not easy being green. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 211–222, 2012. (Cited on page 90)

[130] Jesús García-Galán, Omer Rana, Pablo Trinidad, and Antonio Ruiz-Cortés. Migrating to the cloud: a software product line based analysis. In *3rd International Conference on Cloud Computing and Services Science*, pages 416–426, 2013. (Cited on pages 148, 162, 184, 185, and 187)

[131] L. Garcés-Erice, K.W. Ross, E.W. Biersack, P.A. Felber, and G. Urvoy-Keller. Topology-centric look-up service. In *5th International Workshop on Networked Group Communications*, pages 58–69, 2003. (Cited on page 28)

[132] Simson Garfinkel. The computer utility. In Harold Abelson, editor, *Architects of the Information Society: 35 Years of the Laboratory for Computer Science at MIT*. MIT Press, 1999. (Cited on pages 11, 35, and 219)

[133] Saurabh Kumar Garg, Steve Versteeg, and Rajkumar Buyya. SMICloud: A framework for comparing and ranking cloud services. In *4th IEEE International Conference on Utility and Cloud Computing*, pages 210–218, 2011. (Cited on page 91)

[134] Vijay K. Garg, Ph.D. *Elements of Distributed Computing*. John Wiley & Sons, 2002. (Cited on page 12)

[135] Rong Ge, Xizhou Feng, Wu-chun Feng, and Kirk W. Cameron. CPU MISER: A performance-directed, run-time system for power-aware clusters. In *International Conference on Parallel Processing*, pages 18–26, 2007. (Cited on page 91)

[136] Christos Gkantsidis, Milena Mihail, and Amin Saberi. Random walks in peer-to-peer networks: Algorithms and evaluation. *Performance Evaluation*, 63(3):241–263, 2006. (Cited on pages 24 and 28)

[137] Íñigo Goiri, Ferran Julià, J. Oriol Fitó, Mario Macías, and Jordi Guitart. Supporting CPU-based Guarantees in Cloud SLAs via Resource-level QoS Metrics. *Future Generation Computer Systems*, 28(8):1295–1302, 2012. (Cited on page 55)

[138] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer*, 7(9): 34–45, 1974. (Cited on pages 12, 37, and 38)

[139] Pam Frost Gorder. Coming soon: Research in a cloud. *Computing in Science and Engineering*, 10(6):6–10, 2008. (Cited on page 35)

[140] Andrzej Goscinski and Michael Brock. Toward dynamic and attribute based publication, discovery and selection for cloud computing. *Future Generation Computer Systems*, 26(7):947–970, 2010. (Cited on page 52)

[141] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete Mathematics*, 5:287–326, 1979. (Cited on page 195)

[142] Greenpeace. Make it green: Cloud computing and its contribution to climate change. Technical report, Greenpeace International, March 2010. URL `greenpeace.org/usa/Global/usa/report/2010/3/make-it-green-cloud-computing.pdf`. Last accessed in April 2014. (Cited on pages 2 and 90)

[143] Sam Griffiths-Jones, Simon Moxon, Mhairi Marshall, Ajay Khanna, Sean R. Eddy, and Alex Bateman. Rfam: annotating non-coding RNAs in complete genomes. *Nucleic Acids Research*, 33:D121–D124, 2005. (Cited on page 136)

[144] Bernd Grobauer, Tobias Walloschek, and Elmar Stocker. Understanding cloud computing vulnerabilities. *IEEE Security and Privacy*, 9:50–57, 2011. (Cited on page 32)

[145] Nikolay Grozev and Rajkumar Buyya. Inter-Cloud architectures and application brokering: taxonomy and survey. *Software: Practice and Experience*, 44(3):369–390, 2014. (Cited on pages 48 and 49)

[146] P. H. Gum. System/370 extended architecture: Facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, 1983. (Cited on page 12)

[147] Tian Guo, Upendra Sharma, Prashant Shenoy, Timothy Wood, and Sambit Sahu. Cost-aware cloud bursting for enterprise applications. *ACM Transactions on Internet Technology*, 13(3):10:1–10:24, 2014. (Cited on page 60)

[148] Hongmu Han, Jie He, and Cuihua Zuo. A hybrid P2P overlay network for high efficient search. In *2nd IEEE International Conference on Information and Financial Engineering*, pages 241–245, 2010. (Cited on page 27)

[149] Vinay Hanumaiah and Sarma Vrudhula. Energy-efficient operation of multi-core processors by DVFS, task migration and active cooling. *IEEE Transactions on Computers*, 99, 2012. (Cited on page 91)

[150] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. SkipNet: A scalable overlay network with practical locality properties. In *4th Conference on USENIX Symposium on Internet Technologies and Systems*, pages 9–21, 2003. (Cited on page 25)

[151] Mohammad Mehedi Hassan, M.Shamim Hossain, A.M.Jehad Sarkar, and Eui-Nam Huh. Cooperative game-based distributed resource allocation in horizontal dynamic cloud federation platform. *Information Systems Frontiers*, pages 1–20, 2012. (Cited on page 52)

[152] Michael Hauben and Ronda Hauben. *Netizens: On the History and Impact of Usenet and the Internet*. Wiley-IEEE Computer Society, 1997. (Cited on page 212)

[153] Naohiro Hayashibara and Adel Cherif. Failure detectors for large-scale distributed systems. In *21st IEEE Symposium on Reliable Distributed Systems*, pages 404–409, 2002. (Cited on page 190)

[154] Brian Hayes. Cloud computing. *Communications of the ACM*, 51:9–11, 2008. (Cited on pages 1 and 15)

[155] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: A consolidation manager for clusters. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 41–50, 2009. (Cited on pages 111 and 113)

[156] Z. Hill and M. Humphrey. CSAL: A cloud storage abstraction layer to enable portable cloud applications. In *2nd International Conference on Cloud Computing Technology and Science*, pages 504–511, 2010. (Cited on page 33)

[157] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *ACM SIGOPS Operating Systems Review*, 43(3):14–26, 2009. (Cited on page 39)

[158] C. N. Hoefer and G. Karagiannis. Taxonomy of cloud computing services. In *IEEE GLOBECOM Workshops*, pages 1345–1350, 2010. (Cited on page 162)

[159] I. L. Hofacker, W. Fontana, P. F. Stadler, L. S. Bonhoeffer, M. Tacker, and P. Schuster. Fast folding and comparison of RNA secondary structures. *Chemical Monthly*, 125 (2):167–188, 1994. (Cited on page 136)

[160] Steve Hoffmann, Christian Otto, Stefan Kurtz, Cynthia M. Sharma, Philipp Khaitovich, Jörg Vogel, Peter F. Stadler, and Jörg Hackermüller. Fast mapping of short sequences with mismatches, insertions and deletions using index structures. *PLoS computational biology*, 5:e1000502, 2009. (Cited on pages 136 and 137)

[161] Q. Hofstatter, S. Zols, M. Michel, Z. Despotovic, and W. Kellerer. Chordella - a hierarchical peer-to-peer overlay implementation for heterogeneous, mobile environments. In *8th International Conference on Peer-to-Peer Computing*, pages 75–76, 2008. (Cited on pages 27 and 28)

[162] Paul Horn. Autonomic computing: IBM's perspective on the state of information technology. `www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf`, 2001. Last accessed in January 2014. (Cited on pages 16, 77, 78, and 213)

[163] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys*, 40(3):7:1–7:28, 2008. (Cited on pages 76 and 80)

[164] IBM. An architectural blueprint for autonomic computing. Technical Report Third edition, IBM, June 2005. URL www-03.ibm.com/autonomic/pdfs/ACBlueprintWhitePaperV7.pdf. Last accessed in January 2014. (Cited on pages 77 and 79)

[165] Shigeru Imai, Thomas Chestna, and Carlos A. Varela. Elastic scalable cloud computing using application-level migration. In *5th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 91–98, 2012. (Cited on pages 66, 67, 74, and 210)

[166] Anca Iordache, Christine Morin, Nikos Parlavantzas, Eugen Feller, and Pierre Riteau. Resilin: Elastic MapReduce over multiple clouds. In *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 261–268, 2013. (Cited on pages 141 and 143)

[167] Alexandru Iosup, Simon Ostermann, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed System*, 22(6):931–945, 2011. (Cited on pages 32, 147, 164, and 205)

[168] iperf3. software.es.net/iperf, 2014. Last accessed in July 2014. (Cited on pages 150 and 168)

[169] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, revised edition, 2012. (Cited on page 218)

[170] Keith R. Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J. Wasserman, and Nicholas J. Wright. Performance analysis of high performance computing applications on the Amazon Web Services cloud. In *2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 159–168, 2010. (Cited on pages 54 and 164)

[171] H. V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. BATON: A balanced tree structure for peer-to-peer networks. In *31st International Conference on Very Large Data Bases*, pages 661–672, 2005. (Cited on page 25)

[172] H.V. Jagadish, Beng-Chin Ooi, Quang Hieu Vu, Rong Zhang, and Aoying Zhou. VBI-Tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In *22nd International Conference on Data Engineering*, pages 34–34, 2006. (Cited on page 25)

[173] Pelle Jakovits and Satish Narayana Srirama. Adapting scientific applications to cloud by using distributed computing frameworks. In *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 164–167, 2013. (Cited on page 146)

[174] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3):1–36, 2007. (Cited on page 80)

[175] Xing Jin and S.-H.Gary Chan. Unstructured peer-to-peer network architectures. In Xuemin Shen, Heather Yu, John Buford, and Mursalin Akon, editors, *Handbook of Peer-to-Peer Networking*, pages 117–142. Springer, 2010. (Cited on page 24)

[176] Laurent Jourdren, Maria Bernard, Marie-Agnès Dillies, and Stéphane Le Crom. Eoulsan: A cloud computing-based framework facilitating high throughput sequencing analyses. *Bioinformatics*, 28(11):1541–1543, 2012. (Cited on pages 125 and 127)

[177] Robert H. Halstead Jr. Implementation of Multilisp: Lisp on a Multiprocessor. In *ACM Symposium on LISP and Functional Programming*, pages 9–17, 1984. (Cited on page 218)

[178] Narendra Jussien, Guillaume Rochart, and Xavier Lorca. Choco: an Open Source Java Constraint Programming Library. In *Workshop on Open-Source Software for Integer and Contraint Programming*, pages 1–10, 2008. (Cited on pages 6, 149, 164, 188, 216, and 218)

[179] Gideon Juve, Mats Rynge, Ewa Deelman, Jens-S. Vockler, and G. Bruce Berriman. Comparing FutureGrid, Amazon EC2, and Open Science Grid for Scientific Workflows. *Computing in Science & Engineering*, 15(4):20–29, 2013. (Cited on pages 2, 129, 141, and 146)

[180] Steffen Kächele and Franz J. Hauck. Component-based scalability for cloud applications. In *3rd International Workshop on Cloud Data and Platforms*, pages 19–24, 2013. (Cited on pages 60 and 74)

[181] Yoshiaki Kakuda, Hideki Yukitomo, Shinji Kusumoto, and Tohru Kikuno. Scientific computing in the cloud. *IEEE Design & Test*, 12(3):34–43, 2010. (Cited on page 35)

[182] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990. URL `www.sei.cmu.edu/reports/90tr021.pdf`. Last accessed in June 2014. (Cited on pages 147 and 154)

[183] Krishna Kant and Prasant Mohapatra. Internet data centers. *IEEE Computer*, 37: 35–37, 2004. (Cited on page 89)

[184] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *29th Annual ACM Symposium on Theory of Computing*, pages 654–663, 1997. (Cited on page 25)

[185] Lori M. Kaufman. Can a trusted environment provide security? *IEEE Security and Privacy*, 8(1):50–52, 2010. (Cited on pages 33 and 34)

[186] K. Keahey, I. Foster, T. Freeman, and X. Zhang. Virtual workspaces: Achieving quality of service and quality of life in the grid. *Scientific Programming*, 13(4): 265–275, 2005. (Cited on pages 55 and 217)

[187] Katarzyna Keahey and Tim Freeman. Contextualization: Providing one-click virtual clusters. In *4th IEEE International Conference on eScience*, pages 301–308, 2008. (Cited on page 59)

[188] Katarzyna Keahey, Mauricio Tsugawa, Andrea Matsunaga, and Jose Fortes. Sky computing. *IEEE Internet Computing*, 13(5):43–51, 2009. (Cited on page 51)

[189] Gabor Kecskemeti, Attila Kertesz, Attila Marosi, and Peter Kacsuk. Interoperable resource management for establishing federated clouds. In Ivona Brandic, Massimo Villari, and Francesco Tusa, editors, *Achieving Federated and Self-Manageable Cloud Infrastructures: Theory and Practice*, pages 18–35. IGI Global, 2012. (Cited on page 50)

[190] Alexander Keller and Heiko Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11:57–81, 2003. (Cited on page 42)

[191] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003. (Cited on pages 76, 77, 78, 79, and 80)

[192] Anne-Marie Kermarrec and Peter Triantafillou. XL peer-to-peer pub/sub systems. *ACM Computing Surveys*, 46(2):16:1–16:45, 2013. (Cited on page 11)

[193] Hamzeh Khazaei, Jelena Misic, and Vojislav B. Misic. Performance Analysis of Cloud Computing Centers Using M/G/m/m+r Queuing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 23(5):936–943, 2012. (Cited on pages 91 and 111)

[194] Hyunjoo Kim and Manish Parashar. *CometCloud: An Autonomic Cloud Engine*, pages 275–297. John Wiley & Sons, Inc., 2011. (Cited on pages 70, 71, 74, and 210)

[195] Kyong Hoon Kim, Wan Yeon Lee, Jong Kim, and Rajkumar Buyya. SLA-based scheduling of bag-of-tasks applications on power-aware cluster systems. *Transactions on Information and Systems*, E93-D(12):3194–3201, 2010. (Cited on pages 111 and 113)

[196] Alexander Kipp, Tao Jiang, Mariagrazia Fugini, and Ioan Salomie. Layered green performance indicators. *Future Generation Computer Systems*, 28(2):478–489, 2012. (Cited on page 98)

[197] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. In *PLinux Symposium*, volume 1, pages 225–230, 2007. (Cited on page 38)

[198] Dzmitry Kliazovich, Pascal Bouvry, and Samee Ullah Khan. Greencloud: A packet-level simulator of energy-aware cloud computing data centers. In *IEEE Global Telecommunications Conference*, pages 1–5, 2010. (Cited on page 32)

[199] Sonja Klingert, Thomas Schulze, and Christian Bunse. GreenSLAs for the energy-efficient management of data centres. In *2nd International Conference on Energy-Efficient Computing and Networking*, pages 21–30, 2011. (Cited on page 41)

[200] Alexander V. Konstantinou, Tamar Eilam, Michael Kalantar, Alexander A. Totok, William Arnold, and Edward Snible. An architecture for virtual solution composition and deployment in infrastructure clouds. In *3rd International Workshop on Virtualization Technologies in Distributed Computing*, pages 9–18, 2009. (Cited on pages 146, 177, 178, 184, and 187)

[201] Jonathan G Koomey. Estimating total power consumption by servers in the U.S. and the world. Technical report, Lawrence Berkeley National Laboratory and Consulting Professor, Stanford University, February 2007. URL `hightech.lbl.gov/documents/DATA_CENTERS/svrpwrusecompletefinal.pdf`. Last accessed in April 2014. (Cited on pages 2 and 89)

[202] Jonathan G Koomey. Growth in data center electricity use 2005 to 2010. *Oakland, CA: Analytics Press*, 2011. (Cited on page 2)

[203] Pawel Koperek and Wlodzimierz Funika. Dynamic business metrics-driven resource provisioning in cloud environments. In *9th International Conference on Parallel Processing and Applied Mathematics - Volume Part II*, pages 171–180, 2012. (Cited on page 55)

[204] Hugo H. Kramer, Vinicius Petrucci, Anand Subramanian, and Eduardo Uchoa. A column generation approach for power-aware optimization of virtualized heterogeneous server clusters. *Computers and Industrial Engineering*, 63(3):652–662, 2012. (Cited on page 91)

[205] William T.C. Kramer, John M. Shalf, and Erich Strohmaier. The NERSC sustained system performance (SSP) metric. In *9AD*, pages 6–12, 2005. (Cited on pages 55 and 56)

[206] Rouven Krebs, Christof Momm, and Samuel Kounev. Metrics and techniques for quantifying performance isolation in cloud environments. In *8th International ACM SIGSOFT Conference on Quality of Software Architectures*, pages 91–100, 2012. (Cited on page 55)

[207] Rakesh Kumar. Important power, cooling and green it concerns. Technical report, Gartner Report, 2007. URL `gartner.com/doc/500296/important-power-cooling-green-it`. Last accessed in April 2014. (Cited on page 89)

[208] Shun Kit Kwan and J.K. Muppala. Bag-of-tasks applications scheduling on volunteer desktop grids with adaptive information dissemination. In *IEEE 35th Conference on Local Computer Networks*, pages 544–551, 2010. (Cited on page 27)

[209] Bill Laing. Windows Azure service disruption update. `azure.microsoft.com/blog/2012/02/29/windows-azure-service-disruption-update/`, 2012. Last accessed in July 2014. (Cited on page 48)

[210] Klaus-Dieter Lange. Identifying shades of green: The SPECpower benchmarks. *IEEE Computer*, 42:95–97, 2009. (Cited on pages 93 and 94)

[211] Jason Lango. Toward software-defined SLAs. *ACM Queue*, 11(11):20:20–20:31, 2013. (Cited on page 150)

[212] Young Choon Lee and Albert Y. Zomaya. Energy efficient utilization of resources in cloud computing systems. *The Journal of Supercomputing*, pages 1–13, 2010. (Cited on pages 103, 105, 112, and 113)

[213] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. Server-level power control. In *Fourth Autonomic Computing*, 2007. (Cited on page 89)

[214] Alessandro Ferreira Leite and Alba Cristina Magalhães Alves de Melo. Executing a biological sequence comparison application on a federated cloud environment. In *19th International Conference on High Performance Computing*, pages 1–9, 2012. (Cited on pages 5, 116, 127, and 214)

[215] Alessandro Ferreira Leite and Alba Cristina Magalhães Alves de Melo. Energy-aware multi-agent server consolidation in federated clouds. In Mazin Yousif and Lutz Schubert, editors, *Cloud Computing*, volume 112 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 72–81. 2013. (Cited on pages 4, 5, 102, 103, 113, and 214)

[216] Alessandro Ferreira Leite, Hammurabi Chagas Mendes, Li Weigang, Alba Cristina Magalhães Alves Melo, and Azzedine Boukerche. An architecture for P2P bag-of-tasks execution with multiple task allocation policies in desktop grids. *Cluster Computing*, 15(4):351–361, 2012. (Cited on pages 196 and 217)

[217] Alessandro Ferreira Leite, Claude Tadonki, Christine Eisenbeis, Tainá Raiol, Maria Emilia M. T. Walter, and Alba Cristina Magalhães Alves de Melo. Excalibur: An autonomic cloud architecture for executing parallel applications. In *4th International Workshop on Cloud Data and Platforms*, pages 2:1–2:6, 2014. (Cited on pages 5, 6, 129, 143, and 215)

[218] Philipp Leitner, Zabolotnyi Rostyslav, Alessio Gambi, and Schahram Dustdar. A framework and middleware for application-level cloud bursting on top of infrastructure-as-a-service clouds. In *6th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 163–170, 2013. (Cited on pages 68, 74, and 210)

[219] Feng Li, Beng Chin Ooi, M. Tamer Özsu, and Sai Wu. Distributed data management using MapReduce. *ACM Computing Surveys*, 46(3):31:1–31:42, 2014. (Cited on page 45)

[220] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25 (16):2078–2079, 8 2009. (Cited on page 136)

[221] Mei Li, Wang-Chien Lee, and Anand Sivasubramaniam. DPTree: A balanced tree based indexing framework for peer-to-peer systems. In *International Conference on Network Protocols*, pages 12–21, 2006. (Cited on page 25)

[222] Qing Li, Hongkun Li, Paul Russell Jr., Zhuo Chen, and Chonggang Wang. CA-P2P: Context-aware proximity-based peer-to-peer wireless communications. *IEEE Communications Magazine*, 52(6):32–41, 2014. (Cited on pages 199 and 200)

[223] X. Li and J Wu. Searching techniques in peer-to-peer networks. In *Handbook of Theoretical and Algorithmic Aspects of Ad Hoc, Sensor, and Peer-to-Peer Networks*, pages 613–642. Averbach, 2006. (Cited on page 25)

[224] Zheng Li, Liam O'Brien, He Zhang, and Rainbow Cai. Boosting metrics for cloud services evaluation – the last mile of using benchmark suites. In *27th International Conference on Advanced Information Networking and Applications*, pages 381–388, 2013. (Cited on page 55)

[225] Xiangke Liao, Liquan Xiao, Canqun Yang, and Yutong Lu. MilkyWay-2 supercomputer: system and application. *Frontiers of Computer Science*, 8(3):345–356, 2014. (Cited on page 18)

[226] J. C. R. Licklider and Robert W. Taylor. The computer as a communication device. *Science and Technology*, 76:21–31, 1968. (Cited on page 11)

[227] LINPACK. `netlib.org/linpack`, 2012. Last accessed in July 2014. (Cited on pages 93, 164, and 168)

[228] Meirong Liu, Timo Koskela, Zhonghong Ou, Jiehan Zhou, Jukka Riekki, and Mika Ylianttila. Super-peer-based coordinated service provision. *Journal of Network and Computer Applications*, 34(4):1210–1224, 2011. (Cited on page 27)

[229] Yongchao Liu, Douglas L Maskell, and Bertil Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Bioinformatics*, 2, 2009. (Cited on pages 115 and 127)

[230] Zhenhua Liu, Yuan Chen, Cullen Bash, Adam Wierman, Daniel Gmach, Zhikui Wang, Manish Marwah, and Chris Hyser. Renewable and cooling aware workload management for sustainable data centers. In *12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, pages 175–186, 2012. (Cited on page 90)

[231] Boon Thau Loo, Ryan Huebsch, Ion Stoica, and Joseph M. Hellerstein. The case for a hybrid P2P search infrastructure. In *International workshop on Peer-To-Peer Systems*, pages 141–150, 2004. (Cited on pages 26 and 27)

[232] Yung-Hsiang Lu, Qinru Qiu, Ali R. Butt, and Kirk W. Cameron. End-to-end energy management. *IEEE Computer*, 44(11):75–77, 2011. (Cited on page 93)

[233] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) benchmark suite. In *ACM/IEEE conference on Supercomputing*, 2006. (Cited on page 94)

[234] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *16th International Conference on Supercomputing*, pages 84–95, 2002. (Cited on pages 24 and 28)

[235] Qin Lv, Sylvia Ratnasamy, and Scott Shenker. Can heterogeneity make Gnutella scalable? In *International workshop on Peer-To-Peer Systems*, volume 2429, pages 94–103, 2002. (Cited on page 24)

[236] Evaristus Mainsah. Autonomic computing: the next era of computing. *Electronics Communication Engineering Journal*, 14(1):2–3, 2002. (Cited on pages 77 and 80)

[237] Madhavi Maiya, Sai Dasari, Ravi Yadav, Sandhya Shivaprasad, and Dejan Milojicic. Quantifying manageability of cloud platforms. In *5th IEEE International Conference on Cloud Computing*, pages 993–995, 2012. (Cited on page 55)

[238] Attila Csaba Marosi, Gabor Kecskemeti, Attila Kertesz, and Peter Kacsuk. FCM: an architecture for integrating IaaS cloud systems. In *2nd International Conference on Cloud Computing, GRIDs, and Virtualization*, pages 7–12, 2011. (Cited on page 60)

[239] Moreno Marzolla, Ozalp Babaoglu, and Fabio Panzieri. Server consolidation in clouds through gossiping. In *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, pages 1–6. IEEE Computer, 2011. (Cited on pages 40, 80, 88, and 213)

[240] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia distributed monitoring system: Design, implementation and experience. *Parallel Computing*, 30 (7):817–840, 2003. (Cited on page 53)

[241] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *First International Workshop on Peer-to-Peer Systems*, pages 53–65, 2002. (Cited on page 25)

[242] Piyush Mehrotra, Jahed Djomehri, Steve Heistand, Robert Hood, Haoqiang Jin, Arthur Lazanoff, Subhash Saini, and Rupak Biswas. Performance evaluation of Amazon EC2 for NASA HPC applications. In *3rd Workshop on Scientific Cloud Computing Date*, pages 41–50, 2012. (Cited on page 164)

[243] Peter Mell and Timothy Grance. The NIST definition of Cloud Computing, National Institute of Standards and Technology. Technical Report SP800-145, NIST Information Technology Laboratory, 2011. URL `csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`. Last accessed in January 2014. (Cited on pages 29, 30, 45, 46, and 47)

[244] Alexandre Mello Ferreira, Kyriakos Kritikos, and Barbara Pernici. Energy-aware design of service-based applications. In *Service-Oriented Computing*, volume 5900, pages 99–114. Springer-Verlag, 2009. (Cited on page 90)

[245] Víctor Méndez Muñoz, Adrian Casajús Ramo, Víctor Fernández Albor, Ricardo Graciani Diaz, and Gonzalo Merino Arévalo. Rafhyc: An architecture for constructing resilient services on federated hybrid clouds. *Journal of Grid Computing*, 11(4): 753–770, 2013. (Cited on pages 66, 67, and 74)

[246] Kaisa Miettinen. Introduction to multiobjective optimization: Noninteractive approaches. In Jürgen Branke, Kalyanmoy Deb, Kaisa Miettinen, and Roman Słowiński, editors, *Multiobjective Optimization*, volume 5252, pages 1–26. Springer Berlin Heidelberg, 2008. (Cited on page 151)

[247] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *Workshop on Principles of Engineering Service Oriented Systems*, pages 18–25, 2009. (Cited on pages 180, 181, 184, and 187)

[248] Dejan Milojicic, Ignacio M. Llorente, and Ruben S. Montero. OpenNebula: A cloud management tool. *IEEE Internet Computing*, 15(2):11–14, 2011. (Cited on pages 54, 55, 60, and 217)

[249] Laurl Minas and Brad Ellison. *Energy Efficiency for Information Technology: How to Reduce Power Consumption in Servers and Data Centers*. Intel Press, 2009. (Cited on page 214)

[250] Alan Mislove and Peter Druschel. Providing administrative control and autonomy in structured peer-to-peer overlays. In *International workshop on Peer-To-Peer Systems*, pages 162–172, 2004. (Cited on page 27)

[251] Isi Mitrani. Service center trade-offs between customer impatience and power consumption. *Performance Evaluation*, 68(11):1222–1231, 2011. (Cited on pages 91 and 111)

[252] Anirban Mondal, Yi Lifu, and Masaru Kitsuregawa. P2PR-Tree: An r-tree-based spatial index for peer-to-peer environments. In *International Conference on Current Trends in Database Technology*, pages 516–525. Springer-Verlag, 2004. (Cited on page 25)

[253] Rafael Moreno-Vozmediano, Ruben Montero, and Ignacio Llorente. IaaS cloud architecture: From virtualized datacenters to federated cloud infrastructures. *Computer*, 45(12):65–72, 2012. (Cited on page 49)

[254] Mosaic: The First Global Web Browser. `www.livinginternet.com/w/wi_mosaic.htm`, 2014. Last accessed in January 2014. (Cited on page 14)

[255] David Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 1st edition, 2004. (Cited on pages 117 and 122)

[256] Victor Ion Munteanu, Teodor-Florin Fortis, and Adrian Copie. Building a cloud governance bus. *International Journal of Computers, Communications & Control*, 7 (5):900–906, 2012. (Cited on page 51)

[257] Mihir Nanavati, Patrick Colp, Bill Aiello, and Andrew Warfield. Cloud security: A gathering storm. *Communications of the ACM*, 57(5):70–79, 2014. (Cited on pages 33 and 54)

[258] Susanta Nanda and Tzi cker Chiueh. A survey of virtualization technologies. Technical Report TR–179, Stony Brook University, February 2005. URL www.ecsl.cs.sunysb.edu/tr/TR179.pdf. Last accessed in February 2014. (Cited on page 37)

[259] Eric Nawrocki and Sean Eddy. INFERNAL user's guide. selab.janelia.org/software/infernal/Userguide.pdf, 2014. Last accessed in May 2014. (Cited on page 137)

[260] Eric P. Nawrocki, Diana L. Kolbe, and Sean R. Eddy. Infernal 1.0: inference of RNA alignments. *Bioinformatics*, 25(10):1335–1337, 2009. (Cited on page 136)

[261] Mahdi Noorian, Hamidreza Pooshfam, Zeinab Noorian, and Rosni Abdullah. Performance enhancement of Smith-Waterman algorithm using hybrid model: comparing the MPI and hybrid programming paradigm on SMP clusters. In *IEEE International Conference on Systems, Man and Cybernetics*, pages 492–497, 2009. (Cited on pages 115 and 127)

[262] Henrik Nordberg, Karan Bhatia, Kai Wang, and Zhong Wang. BioPig: a Hadoop-based analytic toolkit for large-scale sequence data. *Bioinformatics*, 29(23):3014–3019, 2013. (Cited on pages 129, 134, 141, and 143)

[263] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131, 2009. (Cited on pages 55 and 217)

[264] Anne-Cecile Orgerie, Laurent Lefevre, and Jean-Patrick Gelas. Demystifying energy consumption in grids and clouds. In *International Conference on Green Computing*, pages 335–342, 2010. (Cited on page 90)

[265] Anne-Cecile Orgerie, Marcos Dias de Assuncao, and Laurent Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Computing Surveys*, 46(4):47:1–47:31, 2014. (Cited on page 90)

[266] Anne-Cecile Orgerie, Marcos Dias de Assuncao, and Laurent Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Computing Surveys*, 46(4):47:1–47:31, 2014. (Cited on page 102)

[267] Simon Ostermann, Alexandru Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. A performance analysis of EC2 cloud computing services for scientific computing. In *Cloud Computing*, pages 115–131. Springer, 2010. (Cited on pages 147, 164, and 205)

[268] Andrzej Osyczka. *Multicriteria optimization for engineering design*, pages 193–227. Academic Press, 1985. (Cited on page 151)

[269] Zhonghong Ou, Hao Zhuang, Andrey Lukyanenko, Jukka K. Nurminen, Pan Hui, Vladimir Mazalov, and Antti Ylä-Jääski. Is the same instance type created equal? exploiting heterogeneity of public clouds. *IEEE Transactions on Cloud Computing*, 1(2):201–214, 2013. (Cited on pages 17, 150, and 164)

[270] Harris Papadakis, Paolo Trunfio, Domenico Talia, and Paraskevi Fragopoulou. Design and implementation of a hybrid P2P-based grid resource discovery system. In *Making Grids Work*, pages 89–101. Springer, 2008. (Cited on page 27)

[271] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40 (11):38–45, 2007. (Cited on page 65)

[272] Giuseppe Papuzzo and Giandomenico Spezzano. Autonomic management of workflows on hybrid grid-cloud infrastructure. In *7th International Conference on Network and Services Management*, pages 230–233. International Federation for Information Processing, 2011. (Cited on pages 80, 81, 88, and 213)

[273] Fawaz Paraiso, Nicolas Haderer, Philippe Merle, Romain Rouvoy, and Lionel Seinturier. A federated multi-cloud PaaS infrastructure. In *5th International Conference on Cloud Computing*, pages 392–399, 2012. (Cited on pages 63 and 65)

[274] Przemyslaw Pawluk, Bradley Simmons, Michael Smit, Marin Litoiu, and Serge Mankovski. Introducing STRATOS: A cloud broker service. In *5th IEEE International Conference on Cloud Computing*, pages 891–898, 2012. (Cited on pages 65, 66, and 74)

[275] Dana Petcu. Portability and interoperability between clouds: Challenges and case study. In *4th European Conference on Towards a Service-based Internet*, pages 62–74, 2011. (Cited on pages 33, 50, and 51)

[276] Dana Petcu. Multi-cloud: Expectations and current approaches. In *International Workshop on Multi-cloud Applications and Federated Clouds*, pages 1–6, 2013. (Cited on pages 33 and 52)

[277] Dana Petcu. Consuming resources and services from multiple clouds. *Journal of Grid Computing*, pages 1–25, 2014. (Cited on pages 49, 50, 51, 190, and 217)

[278] Dana Petcu, Ciprian Craciun, and Massimiliano Rak. Towards a cross platform cloud API. In *1st International Conference on Cloud Computing and Services Science*, pages 166–169, 2011. (Cited on pages 50, 63, 74, and 210)

[279] Dana Petcu, BeniaminoDi Martino, Salvatore Venticinque, Massimiliano Rak, Tamás Máhr, GorkaEsnal Lopez, Fabrice Brito, Roberto Cossu, Miha Stopar, Svatopluk Šperka, and Vlado Stankovski. Experiences in building a mOSAIC of clouds. *Journal of Cloud Computing*, 2(1), 2013. (Cited on pages 63 and 64)

[280] Guillaume Pierre and Corina Stratan. ConPaaS: a platform for hosting elastic cloud applications. *IEEE Internet Computing*, 16(5):88–92, 2012. (Cited on pages 60 and 72)

[281] Eduardo Pinheiro, Ricardo Bianchini, Enrique V. Carrera, and Taliver Heath. Load balancing and unbalancing for power and performance in cluster-based systems. *Workshop on Compilers and Operating Systems for Low Power*, pages 182–195, 2001. (Cited on page 91)

[282] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, 2005. (Cited on page 147)

[283] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974. (Cited on page 38)

[284] R. Prodan and S. Ostermann. A survey and taxonomy of infrastructure as a service and web hosting cloud providers. In *10th IEEE/ACM International Conference on Grid Computing*, pages 17–25, 2009. (Cited on pages 162 and 164)

[285] William Pugh. Skip Lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990. (Cited on page 25)

[286] Flavien Quesnel, Adrien Lèbre, and Mario Südholt. Cooperative and reactive scheduling in large-scale virtualized platforms with DVMS. *Concurrency and Computation: Practice and Experience*, 25(12):1643–1655, 2013. (Cited on pages 83, 84, and 88)

[287] Clément Quinton, Nicolas Haderer, Romain Rouvoy, and Laurence Duchien. Towards multi-cloud configurations using feature models and ontologies. In *International Workshop on Multi-cloud Applications and Federated Clouds*, pages 21–26, 2013. (Cited on pages 179, 180, 184, and 187)

[288] Clément Quinton, Daniel Romero, and Laurence Duchien. Automated selection and configuration of cloud environments using software product lines principles. In *7th IEEE International Conference on Cloud Computing*, 2014. (Cited on pages 148, 179, 184, and 187)

[289] Asfandyar Qureshi, Rick Weber, Hari Balakrishnan, John Guttag, and Bruce Maggs. Cutting the electric bill for internet-scale systems. In *ACM SIGCOMM conference on Data communication*, pages 123–134, 2009. (Cited on page 113)

[290] S. Rajko and S. Aluru. Space and time optimal parallel sequence alignments. *IEEE Transactions on Parallel and Distributed Systems*, 15(12):1070–1081, 2004. (Cited on pages 115 and 127)

[291] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, 2001. (Cited on page 25)

[292] Mikael Ricknäs. Microsoft's Windows Azure cloud hit by worldwide management interuption. `pcworld.com/article/2059901/microsofts-windows-azure-cloud-hit-by-worldwide-management-interuption.html`, 2013. Last accessed in July 2014. (Cited on page 48)

[293] Matthias Riebisch, Kai Matthias, Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with UML multiplicities. In *6th World Conference on Integrated Design and Process Technology*, volume 50, pages 1–7, 2002. (Cited on page 156)

[294] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. JouleSort: a balanced energy-efficiency benchmark. In *ACM SIGMOD international conference on Management of Data*, pages 365–376, 2007. (Cited on page 94)

[295] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, Christos Kozyrakis, and Justin Meza. Models and metrics to enable energy-efficiency optimizations. *IEEE Computer*, 40(12):39–48, 2007. (Cited on page 93)

[296] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I.M. 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*, 53(4):4:1–4:11, 2009. (Cited on pages 49, 68, 69, 74, and 210)

[297] Benny Rochwerger, David Breitgand, Amir Epstein, David Hadas, Irit Loy, Kenneth Nagin, Johan Tordsson, Carmelo Ragusa, Massimo Villari, Stuart Clayman, Eliezer Levy, Alessandro Maraschini, Philippe Massonet, Henar Munoz, and Giovanni Tofetti. Reservoir - when one cloud is not enough. *Computer*, 44(3):44–51, 2011. (Cited on page 51)

[298] Ivan Rodero, Hariharasudhan Viswanathan, Eun Kyung Lee, Marc Gamell, Dario Pompili, and Manish Parashar. Energy-efficient thermal-aware autonomic management of virtualized HPC cloud infrastructure. *Journal of Grid Computing*, 10(3): 447–473, 2012. (Cited on pages 85, 88, and 213)

[299] Luis Rodero-Merino, Luis M. Vaquero, Victor Gil, Fermín Galán, Javier Fontán, Rubén S. Montero, and Ignacio M. Llorente. From infrastructure delivery to service management in clouds. *Future Generation Computer Systems*, 26(8):1226–1240, 2010. (Cited on pages 60, 61, and 74)

[300] Paolo Romano, Luis Rodrigues, Nuno Carvalho, and Joäo Cachopo. Cloud-TM: Harnessing the cloud with distributed transactional memories. *ACM SIGOPS Operating Systems Review*, 44(2):1–6, 2010. (Cited on pages 61, 74, and 210)

[301] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *IEEE Computer*, 38:39–47, 2005. (Cited on pages 13 and 38)

[302] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, 2001. (Cited on page 25)

[303] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A simple load balancing scheme for task allocation in parallel machines. In *3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, 1991. (Cited on page 218)

[304] Stefan T. Ruehl and Urs Andelfinger. Applying software product lines to create customizable software-as-a-service applications. In *15th International Software*

Product Line Conference, pages 16:1–16:4, 2011. (Cited on pages 148, 180, 184, and 187)

[305] Stefan T. Ruehl, Urs Andelfinger, Andreas Rausch, and Stephan A. W. Verclas. Toward realization of deployment variability for software-as-a-service applications. In *IEEE 5th International Conference on Cloud Computing*, pages 622–629, 2012. (Cited on pages 148, 180, 182, 184, and 187)

[306] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 3rd edition, 2009. (Cited on page 79)

[307] A. Samba. Logical data models for cloud computing architectures. *IT Professional*, 14(1):19–26, 2012. (Cited on page 1)

[308] Thomas Sandholm and Jarek Gawor. Globus Toolkit 3 core-a grid service container framework. `toolkit.globus.org/toolkit/3.0/ogsa/docs/gt3_core.pdf`, July 2003. Last accessed in January 2014. (Cited on pages 20 and 21)

[309] Sriram Sankar and Kushagra Vaid. Addressing the stranded power problem in datacenters using storage workload characterization. In *1st Joint WOSP/SIPEW International Conference on Performance engineering*, pages 217–222, 2010. (Cited on page 94)

[310] Sriya Santhanam, Pradheep Elango, Andrea Arpaci-Dusseau, and Miron Livny. Deploying virtual machines as sandboxes for the grid. In *2nd Conference on Real, Large Distributed Systems*, pages 7–12, 2005. (Cited on page 213)

[311] Constantine Sapuntzakis and Monica S. Lam. Virtual appliances in the collective: A road to hassle-free computing. In *9th Conference on Hot Topics in Operating Systems*, pages 55–60, 2003. (Cited on page 53)

[312] Yoshikazu Sawaragi, Hirotaka Nakayama, and Tetsuzo Tanino. *Theory of Multiobjective Optimization*. Academic Press, 1985. (Cited on page 152)

[313] Julia Schroeter, Peter Mucha, Marcel Muth, Kay Jugel, and Malte Lochau. Dynamic configuration management of cloud-based applications. In *16th International Software Product Line Conference*, pages 171–178, 2012. (Cited on pages 148, 182, 183, 184, and 187)

[314] Greg Schulz. *The Green and Virtual Data Center*. CRC Press, 2009. (Cited on page 93)

[315] André Schumacher, Luca Pireddu, Matti Niemenmaa, Aleksi Kallio, Eija Korpelainen, Gianluigi Zanetti, and Keijo Heljanko. SeqPig: simple and scalable scripting for large sequencing data sets in Hadoop. *Bioinformatics*, 30(1):119–120, 2014. (Cited on pages 129, 135, and 143)

[316] Glauber Scorsatto and Alba Cristina Magalhães Alves de Melo. GrAMoS: A flexible service for WS-agreement monitoring in grid environments. In *14th international Euro-Par conference on Parallel Processing*, pages 534–543, 2008. (Cited on page 41)

[317] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *IEEE International Conference on Services Computing*, pages 268–275, 2009. (Cited on pages 74 and 210)

[318] Jin Shao and Qianxiang Wang. A performance guarantee approach for cloud applications based on monitoring. In *35th Annual Computer Software and Applications Conference Workshops*, pages 25–30, 2011. (Cited on page 55)

[319] Yuxiang Shi, Xiaohong Jiang, and Kejiang Ye. An energy-efficient scheme for cloud resource provisioning based on CloudSim. In *IEEE International Conference on Cluster Computing*, pages 595–599, 2011. (Cited on pages 106 and 214)

[320] Jacopo Silvestro, Daniele Canavese, Emanuele Cesena, and Paolo Smiraglia. A unified ontology for the virtualization domain. In *Confederated International Conference on On the Move to Meaningful Internet Systems*, pages 617–624, 2011. (Cited on page 162)

[321] Kwang Mong Sim. Agent-based cloud computing. *IEEE Transactions on Services Computing*, 5(4):564–577, 2012. (Cited on pages 83, 84, and 88)

[322] G. Sinevriotis and T. Stouraitis. A novel list-scheduling algorithm for the low-energy program execution. In *IEEE International Symposium on Circuits and Systems*, volume 4, pages IV–97–IV–100, 2002. (Cited on page 91)

[323] A. Singh and Ling Liu. A hybrid topology architecture for P2P systems. In *13th International Conference on Computer Communications and Networks*, pages 475–480, 2004. (Cited on pages 26 and 28)

[324] Rahul Singh, David Irwin, Prashant Shenoy, and K. K. Ramakrishnan. Yank: Enabling green data centers to pull the plug. In *10th USENIX Conference on Networked Systems Design and Implementation*, pages 143–156, 2013. (Cited on pages 92 and 113)

[325] Dinkar Sitaram and Geetha Manjunath. *Moving To The Cloud: Developing Apps in the New World of Cloud Computing*. Syngress Publishing, 2011. (Cited on pages 35 and 37)

[326] Larry Smarr and Charles E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992. (Cited on pages 4, 14, and 219)

[327] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981. (Cited on pages 115 and 117)

[328] Borja Sotomayor, Ruben S. Montero, Ignacio M. Llorente, and Ian Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing*, 13(5):14–22, 2009. (Cited on pages 47 and 55)

[329] Ilango Sriram. SPECI, a simulation tool exploring cloud-scale data centres. In *1st International Conference on Cloud Computing*, pages 381–392, 2009. (Cited on page 32)

[330] Edward Stanford. Environmental trends and opportunities for computer system power delivery. In *20th International Symposium on Power Semiconductor Devices and IC's*, pages 1–3, 2008. (Cited on pages 2 and 89)

[331] John R. Stanley, Kenneth G. Brill, and Jonathan Koomey. Four metrics define data center 'greenness': Enabling users to quantify energy consumption initiatives for environmental sustainability and bottom line profitability. Technical report, Uptime Institute, 2007. URL `uptimeinstitute.org/component/docman/doc_download/16-four-metrics-define-data-center-greenness`. Last accessed in April 2014. (Cited on pages 95, 96, and 97)

[332] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149–160, 2001. (Cited on pages 25, 26, 27, and 196)

[333] Balaji Subramaniam and Wu-chun Feng. The green index: A metric for evaluating system-wide energy efficiency in hpc systems. In *IEEE 26th International on Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 1007–1013, 2012. (Cited on page 93)

[334] Chandrasekar Subramanian, Arunchandar Vasan, and Anand Sivasubramaniam. Reducing data center power with server consolidation: Approximation and evaluation. In *International Conference on High Performance Computing*, pages 1–10, 2010. (Cited on page 102)

[335] Francis Sullivan. Cloud computing for the sciences. *Computing in Science and Engineering*, 11(4):10–11, 2009. (Cited on page 35)

[336] Xin Sun, Yong Tian, Yushu Liu, and Yue He. An unstructured P2P network model for efficient resource discovery. In *First International Conference on the Applications of Digital Information and Web Technologies*, pages 156–161, 2008. (Cited on page 27)

[337] Vaibhav Sundriyal, Masha Sosonkina, Fang Liu, and Michael W. Schmidt. Dynamic frequency scaling and energy saving in quantum chemistry applications. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pages 837–845, 2011. (Cited on page 91)

[338] Hassan Takabi, James B. D. Joshi, and Gail-Joon Ahn. Security and privacy challenges in cloud computing environments. *IEEE Security and Privacy*, 8(6):24–31, 2010. (Cited on pages 32 and 33)

[339] Le Nhan Tam, Gerson Sunyé, and Jean-Marc Jezequel. A model-based approach for optimizing power consumption of IaaS. In *2nd Symposium on Network Cloud Computing and Applications*, pages 31–39, 2012. (Cited on pages 148, 175, 184, and 187)

[340] Le Nhan Tam, Gerson Sunyé, and Jean-Marc Jézéquel. A model-driven approach for virtual machine image provisioning in cloud computing. In *1st European Conference*

on Service-Oriented and Cloud Computing, pages 107–121, 2012. (Cited on pages 148, 175, 176, 184, and 187)

[341] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, 2nd edition, 2006. (Cited on pages 11, 12, and 17)

[342] Peiyi Tang and Pen-Chung Yew. Processor self-scheduling for multiple-nested parallel loops. In *International Conference on Parallel Processing*, pages 528–535, 1986. (Cited on pages 195 and 201)

[343] Andrei Tchernykh, Johnatan E. Pecero, Aritz Barrondo, and Elisa Schaeffer. Adaptive energy efficient scheduling in peer-to-peer desktop grids. *Future Generation Computer Systems*, 2013. (Cited on page 91)

[344] Ralf Teckelmann, Christoph Reich, and Anthony Sulistio. Mapping of cloud standards to the taxonomy of interoperability in IaaS. In *3rd International Conference on Cloud Computing Technology and Science*, pages 522–526, 2011. (Cited on page 54)

[345] Thomas Thum, Christian Kastner, Sebastian Erdweg, and Norbert Siegmund. Abstract features in feature modeling. In *15th International Software Product Line Conference*, pages 191–200, 2011. (Cited on page 155)

[346] Adel Nadjaran Toosi, Rodrigo N. Calheiros, and Rajkumar Buyya. Interconnected cloud computing environments: Challenges, taxonomy, and survey. *ACM Computing Surveys*, 47(1):7:1–7:47, 2014. (Cited on page 146)

[347] Joe Touch. Overlay networks. *Computer Networks*, 36(2–3):115–116, 2001. (Cited on page 23)

[348] Pablo Trinidad, David Benavides, Antonio Ruiz-Cortés, Sergio Segura, and Alberto Jimenez. FAMA Framework. In *12th International Software Product Line Conference*, pages 359–359, 2008. (Cited on page 184)

[349] P. Trunfio, D. Talia, H. Papadakis, P. Fragopoulou, M. Mordacchini, M. Pennanen, K. Popov, V. Vlassov, and S. Haridi. Peer-to-peer resource discovery in grids: Models and systems. *Future Generation Computer Systems*, 23(7):864–878, 2007. (Cited on pages 19 and 29)

[350] Dimitrios Tsoumakos and Nick Roussopoulos. A comparison of peer-to-peer search methods. In *6th International Workhop on the Web and Databases*, 2003. (Cited on pages 24 and 28)

[351] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. Intel virtualization technology. *IEEE Computer*, 38(5):48–56, 2005. (Cited on page 40)

[352] David A. Van Veldhuizen and Gary B. Lamont. Multiobjective evolutionary algorithms: Analyzing the state-of-the-art. *Evolutionary Computation*, 8(2):125–147, 2000. (Cited on pages 151, 152, and 153)

[353] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008. (Cited on pages 29 and 162)

[354] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, 2001. (Cited on page 66)

[355] P. Verissimo, A. Bessani, and M. Pasin. The TClouds architecture: Open and resilient cloud-of-clouds computing. In *IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops*, pages 1–6, 2012. (Cited on pages 63, 65, 74, and 210)

[356] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pmapper: Power and migration cost aware application placement in virtualized systems. In *ACM/IFIP/USENIX International Conference on Middleware*, pages 243–264, 2008. (Cited on pages 111 and 113)

[357] David Villegas, Norman Bobroff, Ivan Rodero, Javier Delgado, Yanbin Liu, Aditya Devarakonda, Liana Fong, S. Masoud Sadjadi, and Manish Parashar. Cloud federation in a layered service model. *Journal of Computer and System Sciences*, 78(5):1330–1344, 2012. (Cited on page 51)

[358] Werner Vogels. Beyond server consolidation. *Joual of ACM Queue*, 6(1):20–26, 2008. (Cited on page 40)

[359] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *1st International Conference on Cloud Computing*, pages 254–265, 2009. (Cited on page 41)

[360] QuangHieu Vu, Mihai Lupu, and BengChin Ooi. Routing in peer-to-peer networks. In *Peer-to-Peer Computing*, pages 39–80. Springer, 2010. (Cited on pages 25 and 34)

[361] Marc Waldman, Aviel Rubin, and Lorrie Cranor. Publius: A robust, tamper-evident, censorship-resistant web publishing system. In *9th USENIX Security Symposium*, pages 59–72, 2000. (Cited on page 24)

[362] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. *Symposium on operating systems design and implementation*, 36(SI):181–194, 2002. (Cited on page 38)

[363] Stefan Walraven, Dimitri Van Landuyt, Eddy Truyen, Koen Handekyn, and Wouter Joosen. Efficient customization of multi-tenant software-as-a-service applications with service lines. *Journal of Systems and Software*, 91:48–62, 2014. (Cited on pages 183, 184, and 187)

[364] Cong Wang, Qian Wang, Kui Ren, Ning Cao, and Wenjing Lou. Toward secure and dependable storage services in cloud computing. *IEEE Transactions on Services Computing*, 5:220–232, 2012. (Cited on pages 32 and 34)

[365] Lei Wang, Jianfeng Zhan, Weisong Shi, and Yi Liang. In cloud, can scientific communities benefit from the economies of scale? *IEEE Transactions on Parallel and Distributed Systems*, 23(2):296–303, 2012. (Cited on page 32)

[366] Lizhe Wang and Samee Khan. Review of performance metrics for green data centers: a taxonomy study. *The Journal of Supercomputing*, pages 1–18, 2011. (Cited on page 92)

[367] Lizhe Wang, Gregor Laszewski, Andrew Younge, He Xi, Marcel Kunze, Tao Jie, and Fu Cheng. Cloud computing: a perspective study. *New Generation Computing*, 28 (2):137–146, 2010. (Cited on page 41)

[368] Lizhe Wang, Gregor von Laszewski, Marcel Kunze, Jie Tao, and Jai Dayal. Provide virtual distributed environments for grid computing on demand. *Advances in Engineering Software*, 41(2):213–219, 2010. (Cited on page 213)

[369] Ucilia Wang. Microsoft pledges to be carbon neutral starting this summer. `gigaom.com/2012/05/08/microsoft-pledges-to-be-carbon-neutral-by-the-summer`, 2012. Last accessed in July 2014. (Cited on page 92)

[370] Dan Werthimer, Jeff Cobb, Matt Lebofsky, David Anderson, and Eric Korpela. SETI@Home–massively distributed computing for SETI. *IEEE Computing in Science and Engineering*, 3(1):78–83, 2001. (Cited on pages 11 and 34)

[371] James Robert White, Malcolm Matalka, W. Florian Fricke, and Samuel V. Angiuoli. Cunningham: a BLAST runtime estimator. *Nature Precedings*, pages 1–6, 2011. (Cited on page 141)

[372] Bhathiya Wickremasinghe, Rodrigo N. Calheiros, and Rajkumar Buyya. CloudAnalyst: A CloudSim-based visual modeller for analysing cloud computing environments and applications. In *24th IEEE International Conference on Advanced Information Networking and Applications*, pages 446–452, 2010. (Cited on page 32)

[373] Erik Wittern, Jörn Kuhlenkamp, and Michael Menzel. Cloud service selection based on variability modeling. In *10th International Conference on Service-Oriented Computing*, pages 127–141, 2012. (Cited on page 148)

[374] Michael Wooldridge. *An Introduction to MultiAgent Systems*. Wiley Publishing, 2nd edition, 2009. (Cited on page 102)

[375] Linlin Wu, Saurabh Kumar Garg, and Rajkumar Buyya. SLA-based resource allocation for software as a service provider (SaaS) in cloud computing environments. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 195–204, 2011. (Cited on pages 106 and 214)

[376] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *10th USENIX Conference on Networked Systems Design and Implementation*, pages 329–342, 2013. (Cited on page 17)

[377] Min Yang and Yuanyuan Yang. An efficient hybrid peer-to-peer system for distributed data sharing. *IEEE Transactions on Computers*, 59(9):1158–1171, 2010. (Cited on page 27)

[378] Sami Yangui, Iain-James Marshall, Jean-Pierre Laisne, and Samir Tata. CompatibleOne: The open source cloud broker. *Journal of Grid Computing*, 12(1):93–109, 2014. (Cited on page 54)

[379] Yuan Yao, Longbo Huang, Abhishek Sharma, Leana Golubchik, and Michael Neely. Power cost reduction in distributed data centers: A two-time-scale approach for delay tolerant workloads. *IEEE Transactions on Parallel and Distributed System*, 25 (1):200–211, 2014. (Cited on page 113)

[380] Stephen S. Yau, Yu Wang, and Fariaz Karim. Development of situation-aware application software for ubiquitous computing environments. In *26th Annual International Computer Software and Applications Conference*, pages 233–238, 2002. (Cited on page 200)

[381] Sungkap Yeo and Hsien-Hsin Lee. Using mathematical modeling in provisioning a heterogeneous cloud computing environment. *IEEE Computer*, 44(8):55–62, 2011. (Cited on page 91)

[382] Christos A. Yfoulis and Anastasios Gounaris. Honoring SLAs on cloud computing services: a control perspective. In *EUCA/IEEE European Control Conference*, 2009. (Cited on page 41)

[383] Lamia Youseff, Maria Butrico, and Dilma da Silva. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop*, pages 1–10, 2008. (Cited on page 162)

[384] Chi Zhang and Arvind Krishnamurthy. Skipindex: Towards a scalable peer-to-peer index service for high dimensional data. Technical Report TR-703-04, Princeton University, 2004. (Cited on page 25)

[385] Qi Zhang, Lu Cheng, and Raouf Boutaba. SWPS3 fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2. *BMC Research Notes*, 1:107–110, 2008. (Cited on pages 115 and 127)

[386] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1:7–18, 2010. (Cited on pages 29, 30, 31, 45, and 46)

[387] Qi Zhang, Eren Gürses, Raouf Boutaba, and Jin Xiao. Dynamic resource allocation for spot markets in clouds. In *11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, pages 1–6, 2011. (Cited on pages 106 and 214)

[388] Shuai Zhang, Xuebin Chen, Shufen Zhang, and Xiuzhen Huo. The comparison between cloud computing and grid computing. In *International Conference on Computer Application and System Modeling*, volume 11, pages V11–72–V11–75, 2010. (Cited on pages 16, 35, and 212)

[389] Tianle Zhang, Zhihui Du, Yinong Chen, Xiang Ji, and Xiaoying Wang. Typical virtual appliances: An optimized mechanism for virtual appliances provisioning and management. *Journal of Systems and Software*, 84(3):377–387, 2011. (Cited on pages 148, 177, 184, and 187)

[390] Zhizhong Zhang, Chuan Wu, and David W.L. Cheung. A survey on cloud interoperability: Taxonomies, standards, and practice. *ACM SIGMETRICS Performance Evaluation Review*, 40(4):13–22, 2013. (Cited on pages 51 and 54)

[391] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004. (Cited on page 25)

[392] Yong Zhao, Xubo Fei, I. Raicu, and Shiyong Lu. Opportunities and challenges in running scientific workflows on the cloud. In *CyberC*, pages 455–462, 2011. (Cited on pages 2, 54, 129, 141, and 146)

[393] Liang Zhou, Baoyu Zheng, Jingwu Cui, and Sulan Tang. Toward green service in cloud: From the perspective of scheduling. In *International Conference on Computing, Networking and Communications*, pages 939–943, 2012. (Cited on pages 112 and 113)

[394] Eckart Zitzler. *Evolutionary algorithms for multiobjective optimization: methods and applications*. PhD thesis, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, 1999. (Cited on page 152)

[395] Stefan Zoels, Zoran Despotovic, and Wolfgang Kellerer. On hierarchical DHT systems - an analytical approach for optimal designs. *Computer Communications*, 31(3): 576–590, 2008. (Cited on page 27)