Universitat Politècnia de Catalunya (UPC)
Departament d'Arquitectura de Computadors (DAC)

Semantic Resource Management and Interoperability between Distributed Computing Platforms

Jorge Ejarque Artigas

Advisor
Rosa Maria Badia Sala

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR PER LA UNIVERSITAT POLITÈCNICA DE CATALUNYA

Barcelona November 3, 2015

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola de Doctorat

Curs acadèmic: 2015/2016

Acta de qualificació de tesi doctoral

Nom i cognoms
Jorge Ejarque Artigas

Programa de doctorat
Arquitectura de Computadors

Unitat estructural responsable del programa
Departament d'Arquitectura de Computadors

Resolució del Tribunal

Reunit el Tribunal designat a l'efecte, el doctorand / la doctoranda exposa el tema de la seva tesi doctoral titulada

" Semantic Resource Management and Interoperability between Distributed Computing Platforms"

Acabada la lectura i després de donar resposta a les qüestions formulades pels membres titulars del tribunal, aquest atorga la qualificació:

☐ NO APTE ☐ APROVAT ☐ NOTABLE ☐ EXCEL·LENT

(Nom, cognoms i signatura) President/a	(Nom, cognoms i signatura) Secretari/ària

(Nom, cognoms i signatura) Vocal	(Nom, cognoms i signatura) Vocal	(Nom, cognoms i signatura) Vocal

_____, _____ d'/de _____ de _____

El resultat de l'escrutini dels vots emesos pels membres titulars del tribunal, efectuat per l'Escola de Doctorat, a instància de la Comissió de Doctorat de la UPC, atorga la MENCIÓ CUM LAUDE:

☐ SÍ ☐ NO

(Nom, cognoms i signatura) President de la Comissió Permanent de l'Escola de Doctorat	(Nom, cognoms i signatura) Secretari de la Comissió Permanent de l'Escola de Doctorat

Barcelona, _____ d'/de _____ de _____

Abstract

Distributed Computing is the paradigm where the application execution is distributed across different computers connected by a communication network. The first important distributed computing platforms were Clusters, where a set of computers were working together in a single location, connected by a Local Area Network. The evolution of the Internet enabled the connection of computers and users across locations, in such a way that computers can be connected to each other and accessed from different parts of the world, resulting in an advanced distributed computing platform called Grid. Grid Computing focuses on sharing computing resources provided by different entities, creating a global computing infrastructure which is available to different user communities. During last years, the Grid Computing has evolved very fast integrating different technologies resulting in what is currently known as the Cloud. The Cloud Computing paradigm has become a revolutionary approach in distributed computing, providing computing and data resources, on demand, in a very dynamic fashion, and following the Utility Computing model where you pay only for what you consume.

Different types of companies and institutions are exploring the potential benefits of moving their IT services and applications to Cloud infrastructures, in order to decouple the management of computing resources from their core business process to become more productive. Nevertheless, migrating software to Clouds is not an easy task, since it requires a deep knowledge of the technology and services offered by providers and how to use them. Among others, developers need to design how to partition the software into Virtual Machines (VMs), how to build the VM images and how to provision the computing resources and deploying the VMs and there is no easy solution for performing all these tasks. Besides this complex deployment process, the current cloud market place has several providers offering resources with different capabilities, prices and quality, and each provider uses their own properties and APIs for describing and accessing their resources. Therefore, when customers want to execute an application in the providers' resources, they must understand the different providers' description, compare them and select the most suitable resources for their interests. Once the provider and resources have

been selected, developers have to inter-operate with the different providers' interfaces to coordinate the application execution steps. To do all the mentioned steps, application developers have to deal with the design and implementation of complex integration procedures.

This thesis presents several contributions to overcome the aforementioned problems by providing a platform that facilitates and automates the integration of applications in different providers' infrastructures lowering the barrier of adopting new distributed computing infrastructure such as Clouds. The achievement of this objective has been split in several parts. In the first part, we have studied how semantic web technologies can improve the description of applications and how to automatically infer a model for deploying them in a distributed platform. We propose an ontology that provides a general-purpose and infrastructure-agnostic model for describing distributed applications. This model consists on a component topology which describes the application components and their communication links. Applying reasoning over the semantic application description, we can classify the components and communication links and infer the implicit affinity constraints. Once the application deployment model has been inferred, the second step is finding the resources to deploy and execute the different application components. Regarding this topic, we have studied how semantic web technologies can be applied in the resource allocation problem. In this case, we have defined an ontology which models the concepts involved on the assignment of resources to the computing tasks. Allocation policies are modeled as horn rules which are applied over a knowledge base composed by the application and the available resources descriptions.

Once the different components have been allocated in the providers' resources, it is time to deploy and execute the application components on these resources by invoking a workflow of provider API calls. However, every provider defines their own management interfaces, so the workflow to perform the same actions is different depending on the selected provider. In this thesis, we propose a framework to automatically infer the workflow of provider interface calls required to perform any resource management tasks. This framework includes a Semantic Annotation/De-annotation component which is in charge of to automatically generating semantic descriptions from interface descriptions and calls and vice versa. Then, the translation among the different models is done by rule reasoning and AI planning. On the one hand, the Resource Mapper component converts data from one provider model applying rules which define the data equivalences. On the other hand, the Action Planner component is in charge of finding the action equivalence. To do it, it generates an AI planning problem form the requested management tasks and the provider model. This problem defines the initial and goal resource states, and an AI

planning domain which models the state transitions with the available provider interface methods. The result of this AI planning problem provides the sequence of provider actions which performs the requested management action.

In the last part of the thesis, we have studied how to introduce the benefits of software agents for coordinating the application management in distributed platforms. We propose a multi-agent system which is in charge of coordinating the different steps of the application deployment in a distributed way as well as monitoring the correct execution of the application in the computing resources. Two types of agents have been defined: Application Agents which are in charge of the application management; and Infrastructure Agents which are in charge of the resource management. When an application is submitted, the Application Agent infers the deployment model and negotiates with the other Provider Agents a resource allocation for the application components. After a successful negotiation, the Provider Agents of the assigned resources are in charge of deploying and running the application components using the Infrastructure Interoperability Framework. Once the application is running, both agents are in charge of monitor the correct application execution. The different contributions have been validated with a prototype implementation and a set of use cases.

Acknowledgements

Elaborating this PhD thesis has required a hard work during a long time. During these years, several people have help me in such a way and I want to thank all of them with these few lines.

First of all I want to thank my thesis advisor, Rosa Maria Badia. Her experience and advice have been really appreciated during the whole life-cycle of the thesis. I also want to mention all my colleagues at UPC and BSC with whom I have share an enjoyable work environment. Specially, those who has worked in the same research group (Raül, Marc, Daniele, Enric, Francesc, Roger, Carlos, Javier, Cristian, Sandra, Pol, Fredy, ...) and those ones that I have collaborated in the different projects (Iñigo, Ferran, Jordi, Josep, Mario, Andras, Xabriel, ...). I do not want to forget my family, who have supported me in every moment, and my wife who has suffer my stress during the last period of the thesis. Finally, I want to dedicate this thesis to my daughter, Carla, who has become the sense of my life.

In few words, thank you very much.

Jorge

This work has been partially funded by the Ministry of Science and Technology of Spain (contracts CICYT TIN2007-60625 and TIN2012-34557), by the CoreGRID European Network of Excellence (contract 004265), by the European research projects BREIN (contract 034556) and OPTIMIS (contract 257115), by the Spanish Avanza NUBA project (contract TSI-020301.1009.3), by Generalitat de Catalunya (contracts 2009-SGR-980 and 2014-SGR-1051) and by the grant SEV-2011-00067 of Severo Ochoa Program, awarded by the Spanish Government.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Distributed Computing is the paradigm where the executions of applications are distributed across different computers connected by a communication network. The first important distributed computing platforms were Clusters, where a set of computers were working together in a single location, connected by a Local Area Network (LAN). The evolution of the Internet enabled the connection of computers and users across different locations, so computers can be connected to each other and accessed from different parts of the world resulting in an advanced distributed computing platform called Grid. Grid Computing [1] is focused on sharing computing resources provided by different entities, creating a global computing infrastructure which is available to different user communities, following a model which is similar to the power Grid. During last years, the Grid Computing has evolved very fast adopting a Service Oriented Architecture [2], where technologies such as virtualization and models such as Utility Computing [3] has been integrated, resulting in what is currently known as the Cloud. The Cloud Computing [4] paradigm has become a revolutionary approach in distributed computing, providing computing and data resources, on demand, in a very dynamic fashion, and following the Utility Computing model where you pay only for what you consume.

This paradigm is becoming more attractive for different types of companies and institutions. Thus, we observe an increasing interest in exploring the potential benefits of moving partial or totally their IT services and applications to Cloud infrastructures in order to decouple the management of computing resources from their core business process to become more productive. Cloud computing has also provided software vendors the opportunity to deliver their software in a more effective way. Integrating software solutions with cloud resources allows software vendors to simplify the distribution and

maintenance processes since: they do not need to provide solutions for the different client platforms; the software can be provided in conjunction with the computing resources that offers better performance; and even applying patches and updates can be done automatically and transparently to the user. Finally, Cloud Computing is also interesting for e-science. With cloud computing technologies, scientist do not require to buy and maintain large infrastructures for launching their computations, they can benefit from the infrastructure flexibility provided by Clouds, and using and paying just the period of time they need. In few words, Cloud Computing has emerged as a new way to obtain computing resources

However, migrating software to clouds is not an easy task, since it requires a deep knowledge of the technology and services offered by the cloud providers and how to use them. Among others, developers need to design how to partition the software into Virtual Machines (VMs), how to build the VM images and how to provision the computing resources and deploying the VMs, and there is no easy solution for performing all these tasks. During the last years, a new market of platform services has appeared to provide solutions to the aforementioned problems. The platform services, that we can find in the current Cloud market, can be classified into two types: the ones derived from big alliances among the most important software vendors and major Cloud providers that offer software in an exclusive way; or small platform providers which focus on the most used software stacks such as traditional well-known Model-View-Controller (MVC) frameworks. However, for other type of software, the offer is limited to simple platform services (such as simple batch job executions) or to the basic infrastructure services which are complex to use. The integration of automatic resource provisioning or adaptation mechanism to tailored scientific applications or independent software vendors requires a lot of programming effort and skills for adapting it to a Cloud provider's API.

Nowadays, there are several hosting providers which follow the Cloud Computing approach, offering resources with different capabilities, prices and quality of service (QoS). In addition to these providers, there are several Grids which also provide heterogeneous resources for executing computational tasks. All of them create a market place, where users can get resources for executing their applications. So, depending on the users' preferences, the type of application and their requirements, resources from one or several providers can fit better for executing an application than others.

In these cases, integration procedures are more complex and must be re-designed and re-implemented for the new providers. Despite these providers offer similar computing resources, each provider uses different metrics and properties for describing their resources and they differ from the used by other providers. When customers want

to execute an application in the providers' resources, they must understand the different providers' descriptions, compare them and decide which is the most suitable for their requirements and interests. Once the suitable resources and providers have been selected, the execution process also arise problems to inter-operate with the different providers' interfaces in order to coordinate the application execution steps and to react to failures and unexpected events.

The work presented in this thesis aims to overcome the aforementioned problems by providing a platform that facilitates and automates the integration of any kind of applications in different providers' infrastructures lowering the barrier of adopting new distributed computing infrastructure such as Clouds.

1.2 Objectives

This section exposes the problems that users have to face for deploying and executing applications in distributed computing platforms and proposes a set of objectives to overcome these problems in order to achieve the main thesis goal.

Problem statement

The process of deploying an application to current distributed computing platforms can be mainly split in three phases: the application deployment model design, the selection of the resources, and the execution of the provisioning and deployment processes. In the first phase, the user has to design how to split the application in different parts and to find the best way to deploy these parts in distributed resources for achieving a certain performance. This application decomposition is what we refer as deployment model. The process of finding this deployment model is a complex engineering process that can be tedious and time-consuming for a common developer. However, developers with a large experience in distributed computing, based on their know-how, can infer what can be the best deployment model by evaluating the properties of the different components of an application. The research question that raises at this point is:

Could the deployment model of an application be automatically inferred by a software expert system which emulates the evaluations performed by experienced developers?

Once the users have found a proper deployment model for their applications, the second problem that they have to face is the selection of the best resources for deploying their application. The resource allocation is not a new problem; it already exists in every shared computing platform. Nevertheless, the complexity in this case is caused

by the wide variety of providers, resources and application types that exist in the current distributed computing platforms market. Due to this heterogeneity, resource allocation systems must understand what are the capabilities offered by the different providers and what are existing capabilities requested by applications. Despite providers and developers are offering and searching for similar capabilities, they use their own vocabulary for describing resource properties and applications requirements according to different schemes and models. Therefore, the second research question we set out about the problem is:

Could an intermediate layer be designed to facilitate the common understand of the different provider's resource capabilities and application requirements?

Finally, in the third phase, developers have to perform the provisioning of the selected resources and the deployment of the application on the selected resources. To perform these processes, the infrastructure providers offer an interface to create and manage their resources. Developers have to use this interface to deploy their applications by implementing a workflow which invokes a set of actions exposed in the providers' interface. The design and implementation of this deployment workflow can be more or less complicated depending on the provider. However, the tedious problem appears if users want to change the provider or combine some of them to perform a more efficient application execution. In this case, they have to design and implement a new workflow per provider in order to implement the application deployment to the provider's interface. Moreover, if a provider changes its interface, developers have to adapt the deployment workflow associated to this provider, complicating the maintenance of the application deployment processes. So, the final research question is:

Could the workflow for deploying an application in a provider be automatically inferred from rich application and providers descriptions?

According to the different aforementioned research questions, we set out the following thesis objectives:

Objective 1

The first objective of this thesis is investigating a methodology for facilitating the integration of the different applications in distributed computing platforms requiring the minimum effort from the application developers. To be more specific, we aim to automatically infer how an application must be deployed in a distributed environment from just a description provided by the application developer.

Objective 2

The second objective of this thesis is providing an intermediate layer where users and providers' schemes can be uniformly understood and processed for enhancing, facilitating and automating the allocation and provisioning processes.

Objective 3

Finally, the last objective of the thesis, but not less important, is solving this provider lock-in, where changing a provider in the market place is transparent to the user and does not imply an extra effort for the users. We aim to provide a framework where the application deployment workflow can be automatically inferred from a rich description of the providers' interfaces.

1.3 Contribution

Different technologies can help to achieve the aforementioned objectives. The Semantic Web [5] provides a technology stack composed by a set of languages, tools and frameworks for specifying and managing the knowledge contained on web resources. As difference from HTML and XML, the semantic languages provide a formal and machine understandable and sharable way of describing knowledge. A semantic description is composed by an Ontology, which provides a common vocabulary to denote the types, properties and interrelationships of concepts in a domain, and the resource description, which provides the description of the ontology concept instances as a set of subject-property-object triples. Moreover, new knowledge can be inferred from semantic descriptions with the help of ontology reasoners and rule engines.

The problem tackled by semantic web has similarities with the thesis objectives. As mentioned in previous sections, computing resources available in the cloud market are a special type of resources which are accessible on the Web. Resource descriptions are different depending on the provider and the type of applications which can use these resources is diverse. In this thesis, we propose the usage of Semantic Web languages for providing uniformed, sharable and machine understandable descriptions of applications and computing resources as well as the concepts involved in application deployment processes. Then, the required information for performing the application deployment model and resource allocation will be inferred by applying ontology and rule reasoning on these semantic descriptions,

Contribution 1: Application deployment Model Inference

Therefore, as a first contribution on this topic, we propose an ontology which describes the concepts about the application deployment domain. This ontology proposes a general purpose infrastructure-agnostic application model which mainly defines the application components, the communication required between the components and a set of rules for defining the required quality. Reasoning on these semantic application descriptions, which are described according to the ontology, we can extract a deployment model which defines atomic group of component instances and their affinity and anti-affinity relationships

Contribution 2: Semantic Resource Allocation

A similar approach is followed for the allocation of these components groups on the available providers' resources. As a second contribution, we propose extensions on the existing computing resource ontologies with the resource allocation concepts. According to these concepts, we propose to define allocation policies as a set of rules which are evaluated by a rule engine in order to obtain the allocation results.

Once the component-resource allocation is found, it is time to perform the deployment on the providers of the selected resource. As mentioned before each provider offers their own interface producing a vendor lock-in, because it is required to implement a different deployment workflow per provider, complicating the usage of multiple providers. An interesting technology for solving this issue is the AI planning [6]. AI planners are able to find the combination of actions which are required to achieve a desired goal. This is performed by a state-space search where actions are defined as state transitions.

Contribution 3: Infrastructure Interoperability

As a third contribution, we propose to apply AI planning for automatically obtaining the required workflow for provisioning the computing resources and deploying the application components. Following this approach, we define the application deployment as a planning problem. To do it, it is required to define the planning domain, which defines the environment states and the possible state transitions, and the planning problem, which describes the initial and goal states. In our case, the planning domain is basically built by the description of the computing infrastructure available on the resource ontologies and the actions provided by the provider's interface to manage them. On the other hand, the initial and goal state are extracted from the application resource allocation extracted in a previous step. The sequence of tasks provided by the planner as a solution of the planning problem is also the required workflow for deploying the application on the providers'

resources.

Contribution 4: Multi agent management

Finally, Multi-agent technologies [7] are used to increase the autonomy and self-management of a system. Agents are proactive, so, they can take decisions by themselves according to their goals and trigger actions by their own initiative. For these reasons, agents are suitable for coordinating the allocation and execution of tasks in these distributed computing environments. Agents can adapt their behavior depending on how the execution of the application is evolving, detecting execution problems and triggering the most appropriate actions for reacting to them depending on the system status and the resource capabilities. Agents are also capable to communicate to each other. They implement negotiation protocols which are very useful to reach agreements and cooperate with other agents. These capabilities can be used to put different interests together, provisioning the resources required by the applications in the most suitable way for users and providers.

Figure 1.1 shows a summary of the main thesis contributions, the connection between them and how they are related with the thesis chapters. In the first contribution, we have designed and implemented an Application Deployment Ontology and an Application Model Reasoner. The Application Deployment Ontology provides the model to describe the application and the Application Model Reasoner infers the deployment model for each application description by applying the rules described in the Application Deployment Ontology. The inferred deployment model is the input of the second contribution, where we have designed and implemented a Semantic Resource Allocation which allocates the application deployment model on the infrastructure resources. To perform it, we have also defined a Resource Allocation Ontology which provides the knowledge to automatically perform the resource allocation. Once the resource has been allocated, the Infrastructure Interoperability framework automatically finds a workflow which provisions the resources and deploys the application component for the different providers. It is the third contribution of the thesis, and the final contribution is a multi-agent distribution of the whole system.

The rest of thesis document is organized as follows. Chapter 2 presents the state of the art on the topics of the thesis. Then, Chapter 3 focuses on the first contribution of the thesis, describing the application ontology and the reasoning process to map applications to a deployment model. Chapter 4 presents the second contribution about the semantic resource allocation and Chapter 5 explains the work on applying AI planning for the providers' interoperability. Afterwards, the last contribution of the thesis about the multi-

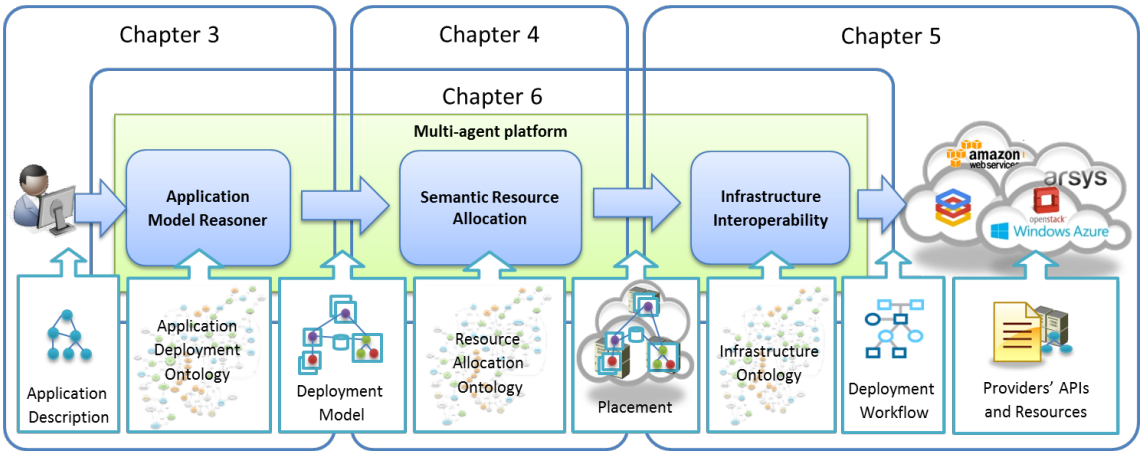


Figure 1.1 Thesis Contributions summary

agent management is presented in Chapter 6. Finally, the thesis is concluded by Chapter 7 where we explain our conclusions and proposes some guides for the future work.

1.3.1 Publications related to the thesis

The work of the thesis has generated the following publications:

Journals

[CAI11] J. Ejarque, J. Álvarez, H. Muñoz, R. Sirvent, R. M. Badia, "Service Orchestration in an Heterogeneous Cloud Orchestration", in *Computer and Informatics*, Vol. 31 (1), pp. 45-60, 2012.
(Presents part of the work on Infrastructure Interoperability)

[CCPE10] J. Ejarque, M. de Palol, I. Goiri, F. Julià, J. Guitart, R. M. Badia, J. Torres. "Exploiting Semantics and Virtualization for SLA-driven Resource Allocation in Service Providers". in *Concurrency and Computation: Practice and Experience*, Vol. 22 (5), pp. 541-572, 2010.
(Presents the part of the work on Semantic Resource Allocation task)

Book Chapters

[OSCCSPP11] J. Ejarque, J. Álvarez, R. Sirvent, R. M. Badia, "Resource Allocation for Cloud Computing: A Semantic Approach", in *Open Source Cloud Computing*

Systems: Practices and Paradigms, pp. 90-112, IGI Global, 2011
(Presents the Multi-agent architecture of our work)

International Conferences

[CLOUD15] J.Ejarque, A. Micsik, R.M. Badia "Towards Automatic Application Migration to Clouds", in *Proceedings of the 8th IEEE International Conference on Cloud Computing*,pp. 25-32, 2015
(Presents the work on Application Model Reasoning and AI planning for Infrastructure Interoperability)

[SEKE12] X.J. Collazo-Mojica, J.Ejarque, S.M.Sadjadi, R. M. Badia, "Cloud Application Resource Mapping an Scaling Based on Monitoring of QoS Contraints model and quality rules", *in Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering*,pp. 88-93, 2012.
(Presents work related with the Quality rules of the Application Model and its relationship with the monitoring and the adaptation)

[CloudCom11] J. Ejarque, J. Álvarez, R. Sirvent, R. M. Badia, "A Rule-based Approach for Infrastructure Providers' Interoperability", *in Proceedings of the 3rd IEEE International Conference in Cloud Computing Technology and Science*, pp. 272-279, 2011.
(Presents the work on Infrastructure Interoperability)

[Ibergrid11] J. Ejarque, J. Álvarez, H. Muñoz, R. Sirvent, R. M. Badia, "Orchestrating Services on a Public and Private Cloud Federation", *in Proceedings of the 5th Iberian Grid Infrastructure Conference*, pp.61-72, 2011.
(Presents part of the work on Infrastructure Interoperability)

[PDP11] J. Ejarque, A. Micsik, R. Sirvent, P. Pallinger, L. Kovacs, R. M. Badia, "Job Scheduling with License Reservation: A Semantic Approach", *in Proceedings of the 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*,pp.47-54, 2011.
(Presents an extension of the Semantic Resource Allocation for supporting software licenses as another kind of resource)

[CloudCom10] J. Ejarque, R. Sirvent, R. M. Badia, "A Multi-agent approach for

Semantic Resource Allocation“, *in Proceedings of the 2nd IEEE International Conference in Cloud Computing Technology and Science*,pp. 335-342, 2010.
(Presents the work on the distribution of the Semantic Resource Allocation in a multi-agent system)

[CCGV10] J. Ejarque, A. Micsik, R. Sirvent., P. Pallinger, L. Kovacs, R. M. Badia, ”Semantic Resource Allocation with Historical Data Based Predictions“, *in Proceedings of the 1st International Conference on Cloud Computing, GRIDs, and Virtualization*, pp. 104-109, 2010.
(Presents an integration of the Semantic Resource Allocation with task requirements predictors)

[e-Science08] J. Ejarque, M. de Palol, I. Goiri, F. Julià, J. Guitart, R. Badia, and J. Torres, ”SLA-Driven Semantically-Enhanced Dynamic Resource Allocator for Virtualized Service Providers“, *in Proceedings of the 4th IEEE International Conference on e-Science*, pp. 8-15, 2008.
(Presents the work in Semantic Resource Allocation for Grids and private Clouds)

[SCC08] J.Ejarque, M. de Palol, I. Goiri, F. Julià, J. Guitart, J.Torres, R. M. Badia, ”Using Semantic Technologies for Resource Allocation in Computing Service Providers“, *in Proceeding of the 5th IEEE International Conference on Services Computing*, pp. 583-587, 2008.
(Presents the initial work on the Semantic Resource Allocation)

Chapter 2

State of the art

This chapter presents the state of the art on the area of this thesis. In the first sections of the chapter, we present the current available options for deploying applications on recent distributed computing platforms such as Clouds. The first option is deploying applications by using the interfaces offered by infrastructure providers to manage computing resources. In this section, we describe how developers can use infrastructure services, the complexity of using several providers and the existing barriers for changing the provider. Afterwards, we will see how current solutions address the infrastructure interoperability and which are their benefits and drawbacks. The second option is deploying applications with Platforms services. In this part, we present the different Platform-as-a-Service alternatives and their main advantages and drawbacks. Finally, we describe approaches which are also using semantic-based or model-based descriptions to deploy applications in clouds. In this part we focus on discussing the advantatges and limitations of these solutions. Then, we will see the different options for allocating computing resources in the distributed computing platforms, which is one of the fundamental parts for deploying and executing application in an efficient way. Finally, the chapter is concluded by describing how the semantic and multi-agent technologies have been applied in the different distributed platforms.

2.1 Direct application deployment on Infrastructure providers

The main building block of modern distributed platforms is the infrastructure services layer which is popularly known as Infrastructure as a Service (IaaS). With these services the Infrastructure Providers offer computing resources in an on-demand and pay-per-use fashion. Each commercial infrastructure provider has their own implementation of infrastructure services, however there are also some cloud middleware frameworks such as OpenNebula [8], OpenStack [9] or CloudStack [10] which allow to any organization

to easily deploy the infrastructure services for building a private cloud with their own computing resources. Infrastructure services offer to users a web interface and API to create and manage Virtual Machines according to their processing requirements. So, developers have to use this interface to provision and manage the resources required by their applications. It implies that applications are bound to this provider interface, and each infrastructure provider develops its own managing interface. Therefore, if for some reason developers want to switch to another provider or use several providers, it can be challenging because it requires the adaptation of the application code to support new providers' interfaces. This effect is known as *vendor lock-in* problem.

2.1.1 Infrastructure Interoperability solutions

The approach proposed to solve the vendor lock-in issues can be summarized in two: the standardization of the management interface, where the main stakeholders of the infrastructure providers market reach an agreement to implement the same interface, or other the plug-in solutions, where a community defines an interface for their interests and the integration with the different infrastructure providers is performed by means of plug-in or drivers which implements the interface functionalities using the specific provider's services. There are several examples of these approaches in the literature; next paragraphs highlight some of them.

Regarding standardization, several standardization organizations are interested in defining their own standard cloud interface. For instance, the Open Grid Forum (OGF) [11] has presented the Open Cloud Computing Interface (OCCI) [12] which describes an interface with the common management actions for managing the compute, storage and network resources of the Cloud. Another similar standardization proposal is vCloud [13] submitted by VMWare in the Distributed Management Task Force (DMTF) [14] which presents similar functionalities. Standards are very useful to develop new markets over infrastructure services. However, the adoption of the standards is not as successful as in other fields. One reason of this is that Cloud Computing is evolving fast and the standardization process is slow. Moreover, providers have to dedicate development efforts to implement standard interfaces and small providers do not want to commit this effort if a clear standard is not broadly adopted. Finally, big vendors (Amazon, Google, Microsoft, etc.) are reluctant to change their interfaces and the vendor lock-in is beneficial for their interests due to the fact that they already have most of users. Keeping this preferential position is easier, if there are barriers in the process of changing the provider.

For plug-in approaches, we have the examples of the Apache projects jClouds [15]

and delta-Cloud [16] or the Dasein Cloud API [17]. These examples are open-source projects which maintain a definition of a common Interface for managing cloud resources from applications and, the different contributors of the projects are in charge of providing the plug-ins to interoperate with the different providers' interfaces. The main drawback of these solutions is that they are difficult to maintain. A change in the providers API requires at least a change in the plug-in for this provider, and potentially, it can also require a change in the common API definition and the reimplementation of all the plug-ins.

Besides the advantages or drawbacks that the mentioned solutions can have, the direct deployment of applications to infrastructure providers is only recommended when the application is simple to deploy (just some file copies in very few VMs or when developers have a deep knowledge of the infrastructure services offered by the providers). However, current applications have complex deployments than just deploy a VM. They require several steps to split the application on VMs, create these VMs, install the application components, configure the infrastructure to communicate VMs and set up a monitor system for controlling the application execution and the resource adaptation. In pure infrastructure solutions, all these issues must be performed by the user which is a complex task for common software developers.

2.2 Application Deployment with Platform Services

An easier option to deploy complex applications is the usage of Platform as a Service (PaaS) offerings. These options provide services for facilitating the resource provisioning, application installation and adaptation. Examples of these services are: Heroku [18] or CloudControl [19]. These platform services have two limitations: first, they focus on a set of application's stacks, traditionally MVC applications developed with Spring [20], Django [21], Ruby on Rails [22], etc. If the application is not using the predefined stack, the user is not able to use the platform; and secondly, they also hide the underlying infrastructure, binding to work only with the Infrastructure Provider the company has an agreement with.

Big vendors also offer their platform services such as Google App Engine [23], Microsoft Azure App Service [24] or Amazon Cloud Formation [25]. In this case, they provide a set of APIs to implement the applications which are able to run only their own computing infrastructure. In both cases, if you are not happy with your platform provider, you are forced to change the deployment of your application. So, developers have again a provider lock-in but in another layer of abstraction.

Regarding this issue, we can find two types of solutions: open Platform as a Service

such as Cloud Foundry [26] or Cloudify [27]; or CloudPier [28], the solution proposed by the Cloud4SOA project [29]. The Cloud4SOA project has worked in providing semantic interoperability between different PaaS platforms, which allow users to find and use platform services which are compatible with the application stack. Cloud Foundry and Cloudify get a similar approach like the common API solution but at the PaaS level. They are open PaaS offerings which provide a set of core capabilities and services to enable the deployment and adaptation of applications for different Infrastructure Providers. The common problem in both types of solutions is that, at the end, they only provide easy deployment and adaptation features for the same type of applications, developed with well-known stacks as other PaaS. In the Cloud4SOA case, the problem is simpler. The Cloud4SOA platform only acts as a broker, so if the user does not use one of the stacks supported by other PaaS, the platform will not be able to find and use any available PaaS offerings.

In the open-PaaS cases, as they are open-source communities, the user could try to extend the platform to support his/her application stack but it will be similar (or worse) than trying to use directly cloud providers with one of the available common APIs. These cases, also inherit the problem of having to change the implementation or definition in the upper layer services as a consequence of a change in the Infrastructure Services. As a conclusion, these approaches are not tackling the core of the cloud deployment problem, i.e. how to deploy applications in the distributed infrastructures in a general way. They just provide solutions for most used cases or for specific infrastructures.

2.3 Application Model driven development and deployment

Other recently appeared solutions to improve Cloud interoperability and to facilitate the cloud deployment are based on extensible and machine-processable models to describe cloud resources and applications. The main idea behind this is the use of these models to inter-operate with the cloud services and resources in different phases of the application life-cycle. The benefits of this idea are that models are not bound to a specific implementation, can be easily extended and can be used to automate processes. The most popular definition of cloud applications is the Open Virtualization Format (OVF) [30] that defines an application as a set of Virtual Appliances through a description of the required VM properties in terms of CPU, disks and files of the VM image to be loaded. This format is supported by several hypervisors and cloud middleware and has been used and extended in several projects like Reservoir [31], OPTIMIS [32], and VENUS-C [33]. However, this

model is close to the infrastructure details, therefore, it is attractive for experienced cloud developers and system administrators, but not productive for general software developers.

For the aforementioned reason, recent projects like mOSAIC [34], REMICS [35], PaaSage [36] or MODAClouds [37] are focused on providing high-level applications models which try to avoid the specification of the infrastructure details. The mOSAIC project proposed an ontology [38] for the IaaS offering and an application model based on design patterns for parallel applications. For each pattern or application, developers must define a set of rules which map the application with the Infrastructure level resources, and these rules are used to deploy the application. The most extensive model for Clouds is the CloudML [39] which is developed in conjunction by the projects REMICS, MODAClouds and PaaSage. The difference with mOSAIC is that instead of matching the application with rules they define a language to specify the deployment of the application in the cloud resources. The common drawback in those approaches is that developers must explicitly specify the details of how the application is deployed in the cloud according to the infrastructure model. The same happens with the Topology and Orchestration Specification for Cloud Applications (TOSCA) [40] proposed by OASIS, developers have to specify the deployment model indicating which components are deployed in each VM. Describing this deployment model is better than implementing the adaptation of applications to the cloud. However, it is still too complex for common developers and it remains as a main barrier to achieve a high-productive deployment of applications for cloud.

2.4 Resource Allocation in distributed platforms

One of the key topics when deploying an application in distributed platforms is the allocation of computational tasks on the infrastructure resources. On the application side, resource allocation is important because it affects on the application execution performance. Depending on how many resources are assigned to the application it can take more or less time to perform the same computation. On the other side, resource allocation is important for providers because it affects on their revenue. They have the interest of maximizing the number of applications, using the minimum number of resources. So, there is a trade-off between users' and providers' interests which must be solved by assigning the appropriate resources to an application providing the best application performance without wasting resources. In the literature, we can find different approaches for solving the resource allocation problem which varies depending on the platform and the application. Next paragraphs provide an overview of the existing

problems and solutions.

Traditional cluster computing allows developers to execute applications in a set of homogenous resources which is shared by different users. The user applications are executed in a cluster by submitting batch computations called jobs. Computing clusters are managed by queue systems which manage the execution of jobs in the cluster's resources. Between other management tasks, these systems are in charge of scheduling the jobs submitted by the users on the different cluster resources. There are several job schedulers available such as the Portable Batch System (PBS) [41], Load Sharing Facility (LSF) [42] or Maui [43] just to mention some of them. These schedulers offer different policies to allocate the resources to jobs by prioritizing some kind of jobs or users depending on the provider's interest.

As mentioned in the introduction, another available distributed computing platform is Grid Computing which allows users to submit jobs at resources belonging to different organizations. In this distributed platform, each organization manages their own resources as traditional clusters and, on top of this a Grid middleware, such as the Globus Toolkit [44], Unicore [45] or gLite [46], is in charge of providing a secure access to the different organization resources and managing the remote execution of jobs across the different sites. In this computing paradigm, we have two levels of resource allocation. In the lower level, organizations have their own local schedulers which are in charge of scheduling jobs inside the organization cluster as mentioned before. In the upper level, the Grid middleware provides a meta-scheduler which is in charge of scheduling jobs across the resources of the different sites. Examples of these meta-schedulers are the Globus Resource Allocation Manager (GRAM) [47] or GridWay [48]. The GRAM service is a meta-scheduler which is developed to perform the scheduling of jobs between the organizations connected by the Globus middleware meanwhile, GridWay is a meta-scheduler, which is capable to interoperate with different Grid middleware by using a plug-in approach. Different methodologies have been proposed for solving meta-scheduling, the first option was to use traditional scheduling policies in grid environments, whose initial evaluation can be seen in [49]. However, a fundamental difference of Grids from Clusters is that resources offered by organizations are different and users can decide where to execute their applications. To take into account these facts, meta-schedulers require to integrate matchmaking, such as the Condor ClassAds [50], and selection mechanisms, such as the rank based selection proposed in [51], the reputation based selection proposed in [52] or market based selection such as [53], between others. Meta-schedulers are focused on scheduling jobs without taking into account if they belong to larger applications. In the literature, we can find more complex approaches which try to

improve the results of meta-scheduling by defining new components on top of the current meta-schedulers. One of these components is the application broker, like the GridBus broker [54], which is in charge of managing the scheduling and execution of all the jobs of an application. Another interesting proposal is the SA-Layer [55] which proposes a set of components for introducing scheduling in advance capabilities to Grids. With these capabilities, users can know how an application will be scheduled on the Grid which is fundamental to know if the application execution will fulfill the quality required by the user as proposed in [56].

Similar resource allocation problems happen in Cloud Computing. Cloud computing also offers a distributed platform where different users can access to resources managed by different providers. The main difference between both systems is the usage model. Cloud computing uses a pay per use model where users request resources with a certain capabilities and the provider gives full access to a virtual machine with the requested capabilities and charges for the usage time. In grid computing, providers only allowed batch execution on their resources for a limited period of time. Regarding resource allocation, the main difference between these computing models is how a computational task is defined. While in grid computing a job description describes the time constraints, in cloud computing, the usage time of a VM is undefined. It means that instead of scheduling jobs at the resources like in Grids, in cloud computing, the cloud middleware has to find the best placement for the VMs depending on the provider's interests. In the literature, we can find several VM placement strategies followed to achieve a certain objective. The most common objective is maximize the provider profit which can be achieved by applying a placement policy for overloading the resources [57] or by applying an energy-efficient placement policy like in [58] or [59] to reduce the operational cost. Another important objective is maximizing the provider reliability which can be achieved by applying a placement policy to minimize the number of VM failures [60] [61].

2.5 Semantic Web in distributed platforms

Grid and Cloud providers share a common problem. They offer distributed computing platforms where heterogeneous computing resources are offered to users for executing a wide variety of applications and services. Each provider uses their own vocabulary for describing their resources' capabilities and the users also describe their applications according to different schemes and models. For deciding the most appropriate provider for an application, users and providers have to understand the different vocabularies and schemes to compare the providers' capabilities with the application requirements. As

introduced in the previous chapter, the Semantic Web technologies are mainly used to allow the specification of semantic meaning to web resource descriptions in a machine processable way, enabling software components to understand these descriptions, taking decisions and performing actions based on the knowledge inferred from these semantic descriptions. The Semantic Grid has been proposed in [62] trying to apply the ideas of the semantic web approach for Grid resources and services, making them understandable by the different meta-schedulers and brokers in a structured and uniform way. Following this approach, the OntoGrid project [63] has proposed a reference model (Semantic OGSA [64]) for attaching and managing semantic descriptions for grid resources. This model defines a special type of grid resource, which links a normal Grid resource with their semantic description, and a set of basic services for managing these special semantic resources.

Focusing on the area of resource allocation, most of the proposals have been concentrated on resource discovering and matchmaking. For instance, [65] and [66] present two similar methodologies for matchmaking, defining a simple ontology for defining resource adverts and job requirements and a set of rules for selecting which advert fulfills the requirements. In [67], the authors present another option for resource discovery based on the evaluation of SPARQL [68] queries in resource descriptions which is simpler than the previous one. Other approaches solving a similar problem are presented in [69] and [70] which have been applied in grid resource brokers [71]. However, these works present two problems. The first problem is that they only address one part of the resource allocation problem, leaving the scheduling decisions to other services which can vary depending on the grid middleware used for accessing to the resource. Our approach is trying to solve this issue applying semantics in the whole resource allocation process, that is, the resource discovery and matchmaking and the scheduling of jobs in the discovered resources according to the resource providers' preferences and the users' interests. The second problem is their assumption that resources and jobs are already semantically annotated and mapped to their ontology but real systems do not offer annotated resource descriptions. Regarding the latter issue, the GRIP project [72] has worked on the interoperability between different Grid middleware using semantics. A result of this project is the work presented in [73]. The authors present a solution for integrating the Globus and Unicore, defining a common ontology and their equivalences to the Globus and Unicore models using the OWL notation [74]. So, when a description from one middleware is provided, it is mapped to an ontology class in the middleware model. The ontology reasoning infers the equivalent class in the common ontology and the other middleware models. However, this solution can only perform one-to-one equivalences

which is valid in this case but insufficient for a general case where other types of mappings are required.

The semantic web can be also applied in Cloud Computing. The IaaS Cloud layer has several similarities with Grid Computing because grid resource descriptions and virtual machine descriptions are very similar. Therefore, discovery and matchmaking solutions proposed for Grids could be also used for Clouds with few modifications such as the approaches presented in [75], [76], [77] and [78]. Another current research topics are focused on providing semantic interoperability at PaaS layer or proposing ontology and models for describing the deployment of applications in the cloud infrastructures. Regarding the first topic, the work presented in [79], [80] and [81] propose ontologies and frameworks for describing and discovering similar PaaS services using semantic web techniques. Regarding the sementic cloud application deployment, the main models and ontologies proposed for describing cloud applications have been already explained in Section 2.3.

2.6 Multi-agent systems in distributed platforms

The idea of using software agents in the management of distributed platforms was introduced in [82], where Foster et al. describe the benefit of integrating the results in both research areas. Software agents could improve the autonomy, flexibility and scalability of current distributed platforms in different areas. However, multi-agent system researchers have mainly focused on the area of resource allocation and job scheduling. Regarding this area, several solutions have been proposed, such as the ones based on market-control, where each agent tries to maximize its benefit and the market mechanism controls them; the ones based on social welfare, where the multi-agent system tries to maximize a collective benefit; and the ones based on game theory. There is a lot of literature about market based allocation solutions. We would like to highlight proposals Challenger [83], which was one of first proposals to distribute the resource allocation through multiple agents and Tycoon [84], which is one of the most used systems. The works on welfare engineering and game theory for multi-agents resource allocation has been compiled in [85] and [86]. Regarding other studies more focused on Grid computing, we can highlight TRACE [87], where a set of homogeneous agent are coordinated to perform a certain volume of tasks; ARAM [88], where different type of agents are in charge of different parts of the resource allocation; the distributed resource allocation methodologies proposed by the CatNets project [89] in [90], where the catallaxy theory of free market systems is applied to allocate the resources to user applications; and finally,

the Sorma [91] project, where a resource allocation mechanism is proposed based on a dynamic trading of ICT resources.

Regarding Cloud computing, our contribution was one of the first one to apply multi-agent systems not only for resources allocation but also to provide an autonomous management of application and resource in Clouds. Posterior to our contribution, we can find other solutions for combining agents and clouds. One of these solutions is proposed by the mOSAIC project. In [92], the authors propose a multi-agent framework for negotiating and managing the agreements between users and providers. Other solutions have been focussed on the distributing the allocation of jobs on VMs accorss several agents such as [93] or [94] solutions, or the monitoring and adaptation such as in [95] and [96].

Other approaches related to Clouds and multi-agent systems are focused on deploying multi-agent platforms on top of the cloud computing infrastructure or on developing distributed applications as a set of software agents. However they are out of the scope of this thesis.

Chapter 3

Inferring the Application Deployment Model

Having in mind the main objective of the thesis about proving a platform to facilitate the deployment of applications in distributed platforms, in this chapter we focus on the first part which aims at studying how a suitable model for deploying an application can be inferred with the minimum human intervention. Starting from an application description, our system must be capable to provide how the different parts of the applications must be deployed in the distributed platforms. The solution presented and evaluated in this chapter proposes a methodology which applies some of the ideas of the Semantic Web area to infer the suitable deployment model. The rest of the chapter is organized as follows: first, Section 3.1 provides an overview of the proposed methodology; then, Section 3.2 and Section 3.3 describe in detail the infrastructure-agnostic model for describing applications; and how the deployment model is inferred. Afterwards, Section 3.4 provides the evaluation and validation of the methodology and the chapter is finalized with Section 3.5, where we draw the conclusions about the topic discussed on this chapter.

3.1 Methodology

A proper execution of applications in distributed platform relies on how the application is decomposed and allocated into the different available computing resources. This task is usually time-consuming and, when the application includes a large number of components, it is also complex and tedious for non-expert developers. Based on their know-how, developers with a large experience in distributed computing can infer which can be the best deployment model by evaluating the properties of the different components

of an application. During this evaluation, the experienced developers implicitly apply a set of rules to identify the different parts of an application, classify them and based on this classification decide which components must be deployed together. If we were able to emulate this reasoning with computers, we would achieve the goal of inferring a suitable deployment model in an automatic way. This key capability is provided by Semantic Web technologies. As introduced in previous section, they allow machines to automatically infer new knowledge from existing data by applying reasoning. In our case, these reasoning techniques are used to infer the deployment model based on the application description. More in detail, the methodology consist on an Application Deployment Ontology which describe the application elements, their types, properties and relationships and a set of rules to classify the application elements according to their properties and, based on the classification, infer the affinity constraints to build the deployment model. Next sections provide more details about this ontology and the inference process.

3.2 Application Deployment Ontology

The proposed application model aims to provide a generic and infrastructure-agnostic way to describe distributed applications. In other words, the application model should be able to describe any kind of distributed application and should not require the re-implementation of the application neither the inclusion of glue-code to match with a certain API or specific platform. Based on these principles, an application in the proposed model is mainly described by component topology, consisting of a set of components and the links to intercommunicate them. For each topology element (components and links), developers have to also define the required quality as well as how it is installed, configured and executed in the computing infrastructure. Figure 3.1 provides an overview of the proposed application model. Next paragraphs describe in detail the different parts of the model. In addition to the description provided by the developer, the Application Deployment Ontology is complemented with a hierarchy of components and communication link types and a set of description logic rules which enables the inference of the deployment model. These parts will be described together with the inference process in Section 3.3.

3.2.1 Component Topology

A *Component* in our model identifies a part of the application functionality. The model does not force developers to map components to specific application parts. So, depending

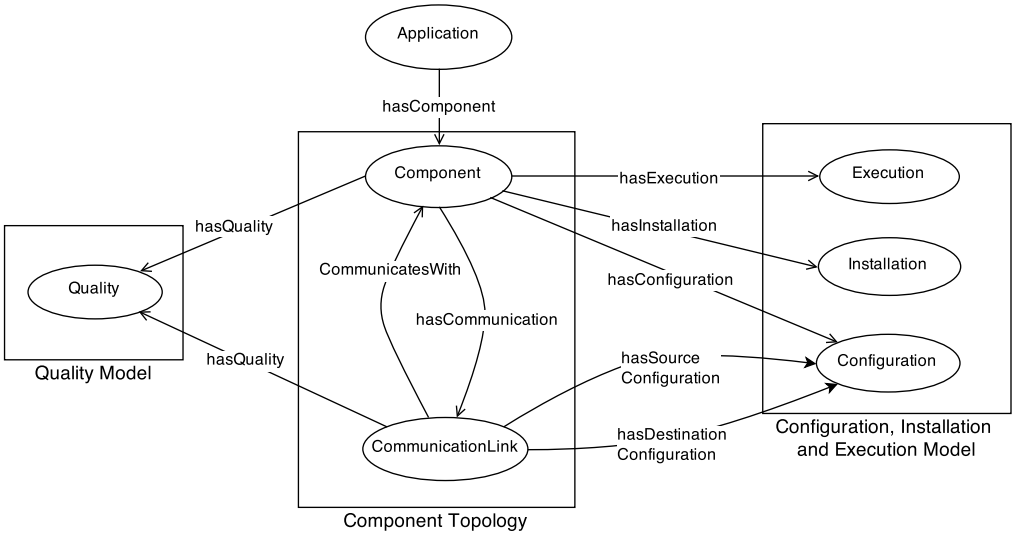


Figure 3.1 Application Description

on the stack used to develop the application, a component can be map to a method, a class, a package, a web service, a database, etc. This property makes our model generic enough to allow the description of any application, in contrast of other solutions that force developers to use the description bounded to the supported software stack.

The other important element of the topology is the *CommunicationLink*. It models the existing communication between two components and it is mainly defined by the cardinality, the source and destination components (one-to-one, one-to-many, many-to-one or many-to-many) and the communication channel, which indicates how the data is interchanged. The values of the communication channel property can be *Memory* (e.g. libraries sharing objects, arrays, etc.), *Disk* (e.g. processes which communicate by writing and reading files) or *Network* (e.g. web services interchanging messages).

3.2.2 Quality Description

For deploying and running a distributed application with a certain quality, there are several requirements that components and communication links must fulfill. These requirements include the number of instances, processing capabilities (such as number of core or the processor speed) and memory and storage usage for components and the bandwidth and latency constraints for communication links. The value of this requirement usually depends on the target quality level and the load of the application. For that reason, we propose the model depicted in Figure 3.2 to describe the quality required by components and communication links. For each *Component* and *CommunicationLink*, a set of *QualityRules* describe what should be the requirements for a given target

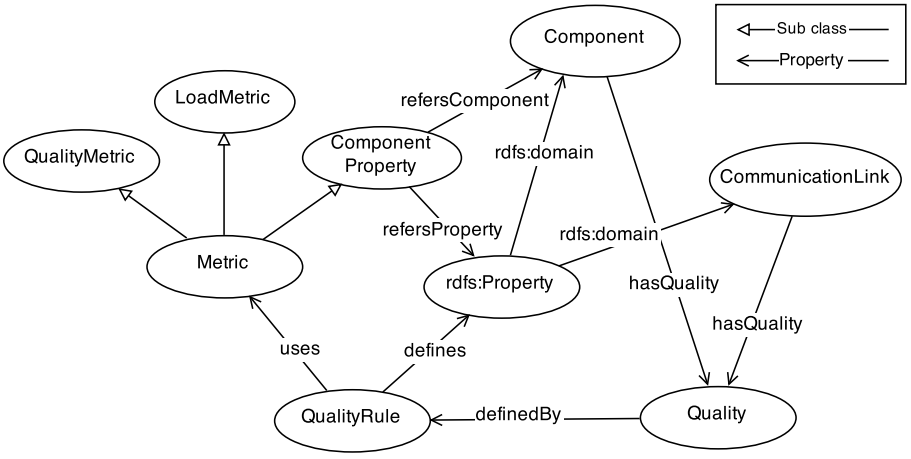


Figure 3.2 Quality Description Model

```
:ExampleComponentQuality rdf:type app:Quality ;
    app:definedBy :MetricExprInstancesRule, :StaticCoresRule;

## Define an static number of cores
:StaticCoresRule rdf:type app:QualityRule ;
    app:defines app:needsCores ;
    app:definedBy “4” .

## Define number of instances by an Expresion of quality and load metrics
:MetricExprInstancesRule rdf:type app:QualityRule ;##
    app:uses :RequestsPerSecond, :Requests;
    app:defines app:numberInstances ;
    app:definedBy ":Requests/(RequestPerSecond *10)" .

:RequestPerSecond rdf:type app:QualityMetric ;
    app:goalValue 20 .
:Requests rdf:type app:LoadMetric ;
    app:source "http://$:KOPIPortal[0].vm.ip/ganglia/requests" .
    app:expectedValue 100 .
```

Figure 3.3 Example of a Component Quality Description.

quality and application load. The description to calculate the Component and Link requirements can be easily specified as a mathematical expression whose variables can include application quality and load metrics or other component requirements. For including these mathematical expressions inside quality descriptions, we have reused the approach described in [97] which embeds mathematical formulas in the RDF objects.

Figure 3.3 provides an example of *Component* quality description, where two *QualityRules* are defined to assess the number of component instances and the required cores. In the first rule, we define a dynamic expression where the number of instances depends on a *QualityMetric* (*RequestPerSecond*) and a *LoadMetric* (*Requests*). In the second rule, we just define a static value for the number cores required by each component instance. This example also provides an example of *QualityMetric* description, where the goal level is defined, and an example of *LoadMetric* description, where the initial expected value and the place to get run-time values is defined.

```
:ExampleLinkQuality rdf:type app:Quality ;
    app:definedBy :CompPropExprBWRule .
 :CompPropertyBWRule rdf:type app:QualityRule ;
    app:uses :Requests, :ComponentInstancesProperty ;
    app:defines app:needsBW ;
    app:definedBy ":Requests / :ComponentInstancesProperty * 10000)" . ## Mbps
:ComponentInstancesProperty rdf:type app:ComponentProperty ;
    app:refersToComponent :Component
    app:refersToProperty app:numberInstances .
```

Figure 3.4 CommunicationLink Quality Description Example.

Similarly, Figure 3.4 shows an example of *CommunicationLink* quality description. In this example, there is a *QualityRule* defined for modeling the required bandwidth as function of a component property. The example also shows how to define the component property by indicating the component and the referenced property (*numberInstances*).

3.2.3 Installation Configuration and Execution Description

To finalize the application description, developers have to describe how the components are installed, configured, executed and finalized. For describing this part, we propose to follow a resource state based model similar than the used by dev-ops tools such as Puppet [98] or LCFG [99](Large Scale Unix Configuration System) to describe the host configuration. More in detail, we apply a semantic annotated version of the Puppet model to describe the component installation, the component and link configuration and the component execution description. The reason of using a semantic version of this model is to enable the semantic reasoning on top of the descriptions to extract implicit useful knowledge for inferring the deployment model. Once the deployment model is inferred we could easily transform the component descriptions into Puppet manifest which will automatically execute required actions on the computing resources to achieve the installation, configuration and execution states.

With this model, depicted in Figure 3.5, developers can describe the installation, configuration and execution of a component, as a set of application resource states. These resource states indicate the desired status that the different application resources should have after installing, configuring and starting the execution of a component. Figure 3.5 shows the core resource types (Files, Services, etc.) and its state is represented by the defined properties. There are some special properties which are used to express relationships between resources. Developers can describe resource state dependencies with the *requires* property or if a restart is required when a resource is updated with the *subscribes* property.

Figure 3.6 shows an example of an installation, configuration and execution of Java component which includes most of the issues commented above. The component

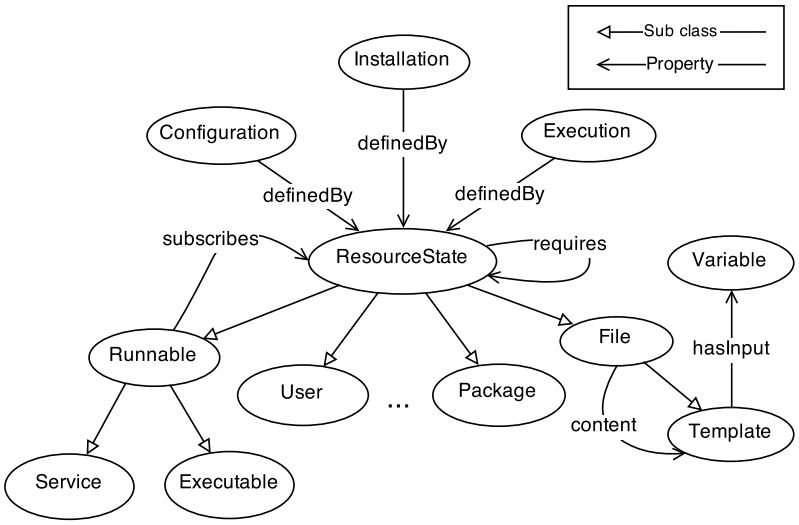


Figure 3.5 Installation, Configuration and Execution Model

```
:JavaComponentInstall rdf:type app:Installation ;
    app:definedBy :JDKPackage , :ComponentFolder, :JarFile .
:JREPackage rdf:type app:Package ;
    app:name "jre" ;
    app:ensures "installed" .
:ComponentFolder rdf:type app:Folder ;
    app:location "/opt/component/" ;
    app:ensures "exists" .
:JarFile rdf:type app:File ;
    app:requires :ComponentFolder ;
    app:location "/opt/component/" ;
    app:name "component.jar" .
    app:source "http://...." ;
    app:ensures "exists" .
...
:JavaComponentConfig rdf:type app:Configuration ;
    app:definedBy :ComponentConfigFile .
:ComponentConfigFile rdf:type app:File ;
    app:location “/opt/component/” ;
    app:name "component.properties" ;
    app:content app:ConfigTemplate ;
    app:ensures "exists" .
:ConfigTemplate rdf:type app:Template ;
    app:source "http://...." ;
    app:hasInput :ComponentsVar .
:ComponentsVar rdf:type app:Variable ;
    app:name “Components” ;
    app:value“$:Component[0..n].vm.ip” .
...
:ComponentExec rdf:type app:Execution ;
    app:definedBy :JavaExecutable .
:JavaExecutable rdf:type app:Executable ;
    app:requires app:JREPackage ;
    app:subscribes app:ComponentConfigFile ;
    app:command “java -jar /opt/component/component.jar Component” ;
    app:ensures "started" .
```

Figure 3.6 Component Installation, Configuration and Execution Description Example.

installation is described by the installation of the Java Runtime Environment (JRE) package, the existence of the component folder and JAR file in a certain location. The component configuration is defined by a configuration file whose content is defined by a template and an input variable. As you can see in the example, the content of this variable can be bound to the future IP addresses of the component's VMs.

As introduced above, one of the benefits of using this kind of model with respect of using a plain-text script definition is that we can reason over the model to extract implicit properties. For instance, Figure 3.6 shows that the *JavaExecutable* subscribes to *ComponentConfigFile*, so a change in the *ComponentConfigFile* will require a *JavaExecutable* restart, producing a downtime. So, evaluating the execution and configuration descriptions, a reasoner can infer if a component can be dynamically reconfigurable or its configuration should remain static. Next section describes in detail how this reasoning is performed.

3.3 Deployment Model Inference

Once a developer has described the application, it is time to apply the ontology and rule reasoning to infer the proper deployment model. Developers have to provide to the Application Model Reasoner the application description with: the definition of the component topology, the required quality including the goal and expected values for the quality and load metrics, and the installation, configuration and execution descriptions. During the reasoning process, the Reasoner is going to classify the topology elements, determine the processing and link requirements and the initial number of instances and infer implicit affinity constraints. The result of this process will be a deployment model description consisting on a set of groups of component instances which must be deployed together in the same virtual machine or the same provider location. This model will be the input for the Semantic Resource Allocation phase described in Chapter 4 where this groups will be allocated to the different available computing resources. Next paragraphs provide more details about the different parts of the introduced reasoning process.

3.3.1 Topology Elements Classification

The Application Model Reasoner classifies the different topology elements into the types depicted in Figure 3.7. This classification is automatically performed by applying description logic rules on the application description. Figure 3.8 shows some examples of these rules which infers the different types of *CommunicationLink*, *Component* and *Configuration* based on the defined element properties.

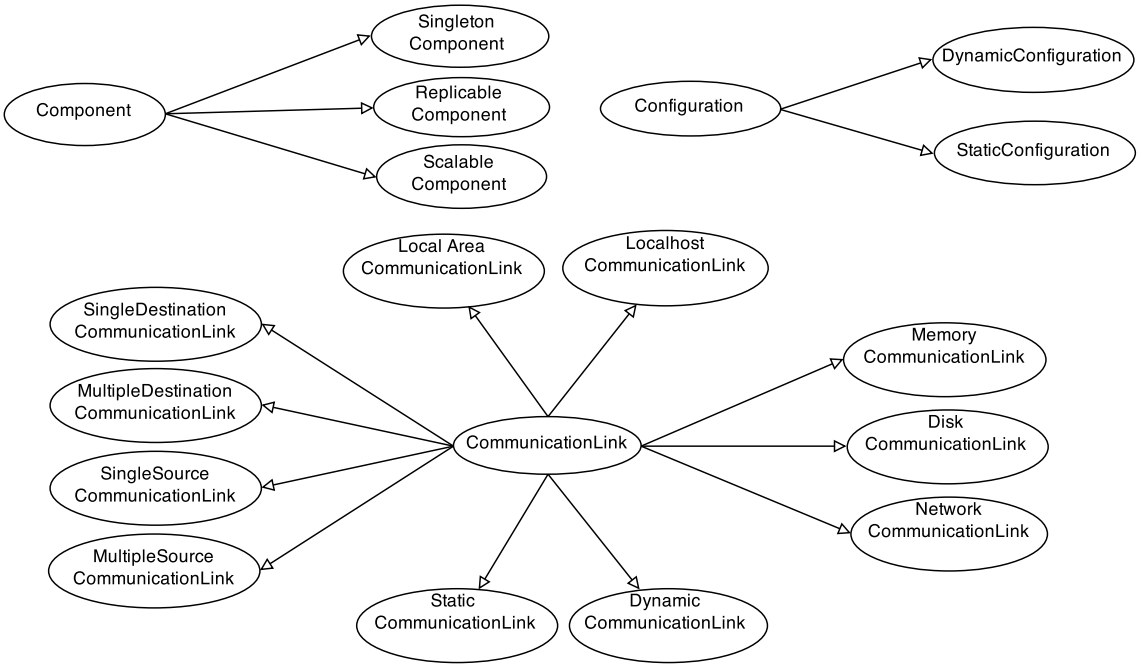


Figure 3.7 Application Elements Classification

For instance, a *Configuration* can be classified as *Static* or *Dynamic* by evaluating the Execution descriptions of the different *Components* and the *Configuration* of their *CommunicationLinks*. If a link configuration description disables the run-time reconfiguration of a component, the link configuration will be classified as *Static*, otherwise it will be classified as *Dynamic*. The first rule described in Figure 3.8 shows an rule example to detect a *StaticConfiguration*. According to this rule, a link configuration is *Static* when a runnable resource, described in a component execution, is subscribed to a resource which is modified during a link configuration.

Regarding *CommunicationLinks*, they are classified in different ways. First, they can be classified depending on the communication channel (*Memory*, *Disk* and *Network*). They can be also classified as *DynamicCommunicationLink* or *StaticCommunicationLink* according to the type of their *Configuration*. Another important link classification is done based on the cardinality of the source and destination components (*one* or *many*). This cardinality classification is the base for the component classification. If the defined communication links for a component only contains *SingleSource* and *SingleDestination CommunicationLinks* the topology only allows a single component instance, therefore the component will be classified as *Singleton*. If the component contains communication links with multiple cardinality (*MultipleSource* and *MultipleDestination*), the topology allows multiple instances of this component and it can be classified as *Replicable*, if it has a *StaticCommunicationLink*, or *Scalable* if all links are *DynamicCommunicationLinks*.

```
## Example of Configuration classification rule
Rule {
    ?Component rdf:type app:Component
    ?Component app:hasExecution ?execution
    ?Component app:communicatesWith ?link
    ?link app:hasConfiguration ?configuration
    ?execution app:hasResourceState ?execResourceState
    ?configuration app:hasResourceState ?confResourceState
    ?execResourceState app:definesResource ?execResource
    ?execResource rdf:type app:RunnableResource
    ?execResourceState app:definesResource ?confResource
    ?execResourceState app:requires ?confResource
=>
    ?configuration rdf:type app:StaticConfiguration
}
...

## Examples of CommunicationLink classification rules
Rule {
    ?link rdf:type app:CommunicationLink
    ?link app:hasChannel app:Memory
=>
    ?link rdf:type app:MemoryCommunicationLink
    ?link rdf:type app:LocalhostCommunicationLink
}
...
Rule {
    ?link rdf:type app:CommunicationLink
    ?link app:sourceCardinality “many”
=>
    ?link rdf:type app:MultipleSourceCommunicationLink
}

...
Rule {
    ?link rdf:type app:CommunicationLink
    ?link app:hasSourceConfiguration ?configuration
    ?configuration rdf:type app:StaticConfiguration
=>
    ?link rdf:type app:StaticCommunicationLink
}
...

## Example of Component classification rule
Rule {
    ?Component rdf:type app:Component
    ?Component app:communicatesWith ?link
    ?link rdf:type app:MultipleSourceCommunicationLink
    ?link rdf:type app:StaticCommunicationLink
=>
    ?Component rdf:type app:ReplicableComponent
}
...
```

Figure 3.8 Rules to classify the application description elements

```
Rule {
## Quality Rule which defines Requirement1 with Parameter1
    ?Component1 rdf:type app:Component
    ?Component1 app:hasQuality ?Quality1
    ?Quality1 app:definedBy ?QualityRule1
    ?QualityRule1 rdf:type app:QualityRule
    ?QualityRule1 app:defines ?Requirement1
    ?QualityRule1 app:uses ?Parameter1

## Quality Rule which defines Requirement2 with Parameter2
    ?Component2 rdf:type app:Component
    ?Component2 app:hasQuality ?Quality2
    ?Quality2 app:definedBy ?QualityRule2
    ?QualityRule2 rdf:type app:QualityRule
    ?QualityRule2 app:defines ?Requirement2
    ?QualityRule2 app:uses ?Parameter2

## Check that Parameter2 refers to Requirement1 and Parameter1 refers to Requirement2
    ?Parameter2 app:refersToComponent ?Component1
    ?Parameter2 app:refersToProperty ?Requirement1
    ?Parameter1 app:refersToComponent ?Component2
    ?Parameter1 app:refersToProperty ?Requirement2
=>
    throwFailure("Deadlock in definition of Quality Rules ?QualityRule1 and ?QualityRule2")
}
```

Figure 3.9 Rule to detect deadlocks in the Quality Rules descriptions.

Finally, *CommunicationLinks* can be also classified as *Localhost*, *LocalArea* and *WideArea* by evaluating the bandwidth and latency requirements. The results of this classification are useful to determine the affinity between components. Section 3.3.2 explains in detail how the *Component* and *CommunicationLink* requirements are extracted from *QualityRules*, and Section 3.3.3 describes how to determine the deployment affinity.

3.3.2 Determine Component and Link Requirements

Once the components have been classified, the Reasoner determines the required quality for each *CommunicationLink*, the processing requirements for each component and in case the component is *Replicable* or *Scalable*, the number of instances. These requirements are inferred by applying the defined *QualityRules* on the *Quality* description. Before applying the rules, the *QualityRule* descriptions are validated to check they will not produce a deadlock during the inference. To detect this deadlock, we have defined the rule described in Figure 3.9 which looks for a pattern in the *QualityRule* descriptions. This rule checks: if there is requirement definition in a *QualityRule* which uses a property defined by another *QualityRule* and, at the same time, another *QualityRule* uses the requirement defined by the former *QualityRule*.

Once the defined Quality Rules are validated, it is time to infer the values of components requirements form the *QualityRule* descriptions, and this is done by applying the rule described in Figure 3.10. For each component *QualityRule* defined it calculates

```
Rule {
    ?Component rdf:type app:Component
    ?Component app:hasQuality ?QualityRule
    ?QualityRule app:defines ?Requirement
    ?QualityRule app:definedBy ?MathExpresion
    getMetricsAndParameters(?QualityRule, app:uses, ?Parameters)
    calculatesValue(?MathExpresion, ?Parameters ?value)
=>
    ?Component ?Requirement ?value
}
```

Figure 3.10 Rule to infer the value of component requirements from Quality Rule descriptions.

```
Rule {
    ?Link rdf:type app:CommunicationLink
    ?Link app:needsLatency ?value
    lowerThan(?value, LH_LATENCY_THRESHOLD)
=>
    ?Link rdf:type app:LocalHostCommunicationLink
}
```

Figure 3.11 Example of rule to classify Communication Links.

the value for the defined requirement and includes it to the Component description. A similar rule is applied to infer the link requirements substituting the *Component* references to *CommunicationLink*.

3.3.3 Infer Component Affinity Constraints

The last step for inferring the deployment model is determining the implicit component affinity constraints. A component affinity indicates if the component instances should be deployed in the same virtual machine, in the same location, or should share a disk and it is basically determined by evaluating the required quality (explained in Section 3.3.2) and type of channels of the defined communication links. For instance, if two components have a memory communication or the required bandwidth and latency can be only achievable in a intra-host environment, their instances must be deployed in the same VM. In the case that the required bandwidth and latency can be only achievable in a local area environment, their instances must be deployed in the same provider location, in other cases there will not be any implicit affinity constraint.

To infer this, the Reasoner classifies the communication links as *LocalhostCommunicationLink*, *LocalAreaCommunicationLink* or *WideAreaCommunicationLink* according to the bandwidth and latency requirements. Figure 3.11 shows an example of a rule which classifies a CommunicationLink as *Localhost* if the required latency is smaller than a threshold.

After classifying communication links, the Reasoner infers the implicit affinity constraints. For components which are communicated by *LocalhostCommunicationLink*,

```
Rule {
    ?link rdf:type app:LocalAreaCommunicationLink
    ?link app:hasChannel app:Disk
    ?component1 app:communicatesWith ?link
    ?component2 app:isCommunicatedWith ?link
=>
    ?component1 app:sameLocation ?component2
    ?component1 app:sharedDiskWith ?component2
}
```

Figure 3.12 Example of rule to infer implicit affinity constraints.

the reasoner includes a component affinity constraint by defining the *sameVM* property between the components. Similarly, for component which is communicated by *LocalAreaCommunicationLink*, the reasoner defines the *sameLocation* property between the components. Finally, if a *LocalAreaCommunicationLink* has a *Disk* channel, the reasoner defines the property *sharedDiskWith* to indicate that a shared disk is required between components. An example of a Rule for making the reasoner to include an affinity constraint is shown in Figure 3.12.

After applying all the rules, we can obtain and enriched application description with the classified configurations, links and components, the inferred requirements, the number of instances as well as the affinity rules. The final deployment model is built by defining the different instances for each component according to the type, the processing requirements and the number of instances. Then, these instances are grouped according to the *sameVM* and *sameLocation* properties, and to finalize the deployment model, the shared disks are including based on the *sharedDiskWith* properties. For each group related by the *sameVM* property, we can calculate the group requirements by simply aggregating each component and link requirement. These aggregated requirements will be used as VM requirements.

3.4 Evaluation and Discussion

A working prototype of the described system has been implemented to validate the concepts presented in this chapter. The Application Deployment Ontology has been implemented with Protege. As explained in Section 3.2, it includes the application model and class hierarchy, which have been described using the OWL2 [100] language, and component classification and affinity inference rules which have been described using the Semantic Web Rule Language SWRL [101]. The Pellet Reasoner [102] has been used to implement the Application Model Reasoner, which applies the rules on the application descriptions provided by the user. The prototype implementation has been deployed in an Intel i5 laptop with 8GB RAM.

The evaluation performed with the prototype consists of three parts. In the first part, we have used the proposed application model to describe several applications and infer its deployment model. This is useful to validate the suitability of the model to describe different distributed applications as well as to better illustrate the system behavior. In the second part, we have evaluated the system overhead and its scalability by measuring the time to obtain the deployment model. Finally, in the third part, we compare the proposed application model with other available application models highlighting the advantages provided by our proposal.

3.4.1 Application Model Validation

To perform the validation of the proposed application model, we have selected a set of applications which represents different types of distributed applications. The first proposed application, the KOPI Application, is an example of N-tier web application which combines a typical 3-tier (Model-View-Controller) model with a computational intensive processing. The second application, Gene Detection, provides an example of a task-based application implemented with the COMPSs programming model [103]. The third application provide an example of how to describe a Map-Reduce [104] application and the last application provides an example of how to describe an hybrid application which combines the usage of Message Passing Interface(MPI) [105] and OpenMP [106]. The following paragraphs provide more details about how the different applications are described with the proposed model and what is the inferred deployment model. The first application includes a graphical representation of the application description and the semantic serialization of the key parts. For simplicity, in the other applications, we just include the graphical representation.

N-tier Application

The KOPI [107] is an On-line Plagiarism Search Portal developed by SZTAKI which implements an innovative cross-language plagiarism detection technique. It gives the opportunity to compare a reference document to other indexed collections of documents and search for potentially translated parts. This new technique is costly in terms of processing and data storage; therefore we sought the service is a potential candidate to be migrated to the cloud environment.

Figure 3.13 shows the component topology of the KOPI application and Figure 3.14 shows a snippet of the semantic application description. The application contains five components: the *KOPI Portal*, where users upload the documents to check; the *KOPI*

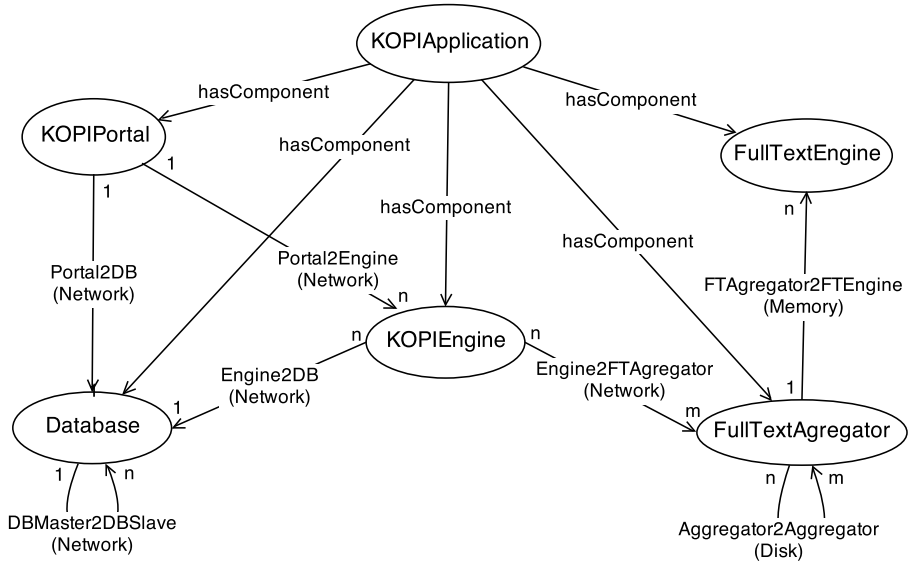


Figure 3.13 KOPI Application Overview

Engine, which is in charge of managing the document checking life-cycle; a *Database* to store documents and plagiarism check results; and the *Fulltext Aggregator* and *Fulltext Engine* to perform various indexed search sub-tasks for *KOPI Engines*. For each of these components, the developers have to define the communications with other components, the required Quality and the installation, configuration and execution description.

Figure 3.15 shows some examples of the communications defined for the KOPI Application. Each description includes the type of communication (one-to-one, one-to-many, etc.), its channel, the component which communicates with, and the link to the Quality description.

As explained in previous sections, the required *Quality* is defined by a set of *QualityRules*. Each *QualityRule* defines how to infer the quantity of resources consumed by a component or communication link according to values of quality and load metrics. In this case, the *KOPI QualityRules* are expressed as a function of the desired quality expressed in *CharsPerSecond* (indicates the document processing speed), and load metrics *NumberOfDocs* (indicates the number of simultaneous documents to be evaluated), *IndexSize* (indicates volume of documents to be compared) and *CharsPerDocument* (indicates the mean number of chars per document) metrics. Figure 3.16 shows the *Quality* description for the *Fulltext Engine* component, where the number of instances and the required hardware (CPU, memory, disk) are expressed as functions of some of the mentioned metrics. Following the same approach, users can define the *Quality* of the other components and communication links. In the case of communication links, *QualityRules* describe the bandwidth and latency requirements.

```
:KOPIApplication rdf:type app:Application ;
    app:hasComponent :KOPIPortal, :KOPIDatabase, :KOPIEngine,
                     :FulltextAgreegator, :FulltextEngine .

:KOPIPortal rdf:type app:Component ;
    app:hasCommunication :PortalDBCommunication ,
                         :PortalEngineCommunication ;
    app:hasConfiguration :PortalConfig ;
    app:hasInstallation :PortalInstall ;
    app:hasExecution :PortalExec ;
    app:hasQuality :PortalQuality .
    ...
:KOPIDatabase rdf:type app:Component ;
    app:hasCommunication :MasterSlaveCommunication ;
    app:hasInstallation :DBInstall ;
    app:hasConfiguration :DBConfig ;
    app:hasExecution :DBExec ;
    app:hasQuality :DBQuality .
...
:KOPIEngine rdf:type app:Component ;
    app:hasCommunication :EngineAgregatorCommunication ,
                         :EngineDBCommunication ;
    app:hasInstallation :KOPIEngineInstall ;
    app:hasConfiguration :KOPIEngineConfig ;
    app:hasExecution :KOPIEngineInstall ;
    app:hasQuality :KOPIEngineQuality .
...
:FulltextAgreegator rdf:type app:Component;
    app:hasCommunication :FTAgregatorAgregatorCommunication ;
                         :FTAgragatorEngineCommunication ;
    app:hasConfiguration :FTAgreegatorConfig ;
    app:hasInstallation :FTAggregatorInstall ;
    app:hasExecution :FTAggregatorExec ;
    app:hasQuality :FTAgregatorQuality .
...
:FulltextEngine rdf:type app:Component ;
    app:hasInstallation :FTEngineInstall ;
    app:hasConfiguration :FTEngineConfig ;
    app:hasQuality :FTEngineQuality .
...
```

Figure 3.14 Component Description Snippet.

```
:FTAgregatorEngineCommunication rdf:type app:CommunicationLink ;
    app:sourceCardinality "one" ;
    app:destinationCardinality "many" ;
    app:communicatesWithComponent :FulltextEngine ;
    app:hasChannel app:Memory ;
    app:hasQuality :AggregatorToEngineQuality ;
    app:hasSourceConfiguration :AggregatorToEngineConfig .
    ...
:FTAgregatorAgregatorCommunication rdf:type app:CommunicationLink ;
    app:sourceCardinality "many" ;
    app:numberSourceInstances "many" ;
    app:communicatesWithComponent :FulltextAgreegator ;
    app:hasChannel app:Disk ;
    app:hasQuality :AggregatorToAggregatorQuality ;
    app:hasSourceConfiguration :AggregatorToAggregatorConfig ;
    app:hasDestinationConfiguration :AggregatorToAggregatorConfig .
    ...
:MasterSlaveCommunication rdf:type :CommunicationLink ;
    app:sourceCardinality "many" ;
    app:numberSourceInstances "one" ;
    app:communicatesWithComponent :KOPIDatabase ;
    app:hasChannel :Network ;
    app:hasSourceConfiguration :MasterSlaveConfiguration ;
    app:hasQuality :DBToDBQuality .
    ...
```

Figure 3.15 Component Communication Description Examples.

```
:FTEngineQuality rdf:type app:Quality ;
    app:definedBy :FTEngineInstancesRule, :FTEngineCoresRule;
                  :FTEngineMemRule, :FTEngineDiskRule;
                  :FTEngineInstancesRule rdf:type app:QualityRule ;
    app:uses :CharsPerSecond ;
    app:defines app:numberInstances ;
    app:definedBy":CharsPerSecond/200" .
:FTEngineCoresRule rdf:type app:QualityRule;
    app:defines app:needsCores ;
    app:definedBy “1”.
:FTEngineMemRule rdf:type app:QualityRule ;
    app:uses :CharsPerSecond, :CharsPerDocument ;
    app:defines app:needsRAM ;
    app:definedBy ":CharsPerDocument / (:CharsPerSeconds * 10000)" . ## MB
:FTEngineDiskRule rdf:type app:QualityRule ;
    app:uses :CharsPerSecond, :CharsPerDocument ;
    app:defines app:needsDisk ;
    app:definedBy ":CharsPerDocument/ 1000000 " . ## GB
:CharsPerSecond rdf:type app:QualityMetric ;
    app:goalValue 20 .
:CharsPerDocument rdf:type app:LoadMetric ;
    app:source "http://$:KOPIPortal[0].vm.ip/documents/meanCharsNum" ;
    app:expectedValue 100 .
```

Figure 3.16 Fulltext Component Quality Description.

```
:DBInstall rdf:type app:Installation ;
    app:hasState :MySQLPackage, :DBFile .
:MySQLPackage rdf:type app:Package ;
    app:name "mysql" .
    app:ensures "installed" .
:DBFile rdf:type app:File ;
    app:requires :MySQLPackage" .
    app:location "/usr/mysql/dbs/" ;
    app:name "kopi.db".
    app:source "http://....";
    app:ensures "exists" .

:DBConfig rdf:type app:Configuration ;
    app:hasState :MySQLConfigFile .
:MySQLConfigFile rdf:type app:File ;
    app:location "file:///etc/mysql/" ;
    app:source "http://....";
    app:name "mysql_config" .
    app:ensures "exists" .

:DBExec rdf:type app:Execution ;
    app:hasState :MySQLService .
    app:requires app:MySQLPackage
    app:subscribes app:MySQLConfigFile
    app:ensures "started" .
```

Figure 3.17 Component Installation, Configuration and Execution Description Example.

The KOPI Application description is finalized by providing the installation, configuration and execution states for component and communication links. Figure 3.17 provides these descriptions for the *KOPI Database* component. The *KOPI Database* component installation (*DBInstall*) requires the installation of the *mysql* package and the file which store the initial database *DBFile*. Its configuration (*DBConfig*) requires the *mysql_config* file (described by *MySQLConfigFile*) which stores the configuration values. Finally, the component execution (*DBExec*) requires to have the *mysql* service started and it is subscribed to the *MySQLConfigFile* resource. It means that, when a change is introduced in this file, the *KOPI Database* component must be restarted.

Regarding the rest of components, a similar description is followed for the *KOPI Portal* and *KOPI Engine* where they require the *httpd* and the *tomcat* packages instead of *mysql*, and require installing some PHP and WAR files instead of the database file. In the case of the *FullText* components, they are Java components whose installation mainly requires to copy some of JAR files and scripts to execute them. Figure 3.6 from subsection 3.2.3 already described how the installation, configuration and execution of a Java component looks like.

Figure 3.18 shows the description of the *MasterSlaveCommunication*. It is an example of how to describe the configuration of a communication link. In this case, it is done by the modification of the *MySQLConfigFile*, whose content should match with a template that contains the master and slave addresses as input parameters. The value of master

```
:MasterSlaveConfiguration rdf:type app:Configuration ;
    app:hasState app:MySQLConfigFile .
:MySQLConfigFile rdf:type app:File ;
    app:hasContent :MasterSlaveTemplate .
:MasterSlaveTemplate rdf:type app:Template ;
    app:hasInput :MasterDBVariable,
                 :SlavesDBVariable .
:MasterDBVariable rdf:type app:Variable ;
    app:hasValue $:KOPIDatabase[0].host.ip$ ;
    app:name "master-node" .
:SlavesDBVariable rdf:type app:Variable ;
    :hasValue $:KOPIDatabase[1...].host.ip$ ;
    :name "slaves-nodes" .
```

Figure 3.18 Communication Configuration Description Example.

and slave are referenced to the IP address of the machine which hosts the first component instance (*:KOPIDatabase[0]*) and the rest of instances (*:KOPIDatabase[1...]*). The KOPI application uses properties files to configure the communication between its components, so a similar pattern is used for describing the rest of communication link configurations.

Finally, note that, as mentioned before, the *subscribed* property defined in the *DBExec*, which indicates that the *MySQLService* must be restarted each time the *MySQLConfigFile* file is updated. So, when a new instance is added, the number of component instances is modified and the configuration file content is changed. According to the description, the *MySQLService* must be restarted, producing a downtime which disables the dynamic scalability capability. As explained in previous sections, this pattern is searched by the Reasoner to infer if a component is just *Replicable* or *Scalable*.

Once the application is defined, we have submitted the application description to the prototype including the goal and expected values for *NumberOfDocuments, IndexSpace* and *CharactersPerSecond*. During the reasoning process, the system classifies the *KOPI Portal* as *Singleton* because the link topology only allows one component instance. The rest can have several instances. However, the system has classified the *Database* component as *Replicable* because of the configuration-installation pattern explained above. The rest of the components do not follow a similar pattern, they have been classified as *Scalable*. Applying the *QualityRules*, the Reasoner infers the processing, bandwidth and latency requirements as well as the number of instances. Evaluating these requirements and the communication channels types, the reasoner deduces that *KOPI Portal*, *KOPI Database* and *KOPI Engines* must be located in the same provider location because of bandwidth requirements; *Fulltext Engines* must be located in the same VM as *Fulltext Aggregators* because of the memory communication channel; and *Aggregator* instances must be located in the same location because of the required bandwidth and latency of the disk communication link is not assumable in a wide-area. Based on the described reasoning, the obtained deployment model for the KOPI application is depicted

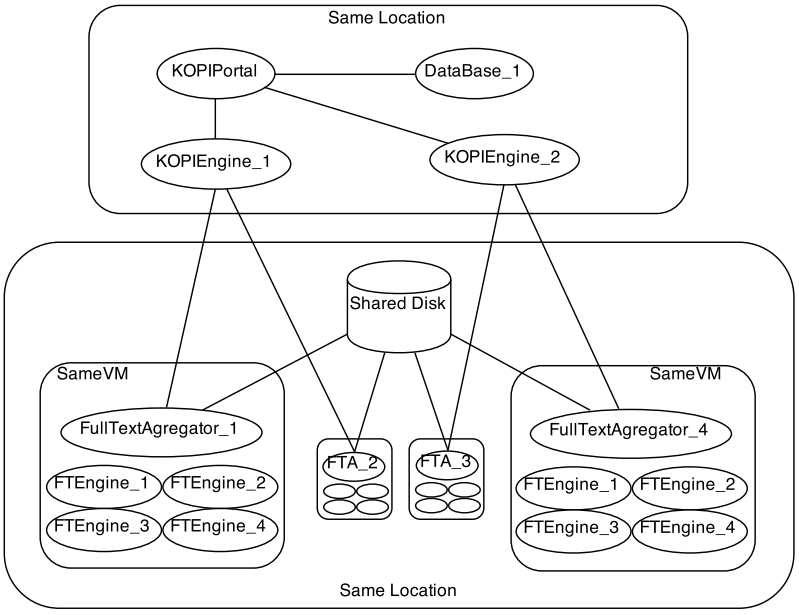


Figure 3.19 Inferred deployment model for KOPI Application Overview

in Figure 3.19.

Task-based Application

The Gene Detection application [108] implements an algorithm to automatically identify the relevant genes in a genomic DNA sequence and perform a functional analysis of these genes, which is faster than scanning the whole DNA. In more detail, the application finds relevant regions in similar DNA sequences by performing a comparison with the BLAST [109] software and after that runs the GeneWise [110] program only on those regions. An overview of the main workflow is depicted in Figure 3.20. Each box represents a different part of the application, which contributes to the overall process: Part A performs a genome format translation; Part B obtains a list of amino acid sequences which are similar to the input sequence; Part C searches the relevant regions comparing the sequences against the genome using BLAST; and finally, Part D performs the functional analysis on these regions by running the GeneWise software.

The application has been implemented with COMPSs which is a task-based programming model which extracts the implicit parallelism between the defined tasks and executes the application in a master-worker mode. With COMPSs developers define the code where the main workflow is implemented (*detectGenes* in this case) and the defined tasks. Following the same definition, we can build the application description using the proposed model as depicted in Figure 3.21. From the *detectGenes* component, there will be a *CommunicationLink* to all the other components, which represent the different type

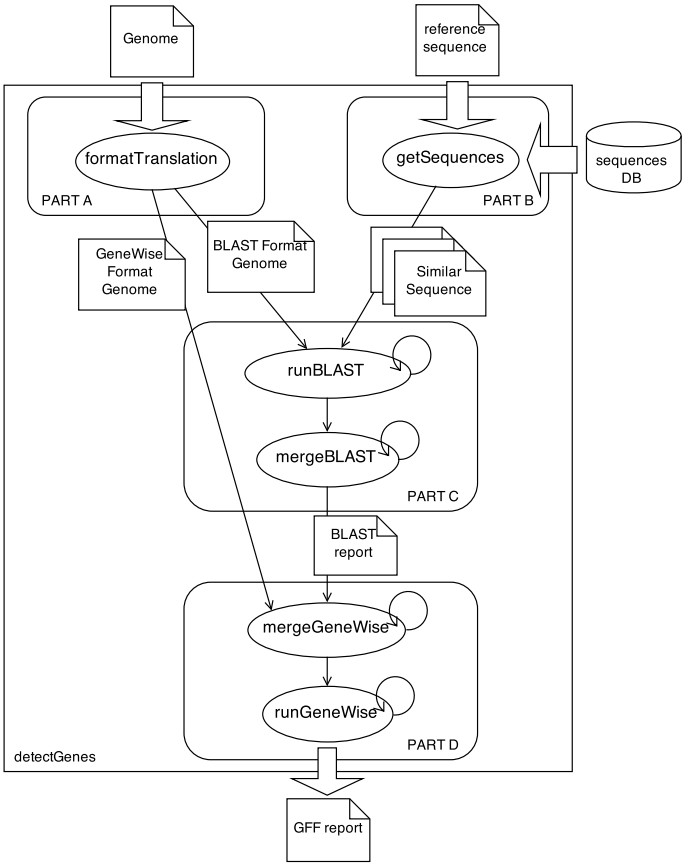


Figure 3.20 Gene detection workflow

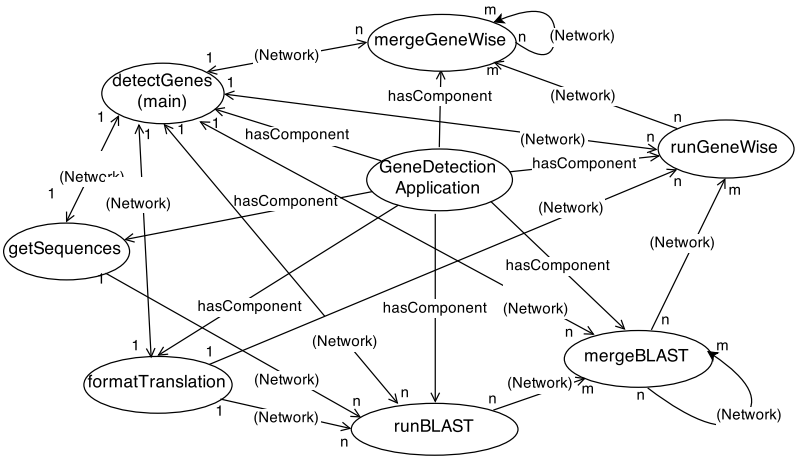


Figure 3.21 Gene detection application description

of tasks, in order to allow the remote execution and data transfer. The communication links between task components are required to enable the transfer of results produced by tasks which are used by the others. The channel for all these *CommunicationLink* will be *Network*.

The quality metric defined for this application is the time to detect the genes. It will depend on the size of the reference sequence and the genome, the number of sequences on the database, which are defined as load metrics, as well as the processing requirements and number of instances of each component which are described by the *QualityRules*. So, the processing and communication requirements of *formatTranslation* and *getSequences* components directly depend on the size of the genome and the number of sequences on the database, and indirectly on the processing time. The processing requirements and the number of instances of *runBLAST* and *runGeneWise* components will be determined by a combination of both genome and some input parameters such as number of similar proteins to search. Finally, the *mergeBLAST* and *mergeGeneWise* are merging reports whose size is small and almost constant. Therefore, the processing requirements are almost constant and the number of instances depends on the number of instances of the *runBLAST* and *runGeneWise* components.

The description of the installation, configuration and execution of this application is very simple; all components require to have the *compss-framework* package installed and the corresponding jars which contains the implementation of the *detectGenes* and its defined tasks. The *runBlast* and *runGeneWise* components also require having the *blast* and *genewise* packages installed. The genome and the sequences database are simple files which must be stored in a certain location. The only required configuration is stored in a *project.xml* file, which is required to run the main process (*detectGenes*). This file

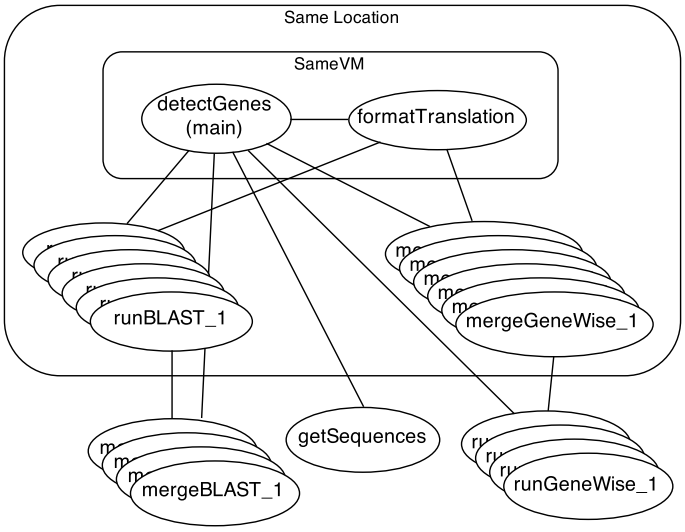


Figure 3.22 Inferred deployment model for the Gene detection application

stores the IP addresses of the hosts containing the component instances. A change in the configuration does not require a process restart.

Based on the provided topology description, the Reasoner classifies the components *detectGenes*, *getSequences* and *formatTranslation* as *Singleton* and the rest as *Scalable*. Evaluating results of applying the *QualityRules*, the Reasoner has introduced a *sameVM* affinity property between *detectGenes* and *formatTranslation* because the bandwidth and latency to perform the Genome format translation, and it has also introduced a *sameLocation* affinity property between the *formatTranslation* instance and the *runBlast* and *runGeneWise* instances because the required bandwidth is also high. Figure 3.22 depicts the inferred deployment model for the Gene Detection Application.

Map-Reduce Application

Map-reduce is a recent programming model specialized on developing Big Data applications for distributed platforms, but its users community has grown very fast during last years. For that reason, we have considered to use Map-Reduce application to validate our model. As example of how a Map-Reduce application can be described with our model, we have selected an application which performs statistical analysis of the data provided by a set of sensors. Figure 3.23 shows the overview of this map-reduce application described with the proposed application model. The *SensorDataManager* collects the sensor data and periodically starts the map-reduce execution to perform the data analysis. To describe the Map-reduce application we have defined a *Map* component which represent the map implementation and the *Reduce* component which represents the

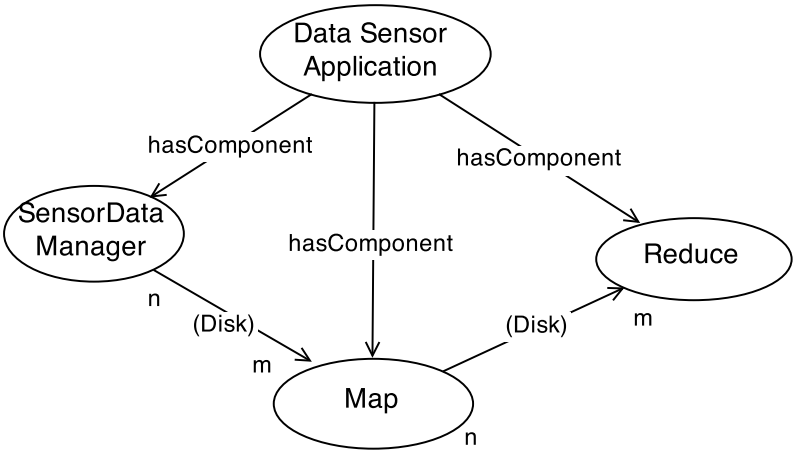


Figure 3.23 Map-Reduce application description

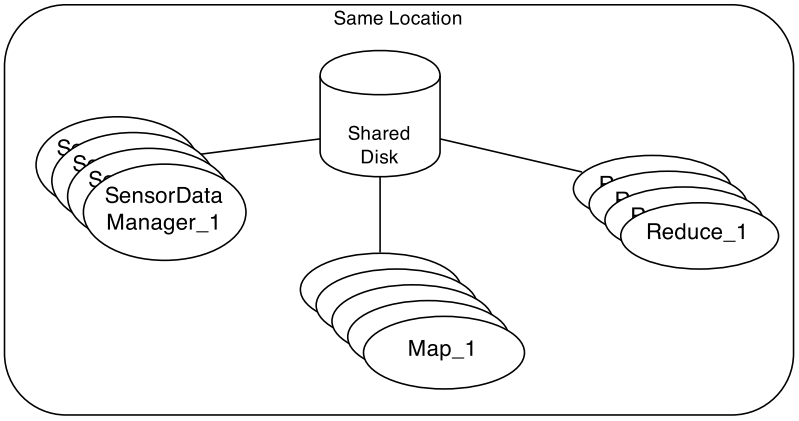


Figure 3.24 Inferred deployment model for a Map-Reduce application

reduce implementation. All the components are passing the data by means of writing files, so three Disk communication channels have been defined to model this fact.

Regarding the quality description, the number of *Map* instances depends on the number of sensors and the frequency of sensor data collections, and the number of *Reduce* instances depends on the number of *Map* instances. The number of sensors and data collection frequency will also determine the bandwidth required for disk communication.

Finally, all the components require to have installed the *hadoop* packages and the JAR files which contain the implementation of the *SensorDataManager* component as well as the *Map* and *Reduce* tasks.

Figure 3.24 shows the deployment model inferred by the Reasoner. All components have been classified as *Scalable* due to the defined topology. The quality of the disk communication links had produced the inference of the *sameLocation* affinity and the shared disk.

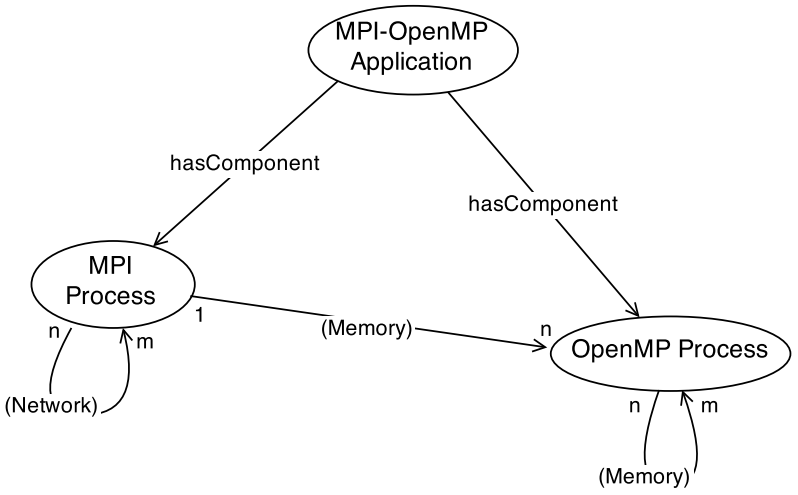


Figure 3.25 MPI-OpenMP application description

Hybrid MPI-OpenMP application

Traditionally distributed HPC applications have been developed by a combination of the MPI and OpenMP programming models. This models allow to execute efficiently applications in clusters by taking profit of shared memory in fast networks. This section provide the details about how this type of applications can be described with our proposed application model and what will be the inferred deployment model by applying the reasoning rules.

The topology of a hybrid MPI-OpenMP application is depicted in Figure 3.25. Two components have been defined to model the MPI and OpenMP processes. MPI processes communicate each other through network messages. So a *CommunicationLink* with a *Network* channel is defined to model it. Each MPI process starts a set of openMP sharing the data stored in memory. To model this fact, two *CommunicationLink* with a *Memory* channel have been defined between the MPI Process to OpenMP Process and between OpenMP Processes.

The *MPI Process* component requires to have the *mpich* package installed which provides the MPI libraries and scripts, and the *OpenMP Process* component requires to have the OpenMP libraries installed. Both MPI and OpenMP are developed as C code so installation must include the compilation of the code. Figure 3.26 shows how to specify this installation performed by means of the Autotools.

Regarding the quality description, it is very similar to the other applications. The quality of an MPI-OpenMP execution is determined by the total execution time and it will mainly depend on the amount of data to process and the number of parallel processes and their computational performance. The *OpenMP Process* and *MPI Process QualityRules*

```
:MPIProcess rdf:type app:Installation ;
    app:definedBy :MPICHPackage, :AutotoolPackage, :MPIProcessFolder .
                  :MPIPorcessConfigureExec :MPIProcessMakeExec .
:MPICHPackage rdf:type app:Package      app:name "mpich" ;
    app:ensures "installed" .
:AutotoolsPackage rdf:type app:Package      app:name "autotools" ;
    app:ensures "installed" .
:ComponentFolder rdf:type app:File ;
    app:location "/opt/mpi-process/" ;
    app:source "http://...." ;
    app:ensures "exists" .
:MPIProcessMakeFile rdf:type app:File ;
    app:location "/opt/mpi-process/" ;
    app:name “Makefile” ;
    app:ensures "exists" .
:MPIPorcessConfigureExec rdf:type app:Executable ;
    app:workingDir "/opt/mpi-process/" ;
    app:command "./configure –prefix=/opt/mpi-process/" ;
    app:generates :MPIProcessMakefile      app:ensures “executed”.
MPIPorcessMakeExec rdf:type app:Executable ;
    app:requires :MPIProcessMakefile      app:workingDir "/opt/mpi-process/" ;
    app:command "make" ;
    app:ensures “executed”.
```

Figure 3.26 Description of MPI process installation with Autotools

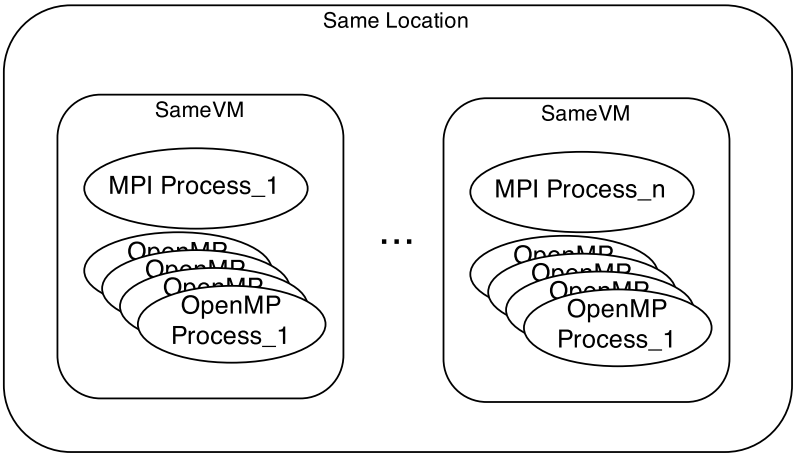


Figure 3.27 Inferred deployment model for an MPI-OpenMP application

provide the expressions to determine the number instances required and their processing requirements as a function of the amount of data to process and the target execution time. Due to the MPI model characteristics, MPI processes require network communication links with low latency and its bandwidth depends on the amount of data to process.

Figure 3.27 shows the model inferred by applying the reasoning rules on the MPI-OpenMP application description. Based on the topology, the reasoner has inferred that both components are *Scalable*. The *sameVM* affinity property has been added because of the memory communication link, and the *sameLocation* affinity property is set because of the network requirements.

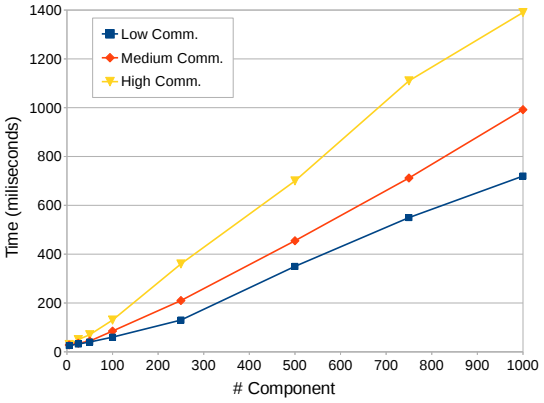
3.4.2 Overhead Evaluation

Besides the validation performed in the previous section, we have also evaluated the overhead introduced by the system prototype. This evaluation has been performed by measuring the time spent by the Reasoner to infer the deployment model for different applications. As input for the evaluation, we have defined several application descriptions with different number of components and communication links. The number of communication links is closely related to the number of components because the more components there are, the more links to communicate them there will be. However, the level of communication can be lower or higher depending on the number of communication links per component. To take it into account, we have defined three scenarios: *Low Communication* where there is one communication link per two components; *Medium Communication* where there is one communication link per component; and *High communication* where there are two communication links per component. For each component and communication link, we have defined several *QualityRules* and configuration descriptions to ensure that all the types of components and links are inferred and different numbers of instances per component are required. Regarding the number of instances, we have also defined three scenarios: *No replication* where there is one instance per component; *Low Replication* where a mean of three instances per component is inferred; *High Replication* where a mean of six instances per component is inferred.

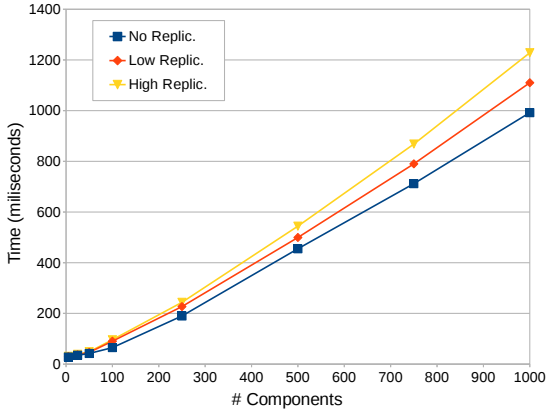
Figure 3.28 shows the results obtained for the reasoning time measurements in each scenario. As we can see in the figure, the overhead grows linearly with the number of components in all scenarios. Regarding the effect of communication and replication scenarios, a growth in the number of links has more effect than a growth in the replication ratio. The reason of this fact is because the evaluation of communication links are involved in all the phases of the reasoning process, while the number instances is just evaluated at the end of the reasoning process in order to build the final deployment model. Anyway, this overhead can be considered low if we compare it with the time spent for deploying applications in the cloud which, even for large number of components and high communication and replication scenarios.

3.4.3 Comparison to Other Application Models

Nowadays, we can find different models to describe applications for Cloud Computing, they have been briefly introduced in Chapter 2. Some of these models are very simple but either they can be only used for a single provider or they only focus on describing one



(a) Reasoning time in different communication scenarios (No replication)



(b) Reasoning time in different replication scenarios (Medium Communication)

Figure 3.28 Overhead introduced by the Application Model Reasoner for different number of components and scenarios

Table 3.1 Application Model comparison

Model	Deployment Model	Configuration, Installation & Execution	Quality Description
OVF	Manually definition of Virtual Systems whose images should contain the application components	Not Provided. Require already created images, where components have been installed and configured in advance.	Statically defined in Virtual System constraints
TOSCA	Manually assignment of components Nodes into VM Nodes	Described as *Plans* (Workflows of interface invocations). Components need to implement a management interface to allow the automatic deployment and configuration	Statically defined in Node capabilities
CloudML	Manually assignment of artifacts to VMs	Scripts written in artifact description	Statically defined in Node capabilities
mOSAiC	Automatic from Parallel Patterns	Not Provided. Require code modification to implement components as *Cloudlets* whose installation configuration and execution process is predefined	Statically defined in SLA requirement description
Proposed Model	Automatic Component topology and quality rules	Richer model which allows reasoning to detect interdependencies between component configuration, installation and execution.	Defined as rules that dynamically generate the infrastructure requirements of each component depending on the application quality and load

type of applications, such as 3-tier web applications.

Therefore, these models do not fulfill the main objective of our model of providing a portable and general-purpose application model. For performing the comparison of our model with existing ones, we have selected a subset of portable and general-purpose application models. Table 3.1 summarizes the comparison between different models and our proposal. Each column represents the different aspects evaluated in the comparison: the *Deployment Model* column describes how the models manage the application distribution in the cloud; the *Configuration, Installation & Execution* describes how the models manage the installation, configuration and execution of the application; and finally, the *Quality Description* column describes how the different models provide the application quality description. The following paragraphs provide more details about the model comparison.

The Open Virtualization Format (OVF) provides a model to describe cloud applications as a collection of virtual machine descriptions. With OVF an application must be manually broken down into different Virtual Systems which will be deployed as VM in the Cloud infrastructure. For each VM, developers have to indicate the image which contains the Operating System the application components and the required software. Therefore, developers have to build the images in advance by manually installing and configuring the application components that they want to deploy in this VM. Regarding the application quality, it is indirectly defined by setting the processing properties in the Virtual System

descriptions. So developers have to previously calculate the processing requirements for each Virtual System and set them into the description in a static way.

Trying to provide a higher level description, the OASIS standardization institution has proposed the Topology and Orchestration Specification for Cloud Applications (TOSCA). It provides model consisting on a topology of *Nodes*. Each *Node* can be a VM or an application artifact and the node relationships are used to specify dependencies, communications between artifacts or assignments of artifacts to VMs. Each node has defined a management interface, and the deployment process is modeled as a Plan, which describes a workflow of interface calls. Using this model, developers have to manually specify which component must be deployed in each VM and how the whole service is deployed. Regarding the application quality, it is defined in the node properties description, so developers have to specify them in a static way

Similar to the TOSCA model, we can find CloudML which has been implemented in conjunction of the Remics, MODAClouds and PaaSage projects. Instead of defining only Nodes, CloudML differentiates between *Artifacts*, which describe application software components, and *Nodes*, which describe VMs. But the assignment of components in the different VMs must be manually specified by the user. For each *Artifact*, developers have to provide the URL to retrieve the artifact and the commands to deploy and run it. As in the previous approaches, the quality is specified by the processing properties of the *Nodes*. So, users have to calculate them in advance and set it in a static way.

Another application model approach was proposed in the mOSAIC project, it proposes an ontology which describes application according to a set of defined parallel application patterns. Each component of this pattern is directly mapped to VMs. From the application description point of view, it describes a very simple model close to the developers' knowledge, if the application is following one of the supported patterns. Otherwise, developers have to define the components and the rules to map how components are deployed in the VMs, and in most cases, it can be complex to define. In any case, the main drawback of this option is that it does not provide a way to describe how to install, configure and execute the different components of the application. It assumes that they have been implemented as *Cloudlets* which are special abstract classes defined in the mOSAIC project and they will be deployed using the rest of the mOSAIC toolkit. So, the mOSAIC model could be a good option for creating new applications but not for migrating existing ones because it will require major code modifications.

Comparing these approaches with our proposal, we have discovered the following benefits. First, developers do not have to describe the mapping of components on VMs; it is automatically inferred by the model reasoner. Regarding the installation configuration

and execution model, our proposal does not require to make any code change to adapt to the application model as in TOSCA and mOSAIC models. Moreover, it is richer than the CloudML model. The resource state model for describing the component deployment, configuration and execution allows the reasoning to infer if a change in a configuration affects to the execution of other components. Doing something similar with the script based descriptions used by CloudML is very complex. Finally, regarding the quality description, our proposal provides a model which relates the processing constraints to the quality and load metrics in a component level. It enables the reasoner to dynamically infer the infrastructure requirements from a target quality and the application load. These requirements can be easily aggregated according to the affinity constraints in order to provide the VM requirements. In the other options, these descriptions are set statically and do not provide a way to relate the infrastructure requirements with the quality and rule metrics.

3.5 Conclusion

In this chapter, we have presented the first step to achieve an automatic migration of applications to Cloud Computing platforms, describing a methodology to automatically infer the deployment model by reasoning on a semantically-annotated application description. First, we have presented an ontology which provides a general-purpose and infrastructure-agnostic model for describing applications. Following this model, the application description is composed by three parts: the component topology, which provides the definition of the application components and their communication links; the installation, configuration and execution, which describes the desired state of the application resources after its installation, configuration and during the execution; and the quality description, which provide a set of rules to infer the processing requirements and component replicas for a given quality and application load. Once the application is described, it is loaded to a reasoner which applies the different rules to classify the components and communication links and the quality rules to extract the component and link requirements. Based on this classification and the extracted requirements, the reasoner also infers the implicit affinity constraints.

A prototype of this approach has been implemented, validated and evaluated. To validate the model, we have described four applications form four types: n-tier, task-based, map-reduce and MPI/OpenMP. We have evaluated the reasoning overhead by measuring the time to infer the deployment model in different scenarios. We have observed the overhead is growing linearly with the number of components and even for

large application the overhead is just few seconds. This overhead can be considered low compared with the time required to deploy VMs on the Cloud Infrastructures, and much faster compared with the time spent by a developer to do the same inference manually.

Finally, our proposal has been compared with other cloud application models (OVF, TOSCA, CloudML and mOSAIC). The main advantage of our approach is that developers do not have to specify how the application is distributed in VMs because the application model reasoner infers it. In our approach the application code remains unchanged, because the model is describing the required installation, configuration and execution procedure, instead of forcing developers to implement management interfaces, as in TOSCA, or implementing components as *Cloudlets*, as in mOSAIC. Regarding the application quality, other approaches indirectly specify the required application quality by statically setting the processing requirements of the VM. In our case, the processing requirements are described by rules, which provides the component requirements as function of the quality and load metrics.

Chapter 4

Semantic Resource Allocation

Once a deployment model for an application has been inferred, the next step is allocating the different components on the available resources. Despite of this work is used for allocating application components in the available resources within the overall thesis storyline, it was started as a Grid scheduler because at the time when we started this work the multi-cloud environments did not exist. Later on, due to the problem similarities, we extended the scheduler to cover other allocation problems which are present on the distributed computing platforms, in order to provide an extensible general-purpose resource allocation solution. For instance, the solution is applied to the already mentioned allocation of application components in the VM types in a multi-cloud scenario, to the allocation of VM deployments on the physical resources of a cloud provider data center, and to the scheduling of jobs in grids or a cluster. Therefore, this chapter of the thesis tackles the problem of the resource allocation in distributed computing platforms.

The rest of the chapter is organized as following: First, Section 4.1 presents an overview of the methodology used to achieve the Semantic Resource Allocation and Section 4.2 and Section 4.3 provide more details about the Resource Allocation Ontology and the allocation process. Then, Section 4.4 shows how the solution is applied for different problems, presents the results of an overhead evaluation and discusses about the benefits provided by these solutions with respect to other allocation solutions. The chapter is finalized by Section 4.5, where we draw the conclusions about this field.

4.1 Methodology

Distributed platforms such as Grids and Clouds provide an ecosystem where different providers offer their heterogeneous resources to users which are willing to execute their applications. The different users' applications have different constraints while the

providers' resources offer different capabilities. Every entity in the ecosystem has their own way to express their needs and capabilities which makes difficult to integrate them in a uniform way to enable a proper assignment of resources to the users' applications in an automatic way. In contrast, semantic web technologies provide a framework to describe web resources in a structured, well-defined and machine understandable way. In this chapter, we aim at applying semantic web technologies to the resource allocation problem in order to enable the assignment of resources to applications which are coming from different sources.

To do it, we have defined a Resource Allocation Ontology to model applications, resources and other concepts involved on the resource allocations. Despite of users' applications are different, they can be modeled in a uniformed way as a set of computing tasks which must be executed on a resource with certain capabilities. The same happens with resources, despite they are heterogeneous they can be described as a set of capabilities and properties. Then, allocation policies are modeled as horn rules which describe how the resources are allocated to computing tasks according to the task requirements and resource capabilities and properties. The final allocation solution is obtained by applying these rules on a knowledge base composed by task descriptions and the available resource descriptions. Next section provides more details about the Resource Allocation Ontology and the rule-based allocation process.

4.2 Resource Allocation Ontology

Figure 4.1 shows the overview of the proposed Resource Allocation ontology, it is composed by three main parts: the resource description, which models the capabilities and properties of the resources; the task description, which models the computation which is going to use resources; and finally, the allocation description which models the assignment of resources to the different tasks.

Regarding the first part, there are several schemes and models in the literature which propose similar descriptions of computing resources such as GLUE [111], the Common Information Model (CIM) [112] and the Grid Resource Ontology (GRO) [113]. These proposals describe the capabilities of computing resources in a more o less extended way. However, they do not describe how the different resource capabilities can be used by the different tasks. The work presented in [114], provides an ontology to describe the coordination of resource usage. They propose a set of interesting resource properties such as *sharable*, if a resource can be shared by several users, *consumable*, if the usage of this resource can make it unavailable and *clonable*, if the resource can be copied.

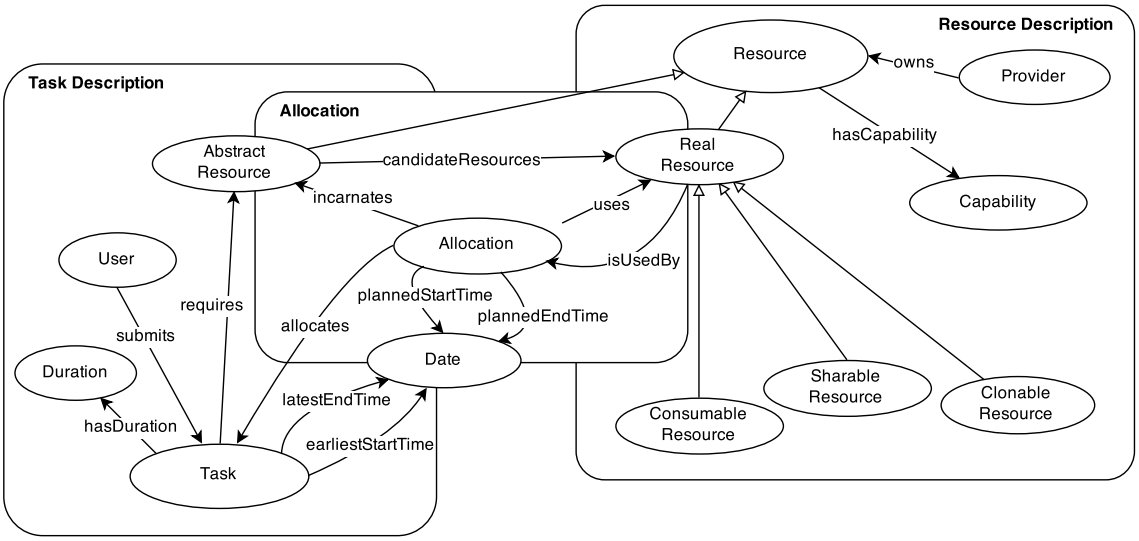


Figure 4.1 Resource Allocation Ontology

Another missing concept is the description of images and software resources. Resource providers describe images and the available software resources as plain text which is easily understood by users but difficult to be automatically processed by computers. The most important information that an image should contain, includes the operating system and the state of its software resources. For that reason, we propose to use the same description as used in Section 3.2.3 for describing the components installation and configuration. The resource description part, that we propose for the Resource Allocation Ontology, combines the basic computing capabilities description and resource definition proposed by the previous resource schemes, together with a resource classification based on the defined resource sharing properties. The basic resource description has been extended to support the cloud resources, such as VMs and images, as mentioned above.

The second part is focused on the tasks' description. This part should describe the required resource capabilities and time constraints to perform the computation as users expect. To perform this description we propose to model these requirements as a definition abstract resource indicating the minimum required capabilities, such as the minimum quantity of memory, disk, network and processors. If a task has time constraints, they can be modeled by the specification the earliest initial time, latest end time and the expected duration properties.

Finally, the third part is about modeling the allocation of resources to tasks. Each abstract resource requirement defined in tasks is related by *candidateResources* property to the real resources which fulfills its minimum capabilities. At the end of the allocation process one of this candidate resources, will be finally allocated to a task by means of an

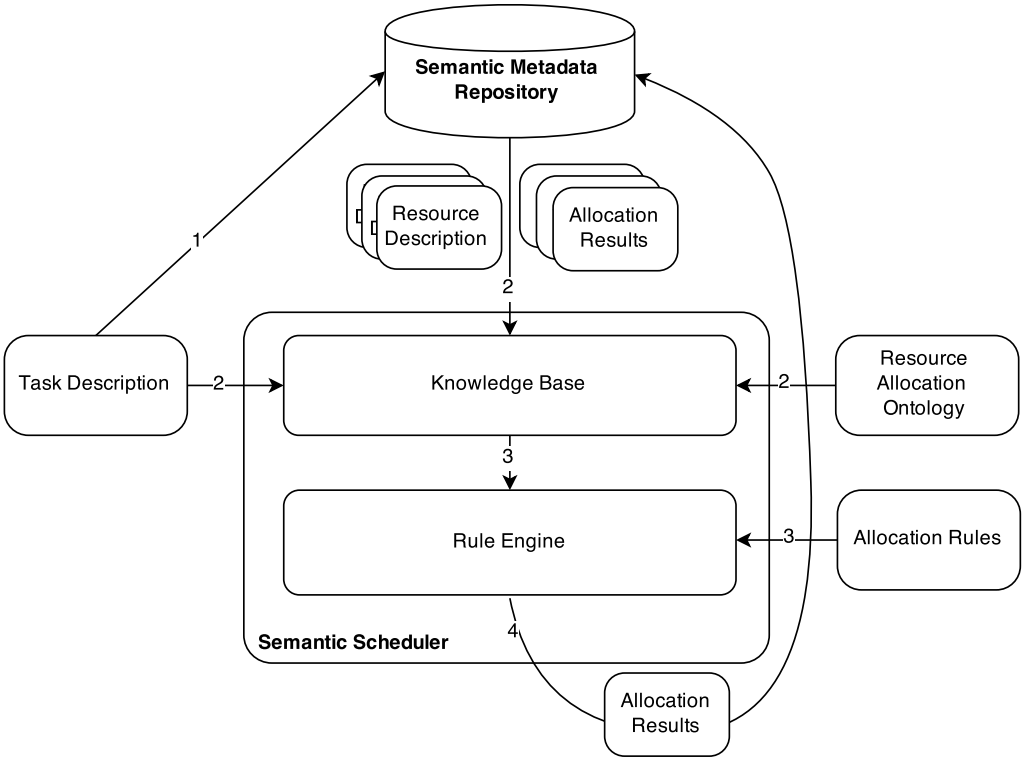


Figure 4.2 Rule-driven Resource Allocation Architecture

Allocation description which models usage performed by a task in a selected real resource (by means of the *uses* property) during a period of time (by means of *plannedStartTime* and *plannedEndTime*.

The Resource Allocation Ontology is complemented with a set of rules which model the allocation policies. These rules are in charge of inferring the best resource allocation according to the defined task requirements and resource capabilities. Next sections provide more details of this rule-driven resource allocation process and its applicability in different scenarios.

4.3 Rule-driven resource allocation

Figure 4.2 shows an overview of the semantic resource allocation process which allocates resources on tasks by means of an ontology and horn rule reasoning. Two components are involved in this allocation process: the Semantic Metadata Repository (SMR), which is in charge of storing the resources and tasks descriptions as well as the previous allocation results; and the Semantic Scheduler (SeS), which is in charge of allocating and scheduling tasks on the available resources. When a new task arrives to the

system, it is stored on the SMR and a SPARQL query is generated to retrieve the available real resources which fulfill the required abstract resource defined on the task description.

The retrieved resource descriptions are linked to the task description requirements by setting them in the *candidateResources* property. Moreover, these resources are already linked with its current allocations. All this information, together with the Resource Allocation Ontology, is introduced into the SeS creating an inference model. This model is attached to a rule engine which will evaluate the allocation policy rules over the inference model data and stores the inferred data in a deductions graph. The allocation rules are evaluated in a cascade mode performing the following actions: First, detecting the shared candidate resources and generating the different possible allocations; then, resolving the conflicts with existing tasks' allocations scheduled at the same resource, and by discarding, sharing or cloning resources according to the resource type (*sharable consumable* and *clonable*); and finally, ranking the remaining allocation and selecting the best. Figure 4.3 shows an example of each type of rule. The first rule provides an example of allocation generation rule. The evaluation of this rule will generate an allocation at the earliest time for each candidate resource. The second rule provides an example of the conflict resolution. This rule will drop the generated allocation if there is an allocation of a task on a non-sharable resource. Finally, the third rule provides an example of selection rule, where the *plannedEndTime* property of two remaining allocations is compared, and the allocation with the latest *plannedEndTime* is removed.

When the inference process has finished, the SeS inspect the deductions graph by looking for the inferred *Allocation* instances which includes the selected resource and the assigned time slot. The SeS interprets the different allocation results appearing in the deduction graph as a new allocation or reallocation of tasks in resources and updates the corresponding task and resource descriptions stored in the SMR.

4.4 Evaluation and Discussion

The evaluation of the Semantic Resource Allocation consists of three different aspects: first, we evaluate how the solution can be applied in different scheduling problems; second, we have evaluated the overhead and scalability of the solution by calculating the time to allocate resources to tasks in different scenarios; and third, we will discuss the benefits and drawbacks of using this technology with respect to current options.

```
Rule { ## Initial Allocation
    ?task rdf:type rao:Task
    ?task rao:status rao:Submitted
    ?task rao:earliestStartTime ?esTime
    ?task rao:latestEndTime ?leTime
    ?task rao:hasDuration ?duration
    sum(?esTime, ?duration, ?endTime)
    lessThan(endTime,leTime)
    ?task rao:requires ?abstResource
    ?abstResource rao:candidateResource ?resource
    makeNode(?Allocation)
=>
    ?allocation rdf:type rao:Allocation
    ?allocation rao:plannedStarTime ?esTime
    ?allocation rao:plannedEndTime ?endTime
    ?allocation rao:incarnates ?abstResource
    ?allocation rao:allocates ?task
    ?allocation rao:uses ?resource
}
...
Rule { ## Drop collided allocation on non-sharable resources
    ?newAlloc rdf:type rao:Allocation
    ?newAlloc rao:allocates ?task
    ?task rao:status rao:Submitted
    ?newAlloc rao:uses ?resource
    ?existAlloc rdf:type rao:Allocation
    ?existAlloc rao:allocates ?task2
    ?existAlloc rao:uses ?resource
    ?task rao:status rao:Scheduled
    ?resource rao:sharable false
=>
    drop(?newAlloc)
}
...
Rule { ## Select the allocation with earliest deadline
    ?alloc1 rdf:type rao:Allocation
    ?alloc1 rao:uses ?resource
    ?alloc1 rao:allocates ?task
    ?alloc1 rao:plannedEndTime ?endTime1
    ?alloc2 rdf:type rao:Allocation
    ?alloc2 rao:uses ?resource
    ?alloc2 rao:allocates ?task
    ?alloc2 rao:plannedEndTime ?endTime2
    ?task rao:status rao:Submitted
    lessThan(?endTime1, ?endTime2)
=>
    drop(?alloc2)
}
```

Figure 4.3 Allocation rules examples

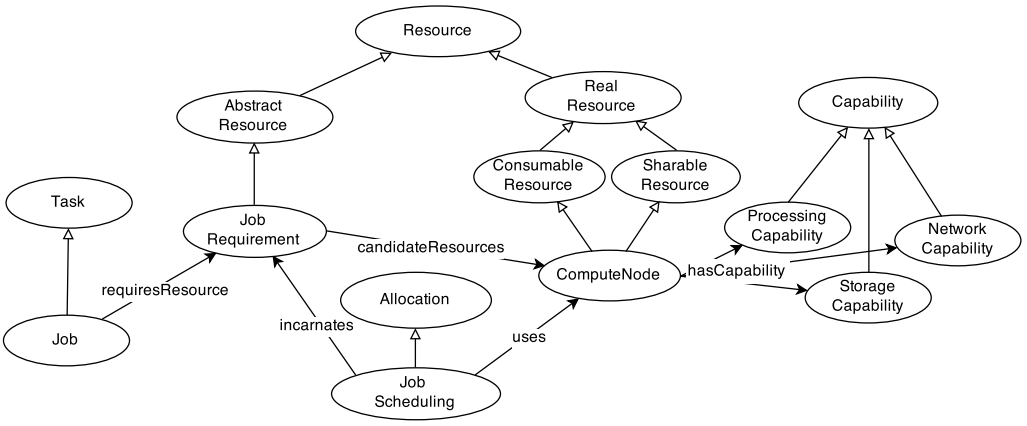


Figure 4.4 Map of Job scheduling in Grid Computing to the Resource Allocation Ontology

4.4.1 Applicability

Previous sections have provided an overview of the ontology and the rule-based approach for resource allocation. It provides a general and extensible solution for allocation which can be applied in the different allocation problems existing in current distributed platforms. Next paragraphs provide examples of how the semantic resource allocation solution can be applied to three different problems: the scheduling of jobs in computing nodes in a Grid Computing scenario, the allocation of VMs in the Data Center servers in a private Cloud scenario and the allocation of the application components on the VM types in a multi-cloud scenario. In each scenario, we show how the different elements are mapped to the ontology classes and which allocation policy rules are defined to describe the allocation generation, evaluation and selection rules according to the type defined tasks and resources.

Scheduling of Jobs in a Grid Computing Scenario

In a Grid computing scenario, applications are composed by a set of batch jobs which are scheduled in the different computing nodes available in the Grid infrastructure. Figure 4.4 shows how the different elements of job scheduling are mapped to the Resource Allocation Ontology. First, a *Compute Node* is defined as a *Sharable* and *Consumable* resource, but not *Clonable* because *Compute Nodes* can not be replicated. The node computing capabilities (processing, storage and network) are defined as subclass of *Capability*. Second, *Jobs* are defined as a subclass of task and its *Job Requirements* are defined as a subclass of *Abstract Resources* and the *earliestEndTime* and *duration* properties are used to describe the job deadline and the wall-clock limit

```
Rule { ## Allocation after existing allocated tasks
    ?task rdf:type rao:Task
    ?task rao:status rao:Submitted
    ?task rao:earliestStartTime ?est
    ?task rao:hasDuration ?duration
    ?task rao:latestEndTime ?leTime
    ?task rao:requires ?abstResource
    ?abstResource rao:candidateResource ?resource
    ?allocation rdf:type rao:Allocation
    ?allocation rao:plannedEndTime ?peTime
    ?allocation rao:uses ?resource
    ?allocation rao:allocates ?task2
    ?task2 rao:status rao:Scheduled
    sum(?peTime, ?duration, ?endTime)
    lessThan(endTime,leTime)
    makeNode(?newAlloc)
=>
    ?newAlloc rdf:type rao:Allocation
    ?newAlloc rao:plannedStarTime ?peTime
    ?newAlloc rao:plannedEndTime ?endTime
    ?newAlloc rao:incarnates ?abstResource
    ?newAlloc rao:allocates ?task
    ?newAlloc rao:uses ?resource
}
```

Figure 4.5 Rules to generate possible allocations in the Grid scheduling scenario.

respectively. Finally, *Job Schedulings* are defined as *Allocations* where the selected *Compute Node* is specified by the *uses* property and the assigned time-slot is defined with the *plannedStartTime* and *plannedEndTime* properties.

Regarding allocation rules to perform the job scheduling, we need to define a set of rules which model the allocation policies. First, a set of rules is needed to generate the possible allocations. In this case, a possible allocation is generated for each candidate resource at the earliest time and after the end time of tasks previously allocated in the same resource. They are generated by applying the first example rule of Figure 4.3 in Section 4.3 and the rule described in Figure 4.5, which generates the allocations after the end of the previously allocated task in the candidate resources.

Due to the resources in this situation are always *Sharable* and *Consumable* but no *Clonable*, conflicts are solved by dropping the allocations which collide with previous allocation and the resource capacity is exceeded. Figure 4.6 describes the rule which implements this feature. In this case, we have implemented two rule built-ins, which are parts of code to perform some validation or calculation over the rule facts. The first one, *overlapped*, evaluates if two allocations are overlapped on time and returns the overlapped time slot. The other one, *capacityExceeded*, checks if the capacity of a resource is exceeded in a given time slot. It is calculated by summing the abstract resource requirements of the allocated tasks. Finally, we select the best remaining allocation as the one which is planned to end before. The description of this rule can be found in the third example rule of Figure 4.3 in Section 4.3.

```
Rule { ## Drop allocation which exceeded capacity when collide with another overlapped allocation
    ?newAlloc rdf:type rao:Allocation
    ?newAlloc rao:uses ?resource
    ?newAlloc rao:allocates ?task
    ?task rao:status rao:Submitted
    ?existAlloc rdf:type rao:Allocation
    ?existAlloc rao:uses ?resource
    ?existAlloc rao:allocates ?task2      ?task2 rao:status rao:Scheduled
    ?resource rao:sharable true
    ?resource rao:consumable true
    ?resource rao:clonable false
## Check if overlapped by evaluating plannedEndTime and plannedStarTime
    overlapped(?newAlloc, ?existAlloc, ?startOverlap, ?endOverlap)
## Check if resource capacity is exceeded during newAlloc interval
    capacityExceeded(?resource, ?startOverlap, ?endOverlap)
=>
    drop(?newAlloc)
}
```

Figure 4.6 Rule to drop allocations which exceeded resource capacity in the Grid scheduling scenario

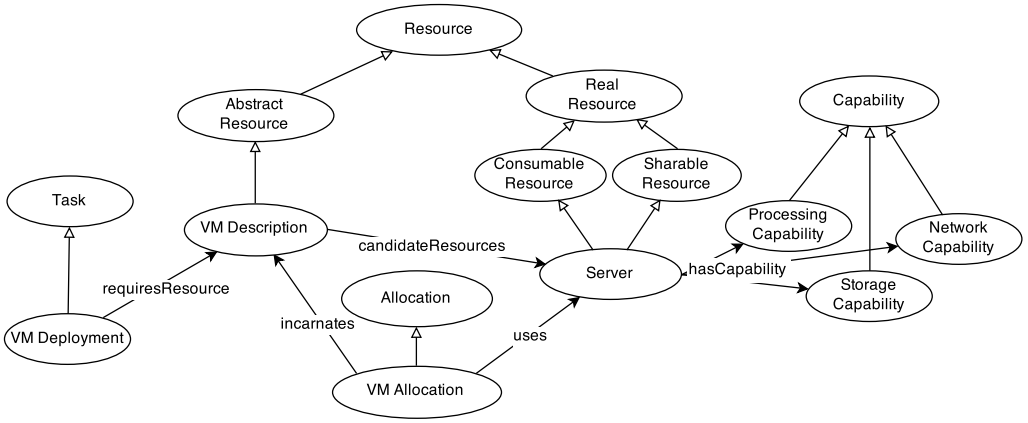


Figure 4.7 Map of VM deployment allocation in Private Clouds to the Resource Allocation Ontology

Allocation of VMs in a Private Cloud Scenario

The Semantic Resource Allocation can be also applied in a Private Cloud Scenario where the deployments of VMs are allocated in the different servers of the provider's data center. Figure 4.7 shows how the elements of the scenario are mapped to the defined classes of the Resource Allocation Ontology. In this case, a *VM Deployment* is defined as a subclass of *Task*. The difference with grid jobs, is that the time that a VM remains deployed is undetermined. Therefore, there are no time constraints defined for this type of task and as consequence no time slots are defined for the VM allocation. On the other hand, *Servers* are defined as *Sharable* an *Consumable* resources, but not *Clonable* because they can not be replicated, in the same way that we have done for the *Compute Nodes* in the Grid scenario.

```
Rule {## Generate Allocation per Candidate Resource
    ?task rdf:type rao:Task
    ?task rao:status rao:Submitted
    ?task rao:requires ?abstResource
    ?abstResource rao:candidateResource ?resource
    makeNode(?Allocation)
=>
    ?allocation rdf:type rao:Allocation
    ?allocation rao:incarnates ?abstResource
    ?allocation rao:allocates ?task
    ?allocation rao:uses ?resource
}
```

Figure 4.8 Rule to generate allocations in the Private Cloud scenario

```
Rule { ## Drop allocation which exceeded capacity when collide with another overlapped allocation on
    ?newAlloc rdf:type rao:Allocation
    ?newAlloc rao:uses ?resource
    ?newAlloc rao:allocates ?task
    ?task rao:status rao:Submitted
    ?existAlloc rdf:type rao:Allocation
    ?existAlloc rao:uses ?resource
    ?existAlloc rao:allocates ?task2      ?task2 rao:status rao:Scheduled
    ?resource rao:sharable true
    ?resource rao:consumable true
    ?resource rao:clonable false
## Check if resource capacity is exceeded with scheduled tasks
    capacityExceeded(?resource)
=>
    drop(?newAlloc)
}
```

Figure 4.9 Rule to drop allocations which exceeded resource capacity in the Private Cloud scenario.

Regarding the rules to allocate *VM Deployments* on *Servers*, they are more simple than the used in the Grid job scheduling case because they do not have to deal with the time slots. To generate the possible allocations, we just need a rule which creates a possible allocation per real resource selected as *candidateResource*, as described in Figure 4.8.

Once the allocations have been generated, we have to remove the allocations that exceeds the resource capabilities when sharing the resource with current allocations. The rule to perform it is almost the same than the used for Job Scheduling (Figure 4.6) with the difference that there is no need to check the overlap between allocations because there are not time slots defined. The rule used in this case is shown in Figure 4.9.

Finally, a last rule should decide which is the best of the remaining allocations. This rule will depend on what is preferred policy for the provider. It can prefer to balance the VM distribution or allocate them in the minimum of servers. Figure 4.10 shows the rules which implements those policies. The first one drops the allocation which already contains more allocations, and the second one drops the allocation to resources where there are no previous allocations.

```
Rule { ## Select the resource with less allocations
    ?alloc1 rdf:type rao:Allocation
    ?alloc1 rao:uses ?resource1
    ?alloc1 rao:allocates ?task
    ?resource1 rao:usedBy all(?allocs1)
    listLength(?allocs1, ?length1)
    ?alloc2 rdf:type rao:Allocation
    ?alloc2 rao:uses ?resource2
    ?alloc2 rao:allocates ?task
    ?resource2 rao:usedBy all(?allocs2)
    listLength(?allocs2, ?length2)
    ?task rao:status rao:Submitted
    lessThan(?length1, ?length2)
=>
    drop(?alloc2)
}

Rule { ## Select the resource which already has allocations
    ?alloc1 rdf:type rao:Allocation
    ?alloc1 rao:uses ?resource1
    ?alloc1 rao:allocates ?task
    ?resource1 rao:usedBy ?alloc)
    ?alloc2 rdf:type rao:Allocation
    ?alloc2 rao:uses ?resource2
    ?alloc2 rao:allocates ?task
    noValue(?resource2 rao:usedBy)
    ?task rao:status rao:Submitted
=>
    drop(?alloc2)
}
```

Figure 4.10 Allocation selection rules examples for Private Cloud scenario

Allocation of VMs in a Multi-Cloud Scenario

The final applicability example is the allocation of the components belonging to an application deployment in the VM types offered by Infrastructure providers. Figure 4.11 shows how the elements in this scenario map with the classes of the Resource Allocation Ontology. For this scenario, an *Application Deployment* is mapped as subclass of *Task*. Each application will require a set of *Components* which describe the abstract resources required by each of the application components. In the real resource description part, we have defined *VM Types* as a *Clonable*, *Sharable* and *Consumable* real resource which in addition to the computing capabilities, it also has other properties such as the price or the location which will be used to select which is the best allocation. Finally, the *Component Allocation* is defined as a subtask of *Allocation* which incarnates a *Component* using a *VM Type*. For the same reason as in the private cloud scenario, *Application Deployments* do not define time constraints and *Component Allocation* neither define time slots.

Regarding how to get the solution for a component allocation problem, we can follow the same procedure as in the other cases to generate a possible allocation per candidate *VM Type* and select the best allocation, for instance, by evaluating the price. In this case, we do not need to have any rule to evaluate capabilities exceed because *VM Type* resources are defined as *Clonable*, and each allocation will have their own *VM Type* instance. So, rules,

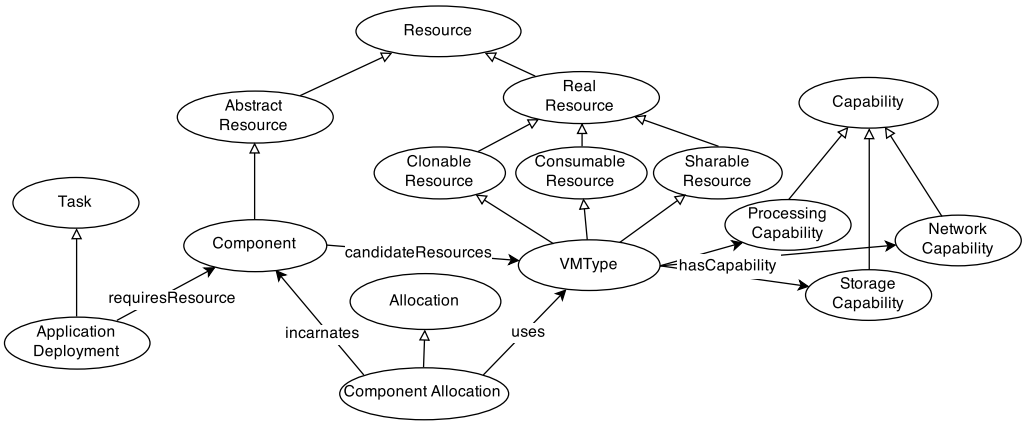


Figure 4.11 Map of Application components allocation in Multi-Cloud to the Resource Allocation Ontology

defined in Figure 4.12, model the allocation of components in the multi-cloud scenario as explained above.

The evaluation of these rules will provide the best allocation for each component. However, a better assignment could be found if some components are assigned together in the same VM type. To include this feature in the allocation, we just need to add another set of rules to generate new application deployments by merging two components and another to compare two application deployments and drop the one which provide a worse allocation. This rules are described in Figure 4.13.

4.4.2 Overhead and Scalability Evaluation

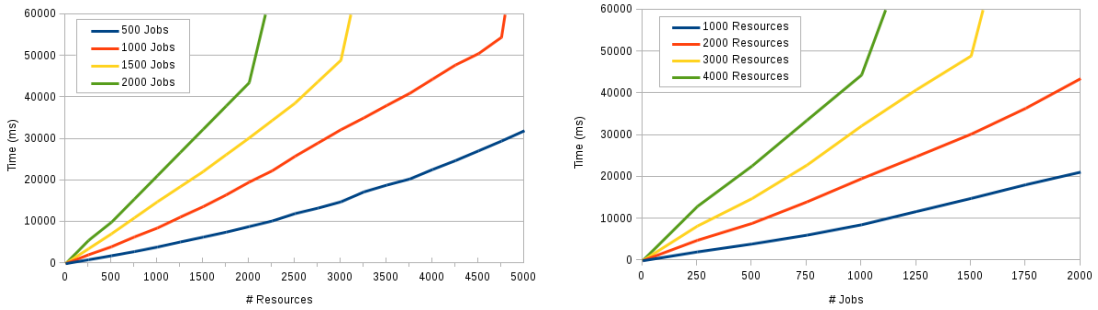
The second part of the Semantic Resource Allocation evaluation focuses on the analysis of the overhead introduced by the system. We have measured the time to make the allocation in different scenarios and how it grows with the number of tasks and resources. To perform this analysis, a prototype of the Semantic Resource allocation system has been implemented using the Apache Jena [115]. It is a Java semantic web framework which provides a repository for semantic RDF description, a set of API to manage RDF documents and make SPARQL queries and an ontology reasoner and rule engine to perform the semantic reasoning. The Semantic Metadata Repository has been implemented as a Jena RDF repository using the SPARQL API to make the queries to get the candidate resources, and the Semantic Scheduler has been implemented by creating a Jena Model with the RDF descriptions and Resource Allocation Ontology which is attached to a the Jena Forward Rule reasoner which implements the RETE algorithm [116] for performing an efficient rule evaluation. The implemented prototype has been deployed

```
Rule { ## Generate Allocation per Candidate Resource
    ?task rdf:type rao:Task
    ?task rao:status rao:Submitted
    ?task rao:requires ?abstResource
    ?abstResource rao:candidateResource ?resource
    makeNode(?Allocation)
=>
    ?allocation rdf:type rao:Allocation
    ?allocation rao:incarnates ?abstResource
    ?allocation rao:allocates ?task
    ?allocation rao:uses ?resource
}

Rule { ## Select the resource with less price
    ?alloc1 rdf:type rao:Allocation
    ?alloc1 rao:uses ?resource1
    ?resource1 rao:hasPrice ?price1
    ?alloc1 rao:incarnates ?abstResource
    ?alloc1 rao:allocates ?task
    ?alloc2 rdf:type rao:Allocation
    ?alloc2 rao:uses ?resource2
    ?resource2 rao:hasPrice ?price2
    ?alloc1 rao:incarnates ?abstResource
    ?alloc2 rao:allocates ?task
    ?task rao:status rao:Submitted
    lessThan(?price1, ?price2)
=>
    drop(?alloc2)
}
```

Figure 4.12 Rules to generate and select the best allocation for the Multi-Cloud scenario

```
Rule { ## Generate new Application Deployments by merging components
    ?appDeploy rdf:type mc:ApplicationDeployment
    ?appDeploy rao:status rao:Submitted
    ?appDeploy rao:requires ?abstResource1
    ?appDeploy rdf:type mc:ApplicationDeployment
    ?appDeploy rao:requires ?abstResource2
    cloneNode(?appDeploy, ?newAppDeploy)
    mergeResources(?abstResource1,?abstResource2, ?newAbstResource)
=>
    drop(?newAppDeploy rao:requires ?abstResource1)
    drop(?newAppDeploy rao:requires ?abstResource2)
    ?newTask rao:requires ?newAbstResource
}

Rule { ## Select the Application with less price less allocations
    ?appDeploy1 rdf:type mc:ApplicationDeployment
    calculateTotalPrice(?appDeploy1, ?price1)
    ?appDeploy2 rdf:type mc:ApplicationDeployment
    calculateTotalPrice(?appDeploy2, ?price2)
    lessThan(?price1, ?price2)
=>
    drop(?appDeploy2)
}
```

Figure 4.13 Rules to generate and select the best application deployment for the Multi-Cloud scenario



(a) Allocation time depending on the number of resources (b) Allocation time depending on the number of jobs

Figure 4.14 Overhead introduced by the Semantic Resource Allocation system for scheduling jobs in a grid scenario

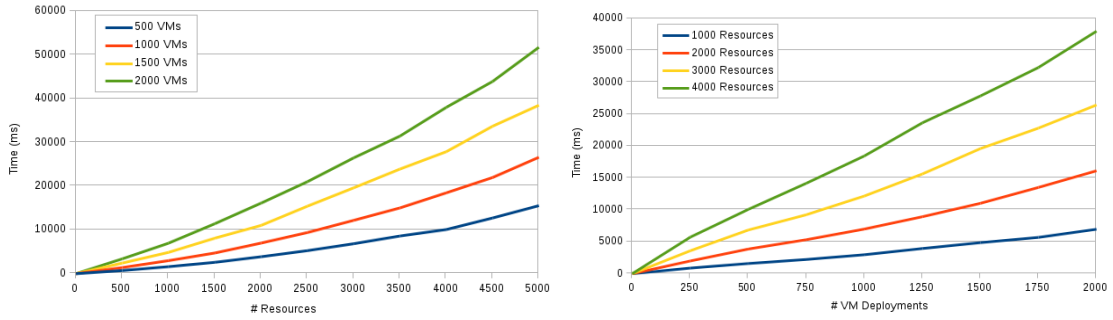
in a laptop with an Intel Core i7 processor and 8GB of RAM with Java 7 and the Apache Tomcat installed to run the semantic repository and semantic scheduler processes.

To evaluate the behavior of the Semantic Resource Allocation system, we have generated different task and resource descriptions for each of the scenarios presented in the previous section, and we have measured the time to allocate tasks on resources for different number of tasks and resources in order to know the time scales and compare it with other overheads and the duration of computational tasks

Figure 4.14 shows the evaluation results for the Grid scheduling scenario. In the plots of the figures, we can see the allocation time grows almost linearly with the number of tasks and resources. The time scale for this range of tasks is about milliseconds for small applications with few jobs and can take seconds for very large applications. These overheads can be assumable if we compare with the job duration and other overheads introduced by grid middleware. Grid jobs are coarse-grained batch executions, whose duration must be in the order of minutes to compensate large-distance data transfers and queue waiting times. So, adding some milliseconds for a single job or some seconds for large applications will not have a very big impact in the application performance.

Note that when a large number of tasks and resources is scheduled, the time is starting to be increased very fast. This is because the memory allocated to the Java Virtual Machine is exhausted and the Java Runtime Environment is spending a lot of time trying to make garbage collection which considerably slows the scheduling process. In this situation, the overhead can reach not assumable boundaries.

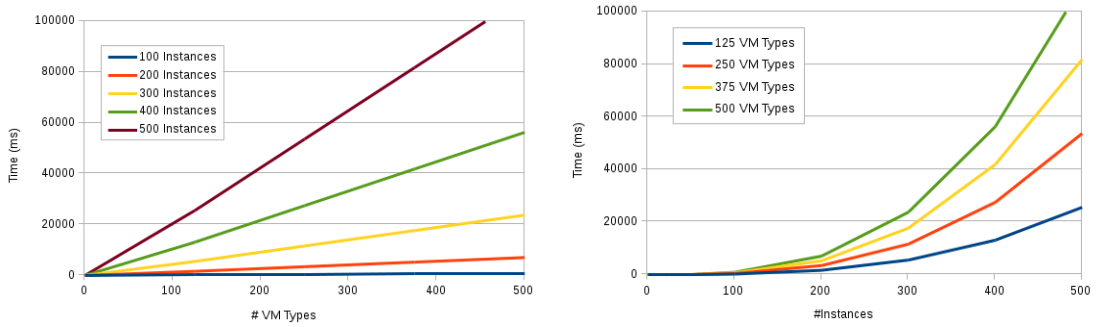
Similar results are found for the private cloud scenario which are depicted in Figure 4.15. In this case, the slope of the growth is smaller due to the number of possible allocations per resource is smaller. So, in the case of a private cloud, the scheduler only has to evaluate an allocation per resource and task, while in the grid scheduling, we have



(a) Allocation time depending on the number of resources

(b) Allocation time depending on the number of VM deployments

Figure 4.15 Overhead introduced by the Semantic Resource Allocation system for allocating VMs in a private cloud scenario



(a) Allocation time depending on the number of VM types

(b) Allocation time depending on the number of service component instances

Figure 4.16 Overhead introduced by the Semantic Resource Allocation system for allocating service component instances in VM Types in a public cloud scenario

to evaluate the allocation proposed after each job scheduled at a resource. For the same reason, the allocation is faster and consumes less memory, therefore we do not observe the memory issues in the plots of the figures.

The last scenario is the public cloud, in this case the allocation time grows linearly for resources but polynomially for the tasks. It is because of the merge rule which merges component instances in order to find better solutions than just allocating each instance in a VM type. Despite the polynomial growth, the overhead is still assumable compared to the application deployment. For small and normal applications (up to 200 instances) the allocation overhead is about in the order of milliseconds or few seconds. However, the deployment of application VMs and booting the Operating System will easily take some minutes. For larger applications, the allocation overhead will grow up to taking minutes, but the deployment is also growing due to the number of application VMs. Just in the case

of very large applications, with thousands of components, the allocation overhead can be important.

4.4.3 Benefit from traditional scheduling approaches

The semantic resource allocation solution proposed in this thesis has several benefits in comparison with the traditional scheduling solutions. Most of schedulers are tailored for a specific problem with the specific workload and platform such as batch job schedulers for clusters and grids or VM schedulers in data centers. They do not allow extensions or it must be done by means of implementing plug-ins. Benefiting from semantic web technologies, our solution provide a way to easily integrate resources and tasks from different sources and the same engine can extended and used without requiring any code modifications or plug-in implementation. As we have seen in previous sections, users just need to map the allocation problem in the tasks and resource and provide the rules which model the allocation policies. Moreover, rules can be loaded and unloaded at runtime, so according to the type of tasks and resources, we can select the rules which provide scheduling results enabling the adaptation of the scheduling to the current load.

4.5 Conclusion

In this chapter, we have studied how semantic web technologies can be applied in the problem allocating application components in computing resources which is the second part for achieving an automatic application deployment. In this case, we have defined a Resource Allocation Ontology, which models the concepts involved on the assignment of resources to the computing tasks. Allocation policies are modeled as horn rules which are a knowledge base composed by the available resources and requested task descriptions.

The evaluation of the proposed solution has been focused in two parts. First, to validate that the solution can be easily used in different scenarios, we have applied it in the scheduling of jobs in Grid environments, the allocation of VM deployments in a Private Cloud environment, and the allocation of application components in the different VM types in a multi-cloud environment. Second, we have measured the overhead introduced by the system in the different scenarios and how it grows with respect of the number of tasks and resources. The overhead is directly related to the complexity of the allocation but in all the cases it is assumable when we compare to other expected overhead or task durations such as queue waiting times and deployments. However, in situations such as grid environments, where the number of available resources and tasks is high, we start to run out of memory and the overhead will increase fast reaching not assumable boundaries.

A solution for this problem is found in Chapter 6, where a distributed resource allocation approach is proposed.

Comparing our solution with other scheduling approaches, it easily integrates resources and tasks from different sources and the same engine can extended and used without re-implementations or plug-ins. Moreover, rules can be loaded and unloaded at runtime, so it enables the adaptation of the scheduling to the current load by selecting the proper scheduling rules for the type of tasks and resources

Chapter 5

Infrastructure Interoperability

Once the deployment model and placement for an application has been inferred, the last step for achieving the automatic deployment is the provisioning and configuration of the resources needed to execute the application components. To perform this step for a single provider, developers have to know which kind of resources are offered by the infrastructure provider, learn how to use its' API and implement a workflow for deploying the computing resources as well as installing and configuring the application components in the deployed resources. Despite it is a complex process, it could be assumable for a single provider. However, each provider offers their own interface. So, if for any reason, developers want to change the provider or combine some of them to perform a more convenient deployment for their applications, they should learn how to use the new API, introduce changes in their deployment code to support the new providers' APIs, multiplying the required development effort and its associated cost. This fact dissuades developers to change the infrastructure provider producing a vendor lock-in.

Solving this provider lock-in issues relies on the ability of making different cloud infrastructures work together. One key issue in this topic is the knowledge and the correct interpretation of the interfaces offered by infrastructure providers. These interfaces are composed by a set of remote methods which exposes the available provisioning and management actions, and a data model which describe the offered resources and the method input and output parameters. When users want to perform a remote action exposed in the API, they have to exchange messages with the provider's services using the protocol, format and model specified by the provider. Despite most of providers offer similar functionalities, protocols, methods and data models defined are different. Therefore, to enable the interoperation between different providers and solving the vendor lock-in, we should be capable to automatically find and invoke the correct methods in each provider's API, which performs the desired resource management tasks.

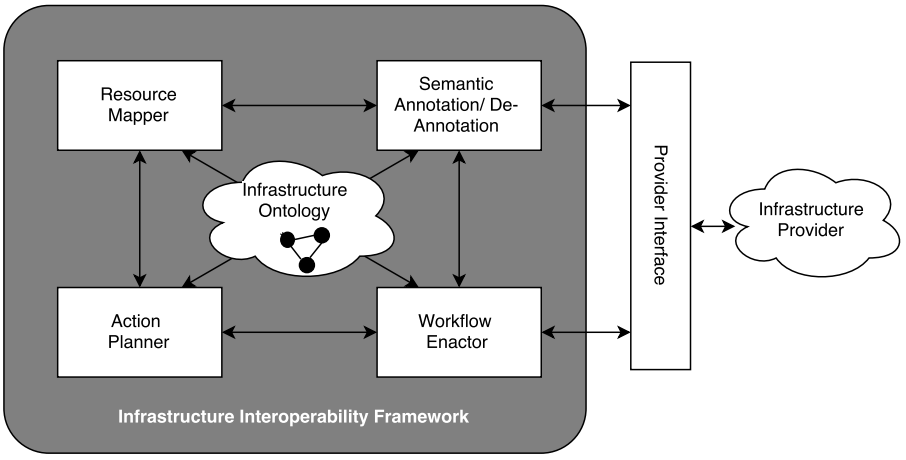


Figure 5.1 Infrastructure Interoperability Framework

5.1 Methodology

To achieve the aforementioned capabilities, we have designed an Infrastructure Interoperability Framework, depicted in Figure 5.1, which applies different Artificial Intelligence (AI) techniques to facilitate the interoperation with different infrastructure providers. This framework includes: the Infrastructure Provider Ontology, which provides a model to uniformly describe the resources and methods offered by the different providers' interfaces; the Semantic Annotation/De-annotation component, which is in charge of automatically create semantic descriptions from the providers interfaces description and messages (a.k.a. semantic annotation) and also create the interface messages from a semantic description (a.k.a semantic de-annotation); the Resource Mapper component, which converts the data descriptions between the different providers' models; and finally, the Action Planner component, which is in charge of inferring the required sequence of providers' actions in order to achieve the desired resource management task. The data translations performed by the Resource Mapper are performed by applying rule reasoning and the action inference performed by the Action Planner is achieved by applying AI planning techniques. Our solution allows any user or piece of software to exploit all available cloud infrastructures, even when they have different APIs defined, by using their preferred API. The translation from this API to the others is automatically done by our system.

The following sections describe in detail the different Infrastructure Interoperability Framework components. Afterwards, Section 5.5 provides examples of how the framework is applied in different situations, evaluates the introduced overhead and compares our framework with another interoperability solutions. Finally, Section 5.6 draws conclusions about this research topic.

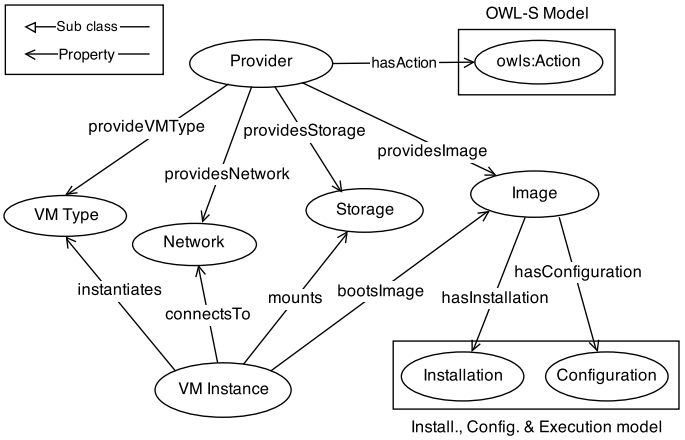


Figure 5.2 Infrastructure Providers Ontology

5.2 Infrastructure Providers Ontology

Figure 5.2 gives an overview of the Infrastructure Provider Ontology, which focuses on describing the computing resources offered by infrastructure providers (VM types, storage, network). In the literature, we can find several models for describing computing resources such as [117], [112],[34]. Therefore, our contribution has focused on extending these models for including the description of images and the providers' APIs. In the case of images, we propose to use the same resource state model used for the component installation and configuration. In current approaches, image descriptions are provided in plain text which are hard to be processed by machines. Our Infrastructure Ontology provides a model for describing the current status of an image which can be easily processed by computers in order to check if an image totally or partially contains the resources required by the application components, and infer what are the required changes in order to fulfill the resource status required by the component deployment and execution. On the other hand, we propose to model providers' API following OWL-S [118] concepts where different actions are described by indicating the input and output parameters and the prerequisites and effects on the infrastructure state as variables and predicates. Figure 5.3 shows the description of the Amazon EC2 *createInstance* action with the defined parameters, preconditions and effects.

5.2.1 Semantic Annotation/De-annotation

One of the key aspects of Cloud Computing is the exposition of providers' functionalities as web services. These services exchange messages to invoke methods which implement the requested functionalities in the provider's infrastructure. Unfortunately,

```
ec2:createInstance rdf:type owls:Action;
    owls:hasInput “?instanceType xml:String” ,
                  “?location xml:String”;
    owls:hasOutput “ ?instanceID xml:String” ;
    owls:hasPrecondition “ ”;
    owls:hasEffect “ ?vm rdf:Type infra:VMInstance” ,
                    “?vm instantiates ?instanceType” ,
                    “?vm location ?location”,
                    “?vm id ?instanceID”,
                    “?vm status created”;
```

Figure 5.3 API Action Description Example.

Table 5.1 Cloud Providers Protocols and Formats.

API	OCCI v1.1	Amazon	Flexiant	Rackspace	ElasticHosts
Protocol	HTTP/REST	HTTP/REST HTTP/SOAP	HTTP/SOAP	HTTP/REST	HTTP/REST
Format	HTTP Hdrs. XML	XML	XML	XML/JSON	JSON

the protocols and formats of these messages vary depending on the provider, so, the aim of the Semantic Annotation/De-annotation component is to solve the protocol and format heterogeneity unifying them into a machine understandable format such as RDF.

Table 5.1 shows the different protocols and formats used by several infrastructure provider's APIs. They mainly use SOAP and REST on top of the HTTP request and response messages. The SOAP protocol exchanges messages whose content include the method name and input parameters in the request messages and the output data in the response messages. On the other hand, REST is an architectural style for implementing Web Services which defines the set of possible actions (HTTP verbs) performed in a resource (URI). In this case, the method is determined by the HTTP verb and the URI path, the input data can be included in the URI as query parameters or in the request message content, and the output is returned in the response message. Another difference between REST and SOAP is the format. While SOAP only uses XML format, the content format in the REST implementations varies depending on the provider (Table 5.1). The main APIs use HTTP headers, XML or JSON and, in all cases, they are used to describe the cloud entities specifying their properties and values.

The RDF is also valid for this propose as it is based on a set of triples which define resource's property values. Moreover, this format also is able to bind resource and properties to semantic concept defined in an ontology. So, RDF graphs can be easily generated extracting these resources, properties and values from the selected API format applying some format translation method such as [119] for XML or other tools which converts other data formats to RDF [120].

In an initialization phase, the Annotation/De-annotation component generates the semantic descriptions for the providers' service descriptions to describe the different

types of actions and data. As explained in 5.2, the interface methods are modelled by using the OWL-S ontology. These methods are defined as subclasses of the *owls:simpleProcess* and the input and output datatypes are set in the range of the *owls:hasInput* and *owls:hasOutput* properties. These ontologies are automatically extracted from WSDL [121], WADL [122] or hREST [123] documents, parsing different key parameters for identifying the actions and datatypes. In the SOAP case, the operations and their input and output datatypes defined in the WSDL determine the provider's action type and input and output data ranges similarly to [124]. In the REST services, the HTTP verb and URI path (defined in the WADL or hREST) determine the provider's action type, and the query parameters, the content type and the schemas determine the input and output range.

During operation, the protocol messages are annotated to RDF graphs following the generated providers' ontologies. For achieving it, the Annotation/De-annotation component automatically captures the network message and parses the same key parameters as used in the service description for identifying the requested action type contained in the protocol message. Moreover, it also extracts the input and output data from the message content, getting the entities types, properties and values described in the providers' formats and annotates them in RDF graphs. Once a message is annotated, the extracted RDF graph contains two parts: one part describing the requested action and another part describing the input data. The data part will be processed by the Resource Mapper to translate it to the target model and the action part will be treated by the Action Planner to find the equivalent actions. The execution of these equivalent actions in the target provider generates response messages describing the output data which must be annotated and translated back to the source model following the same process that for the input data. Once the data has been translated, the A/D de-annotates the RDF graphs to the format expected by the provider. For doing this process, the action is translated to the corresponding XML tag (in the SOAP case) or the HTTP verb and URI path (in the REST case) and the input and output data are included in the message content with the corresponding provider format.

5.3 Resource Mapping

Once the source provider data has been unified in RDF graphs, they could be translated to other provider's concepts, defining equivalences between similar concepts. There are several options to express equivalences and mappings in the literature. One option is the Ontology Web Language (OWL) which contains primitives to define equivalences

Table 5.2 Cloud Providers Resource Data Mapping.

API	OCCI v1.1	Amazon	Flexiscale	Rackspace	ElasticHosts
Virtual Machine	Compute (Class)	Instance (Class)	Server (Class)	Server (Class)	Server (Class)
	compute.cores (int) compute.speed (int) compute.arch (int) compute.mem (int)	Instance.Type (string)	server_product_offer (int)	Server.flavor (string)	Server.smp (int) Server.cpu (int) Server.mem (int)
	StorageLink (Class)	Optional: blockDevice (Volume)	Optional: disk_product_offer (int)		Server.ide (string)
	NetworkInterface (Class)	subnetId (string)	vlan_id (string)	(public network)	Server.nic (string)
Storage	Disk (Class)	Volume (class)	Disk (Class)	(machine disk)	Drive
	compute.size (int)	size (int)	capacity (int)		size (int)
Network	Network (class)	Subnet(class)	VLAN(class)	(public net)	VLAN(class)
	address (string)	cidrblock (block)	ip_address(string)	-	-

```
Rule { ## Compute-to-Server
    (?resource rdf:type occi:Compute)
  =>
    (?resource rdf:type flexiscale:Server)
}
```

Figure 5.4 Example of rule to model a Class-to-Class equivalence

between classes (*owl:equivalentClass*) and properties (*owl:equivalentProperty*). However, it only allows one to one equivalences, so property to class equivalences can not be modeled with this approach. The RDTL [125] extends the one to one mappings with new mapping types, however it only treats mappings between ontology concepts (classes, properties and datatypes) but it does not consider the property values which are required for mapping different providers' concepts. In our proposal, the equivalences between providers' concepts are modeled by rules. A rule is composed by the body, which contains a set of conditions (premises), and the head which defines the inferred facts (conclusions) when the body conditions are fulfilled. So, a rule definition can be easily interpreted for defining equivalences and mappings. Classes, properties and values from the source model are easily expressed as body conditions,; meanwhile the equivalent classes, properties and values in the target model are mapped to head facts. Moreover, rules can contain built-ins which are useful for comparing and operating with the property values, increasing the number of possible mappings expressed with rules.

Table 5.2 shows the relationship between the main concepts defined in different providers' interfaces. Rows in this table group concepts which have similar meaning where a kind of equivalence or mapping can be defined to translate them. Based on these rows, different types of mappings have been identified. In Figure 5.4 , we enumerate them presenting examples about how users can express them as rules. The first type of mappings defines equivalences between classes, which model the same concept. (e.g. *Compute* in OCCI is equivalent to *Instance* in Amazon or *Server* in Flexiscale).

```
Rule { ## Compute_mem-to-Server_mem
    (?resource occi:mem ?value)
  =>
    (?resource elasticHosts:mem ?value)
}

Rule { ## Compute_properties-to-Server_product_offer
    (?resource occi:cores ?cores)
    (?resource occi:mem ?memory)
    lessEqual(?cores, 2),
    lessEqual(?memory, 512),
  =>
    (?resource flexiscale:server_product_offer 6)
}
```

Figure 5.5 Examples of rules to model a Property-to-Property equivalences

```
Rule { ## Server.ide-to-StorageLink:
    (?resource elasticHost:ide ?storage)
    makeNode(?link)
  =>
    (?link rdf:type occi:StorageLink)
    (?link occi:source ?resource)
    (?link occi:target ?storage)
}
```

Figure 5.6 Example of rule to model a Property-to-Class equivalence

Another type of mapping is related to property equivalences, where a value of one property is used to infer the value of another property (e.g. the *mem* property of *Compute* is equivalent to the *mem* property of *Server* in ElasticHosts), or where the value of a group of properties determine another property value (e.g. *Compute*'s properties values determine the *server_product_offer* value and vice versa). Examples of these type of equivalences can be found in Figure 5.5.

Finally, there are also mappings between classes and properties which define equivalences where classes are equivalent to properties and vice versa. Figure 5.6 shows an example of these equivalences where the *ide* property of *Server* in ElasticHosts is equivalent to a *StorageLink* class in OCCI.

Finally, the Resource Mapper uses a rule engine to apply data mappings and translate data to the target provider. This mapping process is depicted in Figure 5.7. The source data, provided by the A/D component in an RDF graph is loaded into an Inference Model which is created with the source and target data ontologies and the rule engine. Afterwards, the rule engine applies the translation rules over the input graph, which generates a deductions' graph. This graph contains the facts generated by the rules, which correspond to the data translated from the input graph. As a result of the data mapping process, the deductions' graph is serialized in an RDF file which can be also converted to the target provider's format applying the de-annotation process explained in Section 5.2.1.

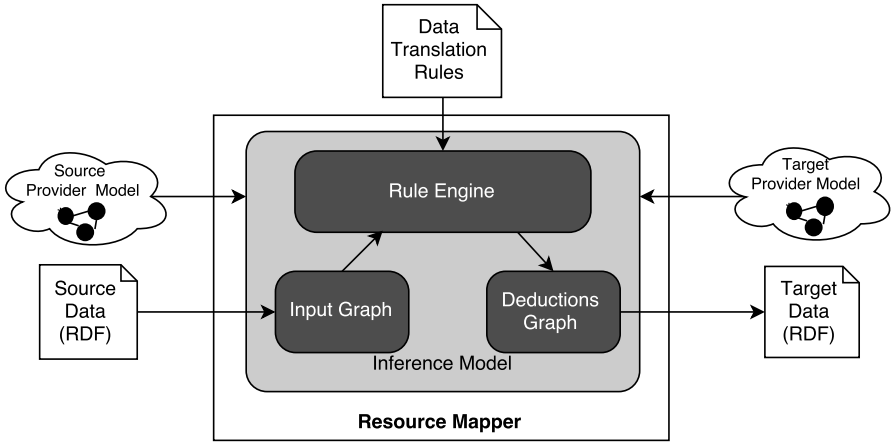


Figure 5.7 Resource Mapper Internal Design.

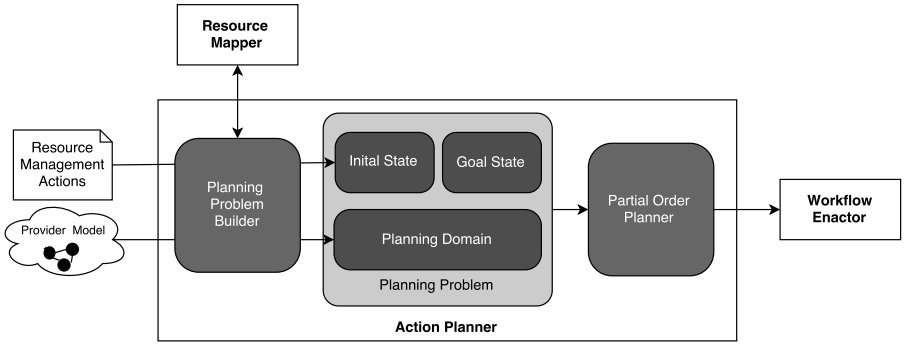


Figure 5.8 Action Planner Internal Design.

5.4 Action planning

The Action Planner is in charge of finding a workflow to perform the desired resource management action in a selected provider by applying AI planning. AI planners are systems which try to find a sequence of actions which are required to achieve a goal state from an initial state. An AI planning problem is defined by an initial and goal states and a planning domain which models how to describe the states and what are the possible state transitions. The planning problem is basically solved by applying an state-space search.

Figure 5.8 shows how the AI Planning is used in the Action Planner component. The management actions that users want to perform can be modeled as a planning problem whose initial state describes the current status of the resources to manage and the goal state describes what will be the desired status of the resources once the actions have been applied. The semantic description of the provider models how the resources are described and the possible actions of the provider interface. The information stored in the Provider

model matches with the information provided by a planning domain definition where the interface description could model the available as state transitions in the planning domain. So, with the management actions and the provider model, the Action Planner builds a planning problem. The generated planning problem is introduced to a Partial Order Planner [126] which performs the search and provides a sequence of partially ordered actions which produces the goal state from the initial state.

Finally, the Workflow Enactor executes the solution of the planning problem. This component will make use of the Semantic Annotation/De-annotation component which transforms the semantic description of the required interface methods to the real interface invocations.

5.5 Evaluation and Discussion

A working prototype of the described system has been implemented to validate the concepts presented in this chapter. The Infrastructure Provider Ontology has been described using OWL [100] and the Apache Jena [115] is used to describe data mapping rules and to perform the rule reasoning described in the Resource Mapper. For implementing the Action Planner, we have used Planning4J [127] to generate the planning domain and problem which is solved using the FF planner [128] which is an efficient implementation of a Partial Order Planner. The prototype has been deployed in an Intel i5 laptop with 8GB RAM and we have evaluated different aspects of it. First, we have applied the Infrastructure Interoperability Framework in two scenarios. In the first scenario, the framework is applied to infer the workflow for deploying the components of an application. In the second scenario, the Infrastructure Interoperability Framework is applied to translate interfaces between providers. In both cases, we evaluate the system overhead and its scalability. In the last part of the section, we have compared our solution with other interoperability solutions.

5.5.1 Inferring the Deployment Workflow

Following the overall storyline of the thesis, the Infrastructure Interoperability Framework can be used to transparently deploy the required resources to run the application components in the infrastructure providers. In this case, the framework is in charge of finding a workflow to provision the required resources (VMs, shared disks and networks), configuring the communication links and installing, configuring and running the components on the provisioned resources. To achieve it, the framework components are used in the way as depicted in Figure 5.9.

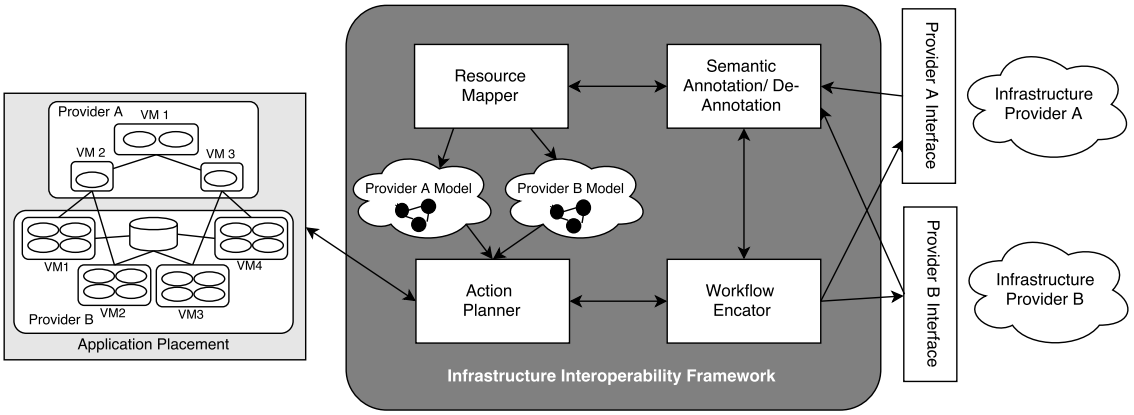


Figure 5.9 Usage of the Infrastructure Interoperability Framework for Inferring Deployment Workflow.

In an initialization phase, the different infrastructure providers interfaces and resources are described semantically with the Semantic Annotation/De-annotation and Resource Mapper components. Then, the application placement solution, obtained following the methodology explained in Chapters 3 and 4, can be seen as the desired infrastructure state and the providers' actions exposed in the interface define the state transitions in a planning domain. Therefore, we use the Action Planner to generate the AI planning problem whose initial state is empty state and the application placement is goal state. As explained in Section 5.4, it also generates the planning domain from the infrastructure provider model. Then, the generated problem is introduced to a Partial Order Planner which performs the search and provides a sequence of actions which produces the goal state from the initial state. For the installation, configuration and execution of components, we have defined a set of extra common action to apply changes on the image resource status. The Planner will inspect the resource status of the images assigned to each VM and compares it with the installation, configuration and execution description of the components assigned to these VMs. The required resources changes will be the input of this action, which will generating a manifest to automatically apply the VM configuration changes by using dev-ops systems like Puppet [98]. Finally, the Workflow Enactor executes the deployment workflow obtained from the Action Planner by invoking the required actions of the providers' API. This component uses the Semantic Annotation/De-annotation to semantically de-annotate the deployment workflow actions converting them to the real infrastructure provider calls.

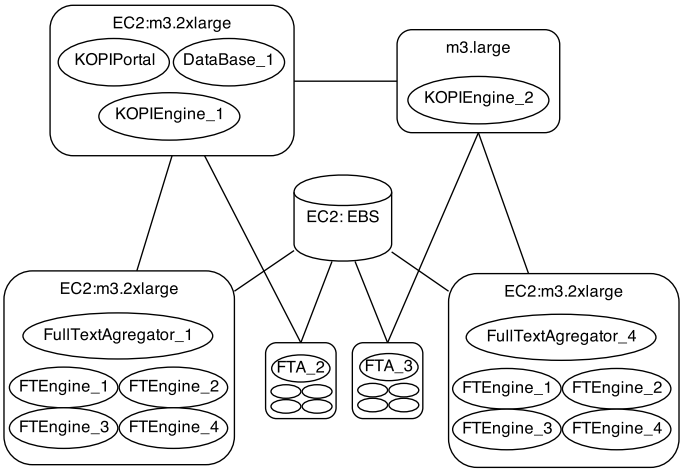


Figure 5.10 Placement solution for the KOPI Application in Amazon EC2.

Overhead and Scalability

To validate the system capabilities in this scenario, we have inferred the deployment workflow of the KOPI application, described in Chapter 3, to Amazon EC2 provider. The deployment placement solution obtained for the KOPI application in the EC2 provider is depicted in Figure 5.10, this placement result is passed to the Action Mapper of the Infrastructure Interoperability Framework which returns a plan for deploying the KOPI application in EC2. This plan is composed by a sequence of provider API calls to provision the computing resources, and a set of Puppet manifests for deploying the components in each VM. Figure 5.11 shows a snippet of the deployment workflow where a Disk volume and a VM are created and the disk is attached to this VM. Then, the VM is started and the IP address is obtained from its description. Finally, the installation configuration and execution manifest is applied.

For this example, the Action Planner has taken 65 milliseconds for getting the workflow which deploys the placement solution. To evaluate how the system performs for larger applications, we have generated synthetic application placement descriptions with different number of VMs to deploy. We have measured the time spent by the system to infer the deployment workflow for the different placements. The result of these measurements are depicted in Figure 5.12 where we can see how the inference time is related with the number of VMs. As we can see in the figure, the processing times grow polynomially with the number of VMs. In terms of time scale, inferring the deployment for large applications whose deployment requires hundreds of instances and VMs is obtained in several seconds. Despite, the time can be large for other problems; in the case of application deployment it is acceptable for the users because the deployment, booting, installation, configuration and execution of applications in the Cloud with a large

```
sequence{
    ...
    action{ ec2:createVolume
        input{ size = 50; location = us-east;};
        output{ volumeID = ?ID_2;};
    };
    action{ ec2:createInstance
        input{ type = m3.2xlarge; location = us-east;};
        output{ instanceID = ?ID_4;};
    }
    action{ec2:attachVolume
        input{volumeID = ?ID_2; instanceID= ?ID_4;};
    }
    action{ec2:startInstance
        input {instanceID= ?ID_4;};
    };
    action{ec2:describeInstance
        input {instanceID= ?ID_4;};
        output{ description= ?InstanceDescription_4};
    };
    (?InstanceDescription_4 ec2:ip ?ip)
    action{puppet:applyManifest
        input {node= ?ip ; manifest=node4.pp};
    };
    ...
}
```

Figure 5.11 KOPI Deployment Workflow Snippet.

number of VMs can take several minutes or even hours.

5.5.2 Interface Translation

The second scenario, where the Infrastructure Interoperability framework is applied, is the translation of interfaces. In this scenario, the calls performed in one provider API are transparently translated to another API. Figure 5.13 shows how the Infrastructure Interoperability Framework is used in this scenario. The Semantic Annotation/De-annotation component generates the semantic descriptions from the interface descriptions and the requested methods calls and vice versa. The Resource Mapper translates the data from one provider's model to another provider's model by applying the defined data mapping rules. The Action Planner infers the target provider's actions which are equivalent to the invoked provider call. Finally, the Workflow Enactor is in charge of executing the translated actions and getting the output data which will be translated back to the source model by using the Semantic Annotation/De-annotation and Resource Mapper components.

Figure 5.14 shows in detail how the framework components are invoked in the different phases. In the initialization phase (Figure 5.14.a), each interface description is semantically annotated creating a provider ontology which models the actions offered by each provider and the data model used to describe action data and resources. In addition to the descriptions, a set of Data Mapping Rules is provided to model the

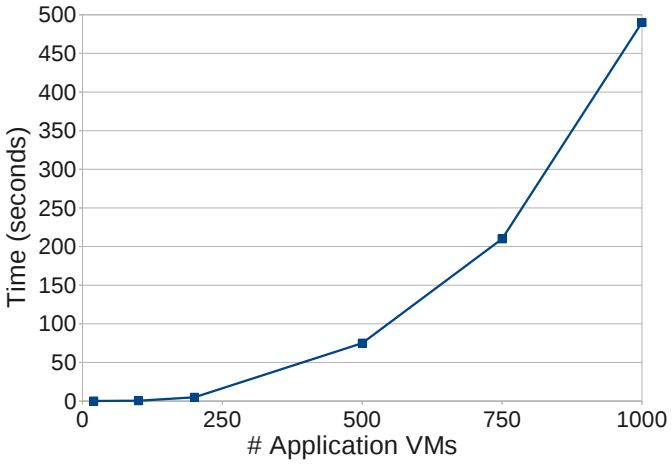


Figure 5.12 Overhead introduced by the Infrastructure Interoperability Framework for Inferring the Deployment Workflow depending on the number of VMs to deploy.

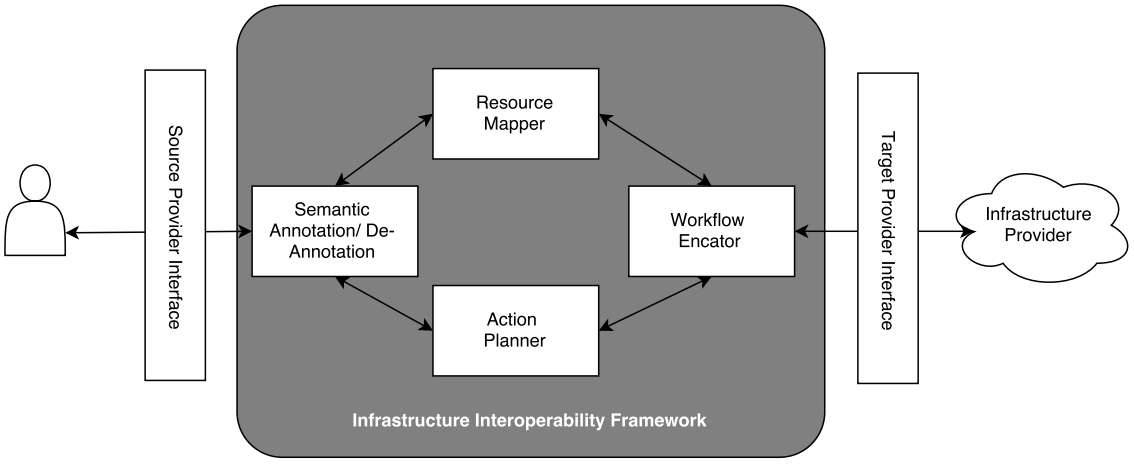
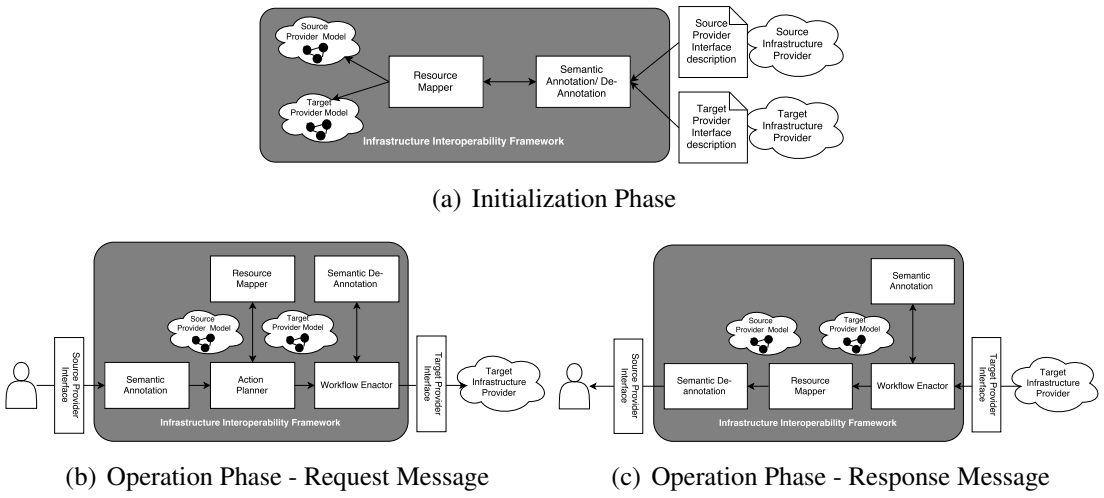


Figure 5.13 Usage of the Infrastructure Interoperability Framework for Interface Translation.



(a) Initialization Phase

(b) Operation Phase - Request Message

(c) Operation Phase - Response Message

Figure 5.14 Interoperability processes

equivalences between data descriptions from different provider's interfaces. During the operation phase, actions are requested using a source provider interface (Figure 5.14.b). These calls are captured and semantically annotated following the source provider model, extracting the requested actions and their input data. Then, the input data is sent to the Resource Mapper which automatically converts the input parameters to the corresponding descriptions in the target model by applying the data mapping rules. Then, the Action Planner creates a planning problem with whose initial state are the translate input and preconditions of the requested actions and the goal states are the translated description of the source action request. The state transitions are described by the available actions of the target provider model. The solution of the planning problem is the sequence of target provider's actions which has the same effect as the invoked source action. Once the requested action and the input parameters have been converted to the target model, they are sent to the Workflow Enactor, which extracts the actions to execute, transforming the converted parameters in the format required by the target provider using the Semantic Annotation/De-annotation component, and invokes the equivalent process using the supported protocol. Finally, the invocation of the workflow actions produce output data which must be translated into the source data model applying the inverse rules and de-annotating the data in the source format (Figure 5.14.c).

Overhead and Scalability

For validating and evaluating the Interoperability system, we have implemented translations between OCCI towards Amazon, FlexiScale or ElasticHosts. We have chosen

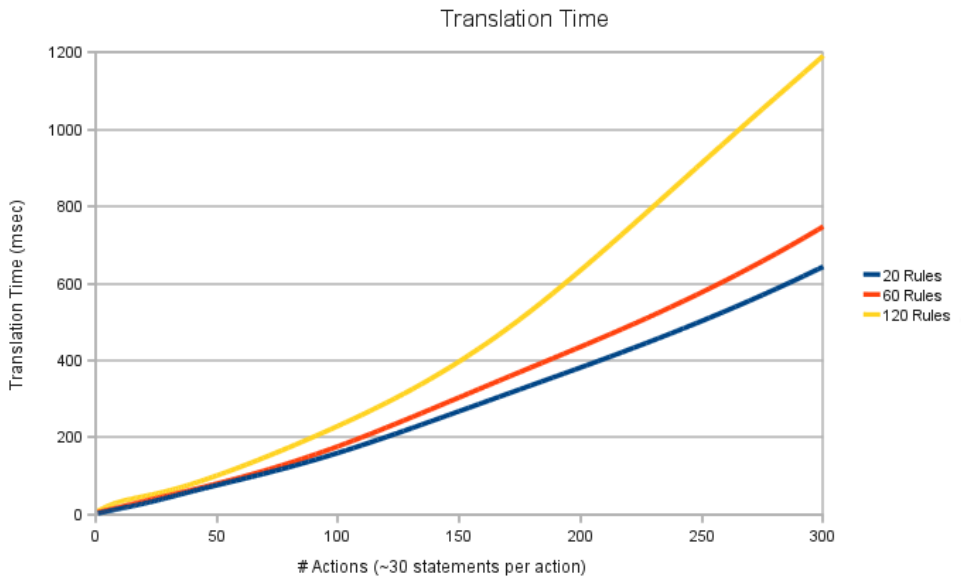


Figure 5.15 Performance Evaluation

these three commercial providers because they use different formats and contain different data and action mapping types. The mapping type complexity is closely related to the number of required rules for translating the APIs. So, few rules are required when APIs are very similar or when the rules used in one way can be used for the inverse translation. This is the case of the OCCI and ElasticHosts where *Compute-Server* classes are almost the same. On the other hand, mappings such as the OCCI's *Compute* to Flexiscale's *Server Product Offers* or Amazon's *InstanceTypes* require more rules because several instance types can be defined depending on the compute properties values.

Figure 5.15 shows the time spent for translating actions. It includes the time spent in translating input data with the translation rules, the inference of the equivalent actions, and the output data evaluating again the data translation rules. The translation time has been measured for different number of rules, which model different degrees of complexity; and the number of actions involved in a cloud application deployment. The *20 Rules* case models easy data translations; The *60 Rules* case models medium complexity translations. Finally, the *120 Rules* case models high complex translations. As we can see in the figure, the translation time grows faster with the translation complexity but almost linearly with the number of actions. When the complexity is low, the number of rules to evaluate is low and the equivalent actions are more or less one or two, so the time to infer this kind of translations is small. If we increase the translation complexity, the number of rules grows and the equivalent actions also grow. The reason why the growth with respect of

Table 5.3 Overhead Impact

OCCI	Interop. + FlexiScale	Flexiscale	Relative Overhead	Interop. + Amazon	Amazon	Relative Overhead
Post Compute	37488 ms	37439 ms	0.13%	2346 ms	2297 ms	2.13%
Get Compute	1030 ms	1078 ms	4.66%	279 ms	231 ms	20.78%
Delete Compute	1462 ms	1415 ms	3.32%	322 ms	275 ms	17.09%
Post Storage	23869 ms	23819 ms	0.44%	448 ms	398 ms	12.56%
Get Storage	1146 ms	1097 ms	4.47%	336 ms	287 ms	17.07%
Delete Storage	1001 ms	949 ms	5.48%	340 ms	289 ms	17.65%
Post Network	10958 ms	10910 ms	0.44%	313 ms	265 ms	18.11%
Get Network	1195 ms	1145 ms	4.37%	256 ms	206 ms	24.27%
Delete Network	1282 ms	1233 ms	3.97%	266 ms	217 ms	22.58%
Post StorageLink	1173 ms	1125 ms	4.27%	566 ms	508 ms	9.45%
Post NetworkInterface	2363 ms	2310 ms	2.29%	-	-	-
Shared Storage (Post Compute x2 + Storage + StorageLink x2)	86580 ms	86512 ms	0.08%	5418 ms	5350 ms	1.27%
Shared Network (Post Compute x2 + Net. + NetIf.x2)	75394 ms	75324 ms	0.09%	4594 ms	4524 ms	1.55%
Service Deployment (Post Compute x3 + Net. + Storage + NetIf.x3 + StorageLink x3)	154051 ms	153931 ms	0.08%	9198 ms	9078 ms	1.32%

the actions is almost lineal, it is because treating several actions is almost repeating the same procedure for the different actions. In this scenario, where we translate API calls, the planning problem is simpler than the previous scenario because equivalences can be found with very few actions, and therefore it also affects to the scale of the inference time, that is considerably smaller in this scenario.

To have an idea about how important the overhead is, we have compared the response times obtained from the invocation of different actions through the Infrastructure Interoperability Framework with the direct invocation of the translated actions. On one hand, different actions supported by the OCCI API have been submitted to the system which has been translated and executed at Flexiscale and Amazon. On the other hand, the translated actions have been directly invoked using the providers' APIs. The response time of these processes has been measured and compared, and the results are shown in Table 5.3. Observing the relative overhead (%), they are not very significant for creating resources, but it can be very important in actions for getting or deleting resources because the response time in these cases is very small. However, the absolute values (calculated as the difference between the direct executions and the interoperability columns) are more or less the same (50 milliseconds) for single actions, so the final user will not appreciate the difference between executing directly or by means of the system.

5.5.3 Comparison with other approaches

Several solutions for solving the vendor lock-in were introduced in Chapter 2. One of these solutions is defining a standard interface and one example of this solution is Open Cloud Computing Interface proposed by Open Grid Forum. The main drawback of this solution is that all the providers have to implement this interface for managing their resources. Despite small vendors be interested on developing this implementation to attract more users if the standard is broadly adopted, big vendors are reluctant to offer standard interfaces because vendor lock-in is beneficial for their interests due to they control the most important part of users. Another type of solutions is the plug-in/driver approaches such as jClouds, deltaCloud or DaseinCloud. They define an open API or interface and the interoperability with the infrastructure providers are done by means or plug-ins or drivers which implements the translation from the common interface to the specific provider's API. Our approach presents several benefits with respect of these proposals which are explained in the following paragraph.

Plug-in approaches define interfaces which must be implemented by programmers. The implementation of a plug-in requires: the knowledge of the plug-in interface (model, technology used, programming language, etc.), the identification of the data and action mappings between both APIs, the implementation of these mappings using the driver's programming model and language, and finally, invoking the interface calls fighting with protocols and formats used in the providers' interfaces. In our case, this complexity is reduced to only identify the mappings and express them as logic rules, without writing code and hiding all protocol and formats issues.

Rule languages provide an easier way to describe equivalences and mappings than implementing them with current programming languages. For implementing data mappings in plug-in approaches, the plug-in programmer has to parse the required properties and classes navigating through the input data and coding the generation of the output data. With rules, only the required source data and equivalent target data must be specified in the rule. The rule and planning engines will automatically detect when the required data is contained generating the equivalent data and the equivalent method calls.

Regarding aforementioned benefits with respect to plug-in approaches, we have compared the complexity of implementing a jClouds plug-in for EC2 with our Infrastructure Interoperability Framework. The available implementation of the compute plug-in for Amazon EC2 has 76 classes with bit more than 2200 lines of code. To develop an equivalent plug-in implementation with our framework, we have defined 84 rules for translating instance types and other data types defined in the EC2 interface used by the jClouds plug-in. The size of the rule definition is about 515 lines. Therefore, our

implementation just require around the 20% of the development and maintenance effort that a plug-in requires.

Moreover, plug-in approaches are statically bound to a specific source interface which is implemented by the different plug-ins and drivers. Our system is more flexible than these approaches, we do not force the user to learn a new interface and data model, we bring the opportunity to use the interface which he/she is more familiar.

Our approach is also easier to maintain, in current approaches a change in a provider's interface will require to update the code of the provider's plug-in. However, in our approach, there is no need to change any code; the user just needs to update mapping rules, if the interface changes affects to data descriptions. The rest is automatically updated by the system. The A/D component will update the semantic description of the providers' interface, and the Action Planner will automatically generate the action equivalences according to the updated descriptions in an on-demand way. So, next time we require to do a resource management action, the updated equivalences will be applied.

Moreover, our approach can support different interoperability scenarios changing only the defined mapping rules. We can support the most common scenario addressed by plug-in approaches, where a user want to deploy resources in different providers. In this case, the framework can be configured only defining mapping rules from the interface used by the user to the provider's ones. In addition to this, if we redefine the rules in an inverse way, the system can support the case where a provider want to support different APIs for facilitating the portability of clients which were using other providers without modifying their original API. Finally, if we charge both sets of rules, the system can also support a brokerage scenario making translations between different interfaces using an intermediate model.

Finally, despite we are using semantic technologies for doing the translation, most of these semantic issues (annotation/de-annotation, ontologies, etc.) are automatically performed and hidden to final users. They only have to use a small set of properties during the rule definition to refer to common properties such as the *rdf:type* for identifying the entity type or the *owls:hasInput* and *owls:hasOutput* for defining action input and output parameters.

5.6 Conclusion

In this chapter, we have proposed a framework for solving the vendor lock-in problem and facilitating the infrastructure provider interoperability. We have presented a methodology to automatically infer the set of provider's interface calls required to

perform any resource management tasks by applying different AI techniques. This framework includes a Semantic Annotation/De-annotation component which is in charge of generating semantic models from interface descriptions and calls and vice versa. Then, translations between the different models are performed by rule reasoning and AI planning. Users define data equivalences by means of rules which are applied by the Resource Mapper component, and the Action planner generates an AI planning problem from the requested resource management tasks and the provider's model. This problem defines the initial and goal resource states from the management tasks and the planning domain which models the state transitions from the available provider's interface actions. The result of this AI planning problem provides the sequence of actions in the provider's interface which performs the requested management action.

The proposed Infrastructure Interoperability Framework has been applied in two scenarios. The first scenario is about inferring the workflow of required provider's actions for deploying the resources for running the application components, and the second scenario is about translating interfaces between providers. In both scenarios, we evaluate the overhead introduced by the framework. For the first scenario, the most important part of the overhead is produced by the Action Planner to infer the deployment workflow. Despite it grows polynomially, it is still reasonable because the deployment of large applications can take several minutes or even hours. The overhead in the case of interface translation, is smaller an grows almost linearly because the problems solved by the Action Planner are simpler and applying the translation rules takes less time.

Finally, we have compared our solution with similar interoperability solutions which are based on open interfaces which interoperate with different infrastructure providers by means of plug-ins. Our approach reduces the complexity of plug-in implementation and maintenance by just defining and maintaining a set of rules for defining equivalences between data models. The rest of the translation is automatically performed by the system. Moreover, plug-in approaches are statically bound to a specific interface which is implemented by the different plug-ins and drivers. Our system is more flexible than these approaches, it does not force users to learn a new interface and data model and it brings the opportunity to use the interface which they are more familiar.

Chapter 6

Multi-agent Management

Previous chapters of the thesis have studied how to automate the different steps involved in a deployment of applications in distributed computing platform. We have seen their benefits and evaluated the overhead and the scalability. During this evaluation, we have seen that, in some scenarios, when the number of tasks and resources to evaluate is large, some issues appear which limit the scalability of the resource allocation process within a single node. Therefore, we require to find a mechanism to distribute the deployment across the different actors involved in the application deployment in order to solve the scalability issues. Besides, once the application deployment has finished and the application users can start to use it, the computational load of the application components can vary depending on the usage performed by these users. If a certain quality of service must be guaranteed, we require a mechanism to modify the number of resources assigned to serve the users request accordingly to the computational load variation. Finally, distributed applications are prone to suffer resource failures or a network outage can happen making part of the resources inaccessible, etc. To reduce the application downtime, we require to have some self-healing features which enable the application to recover from these failures. This chapter aims to develop a system to improve the scalability issues and provide self-adaptation features to control the application execution. To achieve it, we propose a multi-agent autonomous management system which coordinates the different steps of the deployment processes in an scalable way and monitors the application execution detecting and solving failures as well as adapting the number of resources to the computational load. The rest of the chapter is organized as follows: first, Section 6.1 presents the overall solution and Sections 6.2, 6.3, 6.4 provide more details about the features of the implemented software agents. Then, Section 6.5 provides the evaluation results and Section 6.6 compares our solution with other approaches. Finally, Section 6.7 draws the conclusions about the topic.

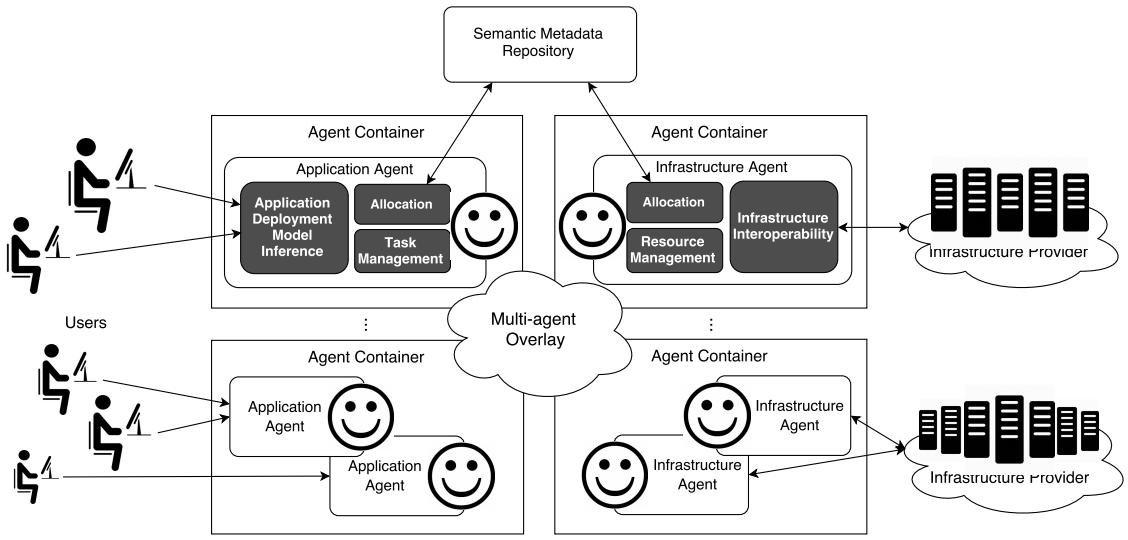


Figure 6.1 Multi-Agent Management Architecture

6.1 Methodology

Multi-agent technologies [7] are used to increase the autonomy and self-management of a system. Agents are proactive, so they can take decisions by themselves according to their goals and trigger actions by their own initiative. For these reasons, agents are suitable for coordinating the deployment and execution of applications in distributed computing environments. Agents can adapt their behavior depending on how the execution of the application execution is evolving, detecting execution problems and triggering the most appropriate actions for reacting to them depending on the system status and the resource capabilities. Agents are also capable to communicate to each other. They implement negotiation protocols which are very useful to reach agreements and cooperate with other agents. It is also very useful for distributing the work and responsibility of deploying applications between a group of software agents, keeping the benefits of the semantics proposed in the former chapters, but adapting the process to a Multi-Agent Resource Allocation (MARA) approach [129].

Figure 6.1 shows the architecture of the distributed management system across multiple agents. Users' applications and provider's resources are represented in the system by software agents which are in charge of performing application and resource management respectively. Moreover, they also share the functionality of semantic resource allocation which has been split into two parts. On one hand, the Application Agent (AA) is in charge of requesting proposals for allocating applications and selecting the best one for the customer's interest. On the other hand, the Infrastructure Agent (IA)

Status	Active Goals	Plan
Submitted	Get Resources	Infer Deployment Model Negotiate Allocation
Scheduled	Find Better Scheduling	Monitors Application Ends Negotiate Allocation
Running	Monitor Execution Recover Degradation	Check Performance Update Deployment Model Increase/decrease Resources
Stopped	Recover Stopped	Negotiate Allocation
No Scheduled	Recover Non Scheduled	Negotiate Allocation
Canceled	Cancel Job	Notify Cancellation to resources

Table 6.1 Application Agent Plans

is in charge of making allocation proposals for applications in its resources according to the provider's interest.

In this approach, agents have been implemented using a Belief-Desire-Intention (BDI) model [130]. For each type of agent, a set of data (Beliefs), goals (Desires) and plans (Intentions) are defined to model the behavior of the agent. At runtime, the agents BDI engine monitors the values of the data. According to these values, it activates the goals to achieve and execute plans with the actions to reach these goals. The following paragraphs describe in detail the Application and Infrastructure Agent functionalities and how the BDI model is applied for implementing them.

6.2 Application Agent

The main beliefs for an Application Agent are the Application descriptions. They have been described following the Application Deployment Ontology presented in Section 3.2 which includes: the different resource requirements, the time constraints and the Quality Rules of the application components and links; the application components' execution status; and other customer's properties such as level of preference, customer's reputation, etc. which can be important for solving conflicts during the application execution. The main goal of an Application Agent is managing the execution of the application components in the resources of the system. This execution has different states. Some of them indicate that the execution is running in a normal way, while others indicate that the execution could not be finalized. Depending on the state, the Application Agent has to act in a different way to reach its main goal.

Table 6.1 lists subgoals and plans for implementing the Application Agent features. When an application reaches one of the status listed on the table, the Application Agent activates one or several subgoals and triggers the execution of plans to achieve the subgoal. These plans model how the agent behaves for each situation. For instance, when an application is submitted for deployment, it starts with a *Submitted* state, the Application

Agent actives the *Get Resources* goal which triggers the execution of *Infer Deployment Model* Plan which perform the ontology and rule reasoning over the semantic application description as described in Chapter 3. The inference result modifies the application description indicating how many instances of the application components must be deployed and their affinity. Once this plan has finished, the application agent executes the *Negotiate Allocation* plan which starts a negotiation with the Infrastructure Agents in order to find the most appropriate resources to execute the application component instances. This negotiation is explained in detail in Section 6.4. A successful negotiation changes the status to *scheduled* and the Application Agent will activate new goals and plans.

Once the application is *running*, the Application Agent activates the goal for monitoring the execution. This goal periodically triggers the mechanism for evaluating if the required performance is fulfilled. It is performed by monitoring the load metrics defined in the application description and evaluating the Quality Rules with the collected values of the load metric and the quality level defined for the application. The evaluation of these rules will provide the required number of resources per component. If it is differs from the actual values, the Application Agent activates the *Recover Degradation* goal which triggers the appropriate plans for adapting the resources to the current load by deploying or removing resources.

The Application Agent also applies fault-tolerance techniques to recover itself from status which can alter the normal execution of the application (*stopped, no scheduled*). In the situations where a running application component has been *stopped*, it will execute plans for redeploying the component. An application can also reach a *No Scheduled* state because the allocated resource has rejected the application component. Then, the Application Agent tries to recover from this status negotiating a new allocation for this component. If the plans for recovering from status *stopped*, and *no scheduled* do not achieve the recovery goal, the plan will be executed again when a system event indicates that some resources have been released (new resource added or an application execution has finished).

6.3 Infrastructure Agent

The main goal of an Infrastructure Agent is the management of resource capabilities for executing the applications requested by the customer. For achieving it, a set of beliefs have been defined: the resources descriptions, where the properties and capabilities of the resources controlled by the agent are described; the allocation of the application

Beliefs	Active Goals	Plans
Schedule Application Components	Monitor Scheduled Applications Components	Evaluate Scheduled Application Components Perform Application Component Execution
Running Application Components	Monitor Running Application Components	Evaluate Running Application Components Treat Status change
Failed	Recover Resource Failure	Try Local Reschedule Notify Application Agents
Running	Register Resource	Annotate resource description

Table 6.2 Infrastructure Agent Plans

components assigned to the resources controlled by the agent; and the status of each resource. These beliefs are described following the Resource Allocation Ontology described in Section 4.2. In Addition to the beliefs, a set of subgoals and plans have been defined to model the Infrastructure Agent behavior. They are shown in Table 6.2.

Infrastructure Agents contain two subgoals for monitoring the scheduled and running application components assigned to their resources. They are periodically activated while the resources have *scheduled* or *running* components. Every time the subgoals are activated, the Infrastructure Agent executes a plan for checking the *scheduled* or *running* application components. The plan executed for the scheduled components (*Evaluate Scheduled Application Components*) checks if the start time has been reached and, if that is the case, the agent proceeds with the deployment and execution of the component at the selected resource. In the case of running components, the plan (*Evaluate Running Application Components*) checks their status and, if a deadline has been defined for the component, the agent checks if it has been reached. If it is the case, the Infrastructure Agent cancels the component execution and notifies the cancellation message to the corresponding Application Agent.

Additionally, the Infrastructure Agent is also prepared to react to resource failures. The plan for recovering the failure communicates the failures sending a *stopped* notification message to agents whose components were scheduled at the failed resources. The Application Agents treat these notifications according to the plans explained in Section 6.2. Once the resource is *running*, the Infrastructure Agent annotates the resource description and registers it in the SMR for enabling the execution of applications in the resource. Semantic Resource Annotation and the actions for deploying and executing the application components are performed by using the Interoperability Framework described in Chapter 5.

Finally, the Infrastructure Agent contains plans for responding to the call for scheduling proposal in order to collaborate in the resource allocation decisions. This distributed allocation process is explained in detail in Section 6.4.

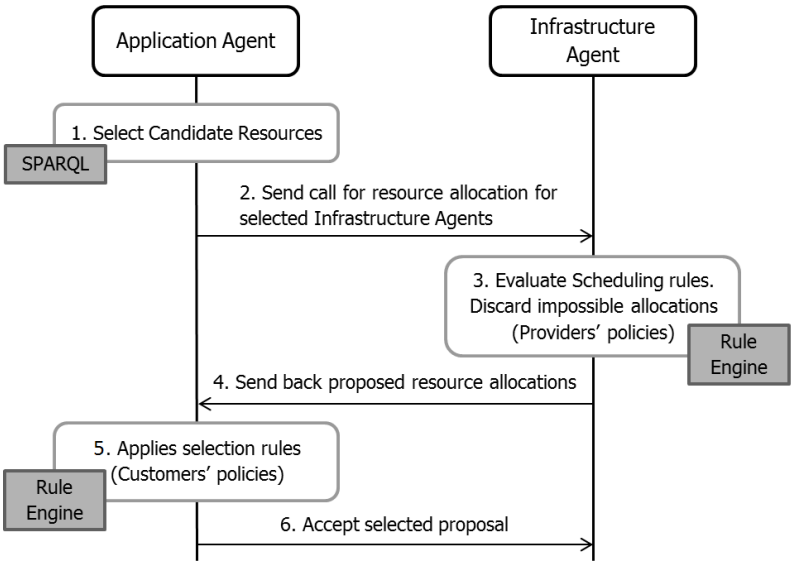


Figure 6.2 Distributed Allocation negotiation protocol

6.4 Distributed Semantic Resource Allocation

Chapter 4 has described a centralized semantic approach for allocating computational tasks on resources. This centralized approach could have scalability issues when the number of tasks and resources is large. Taking profit of the sociability features of software agents, we have distributed resource allocation functionality across the Application and Infrastructure Agents. When a resource allocation for a particular application component or task is requested, a negotiation between the Application Agent and a set of Infrastructure Agents is initiated by using the Contract Net Protocol (CNP) [131]. Figure 6.2 shows the message exchange between these agents for allocating the computational tasks on the resources. When an Application Agent has activated a goal for allocating resources (*Get resources*), it builds a query for selecting candidate resources based on the application component description (1). This query will return the providers which contains resources matching with the component requirements. Once a set of Infrastructure Agents has been selected, the Application Agent initiates a negotiation sending them a call for allocation proposals (2). Each Infrastructure Agent makes its own proposal according to their load and interests and returns it to the Application Agent (3). Finally, the Application Agent evaluates all the received proposals (4) selecting the best one for its interests and rejecting the others (5).

Infrastructure Agent proposals and Application Agent evaluations are done using the Semantic Scheduler module of the Semantic Resource Allocation presented in Chapter 4. In the Infrastructure proposal, the Rule Engine of the module is populated with the

```
Rule { ## Select the allocation with earliest deadline
    ?alloc1 rdf:type rao:Allocation
    ?alloc1 rao:uses ?resource
    ?alloc1 rao:allocates ?task
    ?alloc1 rao:plannedEndTime ?endTime1
    ?alloc2 rdf:type rao:Allocation
    ?alloc2 rao:uses ?resource
    ?alloc2 rao:allocates ?task
    ?alloc2 rao:plannedEndTime ?endTime2
    ?task rao:status rao:Submitted
    lessThan(?endTime1, ?endTime2)
=>
    drop(?alloc2)
}

Rule { ## Select the resource with allocations first
    ?alloc1 rdf:type rao:Allocation
    ?alloc1 rao:uses ?resource1
    ?alloc1 rao:allocates ?task
    ?resource1 rao:usedBy ?alloc)
    ?alloc2 rdf:type rao:Allocation
    ?alloc2 rao:uses ?resource2
    ?alloc2 rao:allocates ?task
    noValue(?resource2 rao:usedBy)
    ?task rao:status rao:Submitted
=>
    drop(?alloc2)
}
```

Figure 6.3 Example of provider's policy rules to generate allocation proposals.

possible allocation based on provider's resource descriptions and components already scheduled on them. This module will apply the rules for generating the Allocation proposals according to the current load, eliminating the allocations which do not fulfill the provider's interest. Figure 6.3 shows examples of rules for evaluating time constraints such as a deadline or a green provider's policy for allocating first resources already in use in order to save energy switching off empty resources.

In the case of the Application Agent proposal selection, the rule engine module is populated with the application description and the allocation proposals received from the selected Infrastructure Agents and apply the rules for selecting the best allocation according to customer's interest, eliminating the others. Figure 6.4 shows examples of these rules. An earliest start date policy can be expressed as described in the first rule of the figure. In a similar way, a rule for selecting the best proposal according to the provider's preference can be defined as shown in the second rule of the figure.

6.5 Overhead and Scalability Evaluation

The multi-agent management system is implemented using different existing Java frameworks. The system agents have been developed with the Jadex framework [132] which offers an easy interface for implementing agent beliefs, goals and plans. It is

```
Rule { ## Select the allocation with earliest start time
    ?alloc1 rdf:type rao:Allocation
    ?alloc1 rao:uses ?resource
    ?alloc1 rao:allocates ?task
    ?alloc1 rao:plannedStartTime ?startTime1
    ?alloc2 rdf:type rao:Allocation
    ?alloc2 rao:uses ?resource
    ?alloc2 rao:allocates ?task
    ?alloc2 rao:plannedStartTime ?startTime2
    ?task rao:status rao:Submitted
    lessThan(?startTime1, ?startTime2)
=>
    drop(?alloc2)
}

Rule { ## Select the resource with preferable provider
    ?alloc1 rdf:type rao:Allocation
    ?alloc1 rao:uses ?resource1
    ?resource1 rao:isProvidedBy biz:ResourceProviderA
    ?alloc1 rao:incarnates ?abstResource
    ?alloc1 rao:allocates ?task
    ?alloc2 rdf:type rao:Allocation
    ?alloc2 rao:uses ?resource2
    ?resource2 rao:isProvidedBy biz:ResourceProviderB
    ?alloc1 rao:incarnates ?abstResource
    ?alloc2 rao:allocates ?task
    ?task rao:status rao:Submitted
=>
    drop(?alloc2)
}
```

Figure 6.4 Example of customer's policy rules for selection the best allocation proposals.

deployed on top of the JADE agent platform [133] which allows us to build a multi-agent platform distributed across several computing nodes. Semantic annotations, queries and reasoning components are implemented with the Jena 2 Semantic Web framework [115]. It provides an API for managing RDF files and SPARQL queries as well as a semantic reasoner with rule engine support.

In a first qualitative evaluation of the architecture, we can see the following benefits: agents and the BDI engine facilitates the execution of complex tasks such as the resource and application management and the reaction to unpredictable events such as failures. With BDI agents, this kind of problems can be solved in a high level view instead of entering in detailed mechanisms for implementing a specific solution. Agents also implement negotiation and coordination protocols which allow the distributed resource allocation to integrate the customer's and infrastructure providers' interests in the same scheduling mechanism. Moreover, the different agents can be deployed across multiple locations deploying agent containers on different machines. Each container implements a messaging system which allows the communication between agents located on different containers. The distributed configuration of the agent platform can improve the scalability of the Semantic Resource Allocation because the different parts can be processed in parallel on agents deployed in multiple hosts.

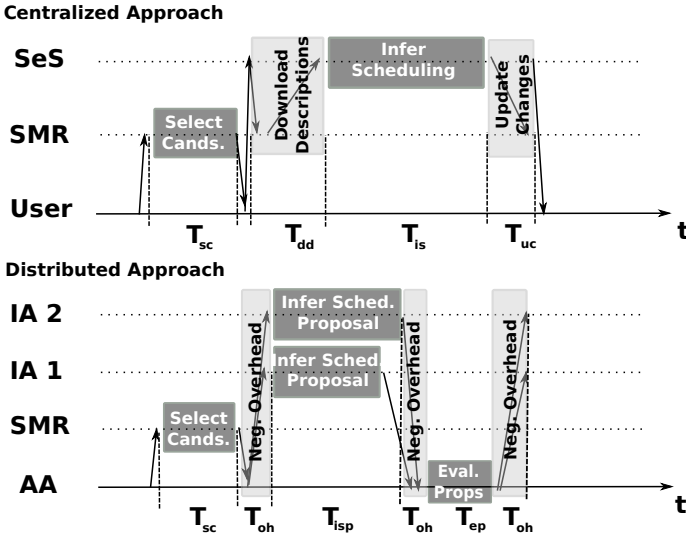
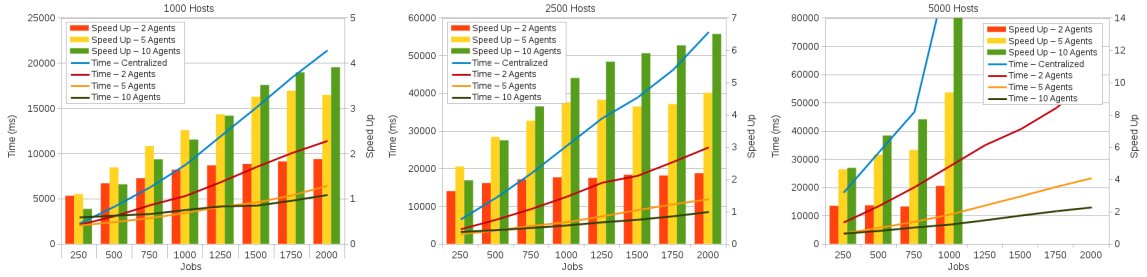


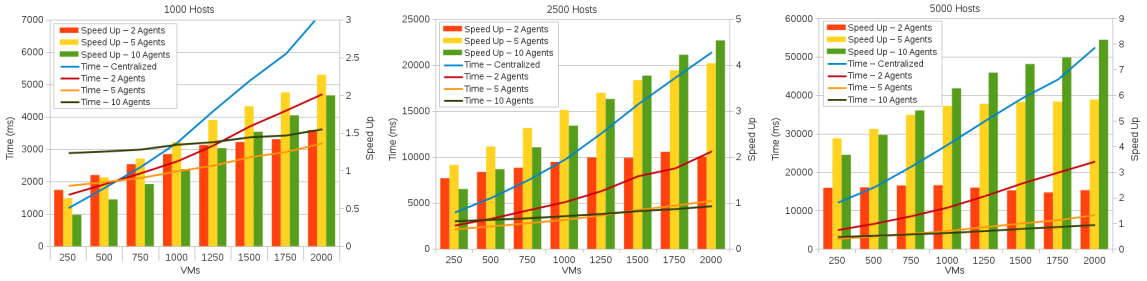
Figure 6.5 Centralized vs. Distributed processes comparison

Finally, we have performed a quantitative evaluation of the overhead and performance of the distributed approach comparing them with the centralized approach. Figure 6.5 compares both resource allocation approaches. The first part in both approaches is almost the same, since queries to select the candidate resources or the candidate resource agents need to evaluate the same resource requirements. So, there is no difference in this overhead. The time for getting the candidate resources depends only on the number of resources, and it is low compared to the other processes. In the centralized approach, the Semantic Scheduler (SeS) required the SMR for getting candidate resources and allocated computational tasks, as well as for updating the allocation results. In the distributed approach, the SMR is only needed to advertise the resources because Infrastructure Agents only evaluate their local resources and tasks assigned to them, and these descriptions are stored locally. In contrast, a negotiation overhead has been introduced. So, the time for negotiating an allocation between Application and Infrastructure Agents is $max(T_{isp}) + T_{ep} + T_{oh}$, meanwhile the allocation time with the centralized approach is $T_{dd} + T_{is} + T_{uc}$. We have evaluated the system in the same scenarios where we evaluated the centralized approach in Chapter 4: the Grid Scheduling, Private Cloud and Multi-cloud. Figure 6.6 shows the comparison between the total allocation times in the different scenarios using the centralized approach and the multi-agent approach distributed across different number of agents.

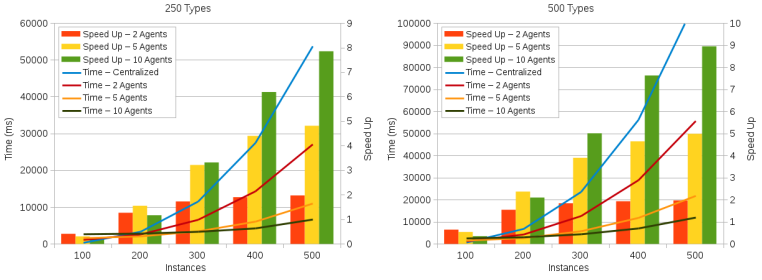
The most important issue we can see in Figure 6.6 is the trade-off between the granularity of the distribution and the time for getting the resource allocation. If the number of resources and tasks is small, the distributed approach performs worse than the



(a) Allocation time comparison for Grid Scheduling scenario



(b) Allocation time comparison for Private Cloud scenario



(c) Allocation time comparison for Multi-Cloud scenario

Figure 6.6 Centralized vs. Distributed allocation times comparison

Agents	1	3	5	7	9
Time(msec)	804	1192	1585	1996	2510

Table 6.3 Negotiation Overhead

centralized one. This is due to the behavior of the different times in the different situations. With small number of resources in the system, T_{dd}, T_{is} and T_{uc} are small, meanwhile, the negotiation overhead (T_{oh}), which is produced by the message exchanges and the BDI engine, is very important (see Table 6.3). On the other case (big number of resources), T_{dd}, T_{is} and T_{uc} grows faster with the number of resources and tasks than T_{isp}, T_{oh} and T_{ep}, because the inference work in T_{isp} is distributed across the Infrastructure Agents and T_{oh} and T_{ep} keeps almost invariant with the increment of resources because they depend on the number of agents.

The time for inferring the allocation proposal with multi-agents (T_{isp}) has a similar behavior than the time for inferring the allocation in the centralized approach (T_{is}). Infrastructure Agents make the same inference than the Semantic Scheduler but only with the local data instead of the global number of resources and tasks. The inference of the scheduling proposals is done in parallel on the selected Infrastructure Agents. The time to perform it varies on each resource depending on the number of resources controlled by the agent and tasks scheduled on these resources. The best performance of this part is found when all Infrastructure Agents control the same number of resources and the resource allocation is well balanced. In this situation, the selected Infrastructure Agents evaluate the maximum number of resources and tasks with the minimum cost in time.

Finally, we have evaluated the effect of deploying several agents on the same machine. Figure 6.7 shows the allocation time deploying different agents to the same node. The allocation time is similar if the number of agents per host is small. The gain in exchanging messages is more or less compensated by the share of resources during the parallel computation in the same node. However, when we deploy several agents in the same node the allocation time starts to grow considerably. It is caused by implementation details of the agent platform. Each agent is implemented as a Java thread, so when the number of threads is bigger than the number of cores of the machine (8 cores in our case), the allocation inference time is degraded because several agents have to share the processing and memory resources which starts to be more important than the network overhead. As a consequence, deploying too many Infrastructure Agents on the same host produces performance degradation in the allocation process.

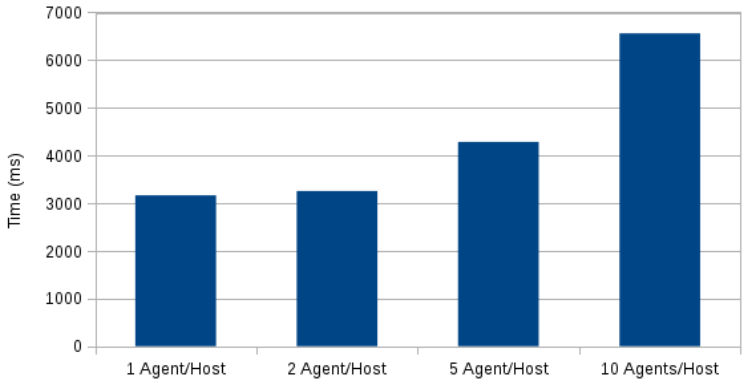


Figure 6.7 Deployment configuration vs allocation time

6.6 Comparison with other approaches

The idea of using software agents in the management of distributed platforms was introduced when in [82] where Foster et al describe the benefit of integrating the results in both research areas. Software agents could improve the autonomy, flexibility and scalability of current distributed platforms in different areas, however multi-agent system researchers have mainly focused on the area of resource allocation and job scheduling. Regarding this area, several solutions have been proposed, such as the ones based on market-control, where each agent tries to maximize its benefit function and the market controls them,; the ones based on social wellfare, where the multi-agent system tries to maximize a collective benefit,; and other ones based on game theory. There is a lot of literature about market based allocation solutions. We would like to highlight proposals like Challenger [83], Tycoon [84], other studies more focused on Grids such as TRACE [87] or ARAM [88] and the projects CatNets [89] and Sorma [91]. The works on wellfare engineering and game theory for multi-agents resource allocation has been compiled in [85] and [86].

Our work does not aim to implement or improve the existing algorithm to solve the multi-agent allocation problem, each of the mentioned solutions have their own advantages and drawbacks. However, they have been tailored for specific problems and systems, so the integration and adaptation to new computing environments is complex. To make this adaptation, developers and system administrator have to adapt descriptions of tasks and resources to the schemes supported by these systems. In contrast, our solution combines the multi-agent resource allocation with the benefits of semantics on interoperability, facilitating the integration of applications and infrastructure resources

coming from different customers and providers. We have introduced the multi-agent solution in the Semantic Resource Allocation leaving users the possibility of extending or changing the policies. In our system, customers and providers can describe the most convenient scheduling rules for their interests. Those policies will be combined during the negotiation, trying to get a solution to satisfy all of them. Moreover, Infrastructure agents are integrated with different Infrastructure Interoperability Framework which allow us to semantically annotate the different provider's resources and interfaces.

Finally, our multi-agent system also integrates fault-tolerance and adaptation capabilities. Our solution was the first solution to introduce multi-agent systems in the semantic resource allocation and to use software agents for managing the application monitoring, adaptation and fault tolerance in an integrated way. Currently, other cloud middleware also include monitoring, fault tolerance and adaptation services. However, these solutions are based on central services which must be configured by the user using the provider's interface. In contrast, the software agents of our multi-agent approach already have integrated the self-adaptation and fault-tolerance features, so they are inherently distributed across the agents and do not require extra configuration steps.

6.7 Conclusion

In this chapter, we have studied how to introduce the benefits of software agents in the application management in distributed platforms. We have designed and implemented a multi-agent system which is in charge of coordinating the different steps of the application deployment introduced on the previous chapters in a distributed way as well as monitoring the correct execution of the application in the computing resources. Two types of agents have been defined: Application Agents, which are in charge of the application management; and Infrastructure Agents, which are in charge of the resource management. Moreover, these agents also share the functionality of semantic resource allocation described in Chapter 4 which has been split into two parts. On one hand, the Application Agent is in charge of requesting proposals for allocating applications and selecting the best one for the customer's interest. On the other hand, the Infrastructure Agent is in charge of making allocation proposals for applications at its resources according to current resource load and the provider's interest.

When the application is submitted to the system, an Application Agent starts to manage it taking different actions according to the application status. The application starts in a *submitted* status, at this state, the Application Agent performs the deployment model inference as presented Chapter 3 and negotiates an allocation for these deployment

model with the other Infrastructure Agents. After a successful allocation negotiation, the Infrastructure Agent of the assigned resource is in charge of deploying and running the application components by means of the Infrastructure Interoperability Framework described in Chapter 5. Once the application is running, the software agents monitor the application execution reacting to unexpected events. From one hand, the Application Agent is in charge of monitoring the application load and recalculating the required components and resources and adapting the deployed application according to the new requirements. On the other hand, the Infrastructure Agent is in charge of monitoring their resources and in case that a failure happens notify the affected agents in order to recover the application from failures.

We have implemented a prototype of this system, and we have measured the overhead in different situations. We have compared the distributed semantic resource allocation approach with the centralized approach described in Chapter 4. We have identified the most important processes and overheads in both approaches. As a result of this evaluation, we have detected the centralized approach performs better when the number of resources and tasks is low due to an important negotiation and agent engine overhead while the distributed approach is a better option when the number of resources is big.

Comparing our solution with other approaches, our solution was the first one to integrate different features in the same management system. First, our system integrated the semantic technologies in a distributed resource allocation process, facilitating the integration with applications and resources coming from different sources as well as integrating the customer's and provider's policies in the same allocation processes. Second, we have taken profit from software agents to integrate adaptation and fault-tolerance capabilities. Current middleware solutions has recently added fault tolerance or adaptation services. These solutions are based on central services which must be configured by the user. In contrast, the software agents of our multi-agent approach already have integrated the self-adaptation and fault-tolerance features, so they are inherently distributed across the agents and do not require extra configuration steps.

Chapter 7

Conclusions

The work presented in this thesis has focused on providing a platform that facilitates and automates the integration of any kind of applications in different providers' infrastructures lowering the barrier of adopting new distributed computing infrastructures such as Clouds. The achievement of this overall objective has been split in several parts which have been treated in the different chapters of the thesis.

In the first part, we have studied how to describe applications and how to automatically infer a model for deploying any kind of application in a distributed platform. It has been performed taking benefit from semantic web technologies. First, we have presented an ontology which provides a general-purpose and infrastructure-agnostic model for describing distributed applications. Following this model, the application description is composed by three parts: the component topology, which provides the definition of the application components and their communication links; the installation, configuration and execution, which describes the desired state of the application resources after its installation, configuration and during the execution; and the quality description, which provides a set of rules to infer the processing requirements and component replicas for a given quality and application load. Once the application is described, it is loaded to a rule reasoner which applies different rules to classify the components and communication links and the quality rules to extract the component and link requirements. Based on this classification and the extracted requirements, the reasoner also infers the implicit affinity constraints. A prototype of this approach has been implemented, validated and evaluated. To validate the model, we have described four applications from four types: n-tier, task-based, map-reduce and MPI/OpenMP. We have evaluated the reasoning overhead by measuring the time to infer the deployment model in different scenarios. We have observed the overhead is growing linearly with the number of components and, even for large applications, the overhead is just few seconds. This overhead can be considered

low compared with the time required to deploy VMs on the Cloud Infrastructures, and much more faster compared with the time spent by a developer to do the same inference. Finally, our proposal has been compared with other cloud application models (OVF, TOSCA, CloudML and mOSAIC). The main advantage of our approach is that developers do not have to specify how the application is distributed in VMs because the application model reasoner infers it. In our approach, the application code remains unchanged, because the model is describing the required installation, configuration and execution procedure, instead of forcing developers to implement management interfaces, as in TOSCA, or implementing components as *Cloudlets*, as in mOSAIC. Regarding the application quality, other approaches indirectly specify the required application quality by statically setting the processing requirements of the VM. In our case, the processing requirements are described by rules, which provides the component requirements as function of the quality and load metrics.

Once the application deployment model has been inferred, the second step is finding the resources to deploy and execute the different application components. This step has been studied in Chapter 4. In this chapter, we have studied how semantic web technologies can be applied in the problem of allocating application components in computing resources which is the second part for achieving an automatic application deployment. In this case, we have defined a Resource Allocation Ontology, which models the concepts involved on the assignment of resources to the computing tasks. Allocation policies are modeled as horn rules which are applied over a knowledge base composed by the available resources and the already allocated task descriptions. The evaluation of the proposed solution has been focused in two parts. First, to validate that the solution can be easily used in different scenarios, we have applied it in the scheduling of jobs in Grid environments, the allocation of VM deployments in a Private Cloud environment, and the allocation of application component instances in the different VM types in a multi-cloud environment. Second, we have measured the overhead introduced by the system in the different scenarios and how it grows with respect of the number of tasks and resources. The overhead is directly related to the complexity of the allocation but in all the cases it is assumable when we compare to other expected overhead or task durations such as queue waiting times and deployments. However, in situations such as grid environments, where the number of available resources and tasks is high, we start to run out of memory and the overhead will increase fast reaching not assumable boundaries. Comparing our solution with other scheduling approaches, it easily integrates resources and tasks from different sources and the same engine can be extended and used without re-implementations or plug-ins. Moreover, rules can be loaded and unloaded at runtime,

so it enables the adaptation of the scheduling to the current load by selecting the proper scheduling rules for the type of computing tasks and resources.

Once the application model has been inferred and the different components have been allocated in the different providers resources, it is time to provision the selected resources as well as deploying and executing the application components on these resources. These tasks are achieved by invoking a workflow of calls exposed in the Infrastructure Provider interfaces. However, every provider defines their own interfaces, so the workflow to perform the same actions will be different depending on the provider, complicating the usage of different providers. In Chapter 5, we have proposed a framework for solving this problem and facilitating the Infrastructure provider interoperability. We have presented a methodology to automatically infer the set of provider interface calls required to perform any resource management tasks by applying different AI techniques. This framework includes a Semantic Annotation/De-annotation component, which is in charge of automatically generating semantic descriptions from interface descriptions and calls and vice versa. Then, translations between the different models are performed by rule reasoning and AI planning. From one hand, users define data equivalences by means of rules, and these rules are applied by the Resource Mapper component to convert data from one provider model to another one. On the other hand, the Action Planner component is in charge of finding the action equivalence. To do it, it generates an AI planning problem form the requested management tasks and the provider model. This problem defines the initial and goal resource states, and an AI planning domain which models the state transitions with the available provider interface methods. The result of this AI planning problem provides the sequence of provider actions which performs the requested management action. The proposed Infrastructure Interoperability framework has been applied in two scenarios. The first scenario is about inferring the workflow of required providers' actions for deploying the application components, and the second scenario is about translating interfaces between providers. In both scenarios, we have evaluated the overhead introduced by the framework. Despite it grows polynomially it is still reasonable for the size of the problem and most of the management actions. Finally, we have compared our solution with similar interoperability solutions which are based on open interfaces that allow the operation with different infrastructure providers by means of plug-ins. Our approach reduces the complexity of plug-in implementation and maintenance by just defining and maintaining a set of rules for defining equivalences between data models. The rest of the translation is automatically performed by the system. Moreover, plug-in approaches are statically bound to a specific source interface which contains the different plug-ins and drivers implementations. Our system is more flexible

than these approaches, it does not force users to learn a new interface and data model; it brings the opportunity to use the interface which they are more familiar.

In the last part of the thesis, we have studied how to introduce the benefits of software agents in the management of applications in distributed platforms. We have designed and implemented a multi-agent system which is in charge of coordinating the different steps of the application deployment in a distributed way as well as monitoring the correct execution of the application in the computing resources. Two types of agents have been defined: Application Agents, which are in charge of the application management; and Infrastructure Agents, which are in charge of the resource management. Moreover, these agents also share the functionality of semantic resource allocation described in Chapter 4 which has been split into two parts. On the one hand, the Application Agent is in charge of requesting proposals for allocating application and selecting the best one for the customer's interests. On the other hand, the Infrastructure Agent is in charge of making allocation proposals for applications at its resources according to the provider's interests. When the application is submitted to the system, an Application Agent starts to manage it taking different actions according to the application status. The application starts in a *submitted* status, at this state, the Application Agent performs the deployment model inference as presented Chapter 3 and negotiates with the other Infrastructure Agents an allocation for the application component instances according to the inferred deployment model. After a successful allocation negotiation, the Infrastructure Agent of the assigned resource is in charge of deploying and running the application components by means of the Interoperability framework described in Chapter 5. Once the application is running, the software agents monitor the application execution and react to unexpected execution events. From one hand, the Application agent is in charge of monitoring the application load and recalculating the required resources and adapting the deployed application according to the new requirements. On the other hand, the Infrastructure Agent is in charge of monitoring their resources and in case that a failure happens notifies the affected agents in order to recover the application from failures. We have implemented a prototype of this system, and we have measured the overhead in different situations. We have compared the distributed semantic resource allocation approach with the centralized approach described in Chapter 4. We have identified the most important processes and overheads in both approaches. As a result of this evaluation, we have detected the centralized approach performs better when the number of resources is low due to an important negotiation overhead while the distributed approach is a better option when the number of resources is big. Comparing our solution with other approaches, our solution was the first one to integrate semantics technologies in the resource allocation, facilitating

the integration with applications and resources coming from different sources as well as integrating the customer and provider policies in the same allocation processes. Moreover, current middleware solutions have recently added fault tolerance or adaptation services. These solutions are based on central services which must be configured by the user. In contrast, the software agents of our multi-agent approach already have integrated the self-adaptation and fault-tolerance features, so they are inherently distributed across the agents and do not require extra configuration steps.

7.1 Future work

Despite of applying the results of this thesis we can facilitate the deployment of applications to distributed computing platforms, there are still some parts were an important improvement can be achieved. One of these parts is the Application Model. We have proposed a general-purpose and infrastructure-agnostic model for describing distributed applications which includes all the required information to automatically deploy the application in the computing infrastructure. However, the description of some parts of this model is not trivial for non-expert developers like scientists. One of these parts is the definition of Quality Rules. They model the relationship between the application load, quality level and the resource requirements. Expert developers could discover this relationship by executing a set of execution with different configuration in order to profile the application and manually estimate the relationship. Machine Learning techniques could replicate this process helping the users to automatically extract the application Quality Rules. With supervised learning, the system could infer a model to estimate the number of required resources depending on the application computational load from the monitoring data extracted from previous executions. Then, the model obtained from machine learning system can be translated to the application Quality Rules. This automatic process for extracting the Quality Rules can considerably simplifies the application description as well as saves a lot of time to developers.

Another drawback of current Quality Rule description solutions is the adaptation solution. The Quality Rules are expressed based on a previous experience, however if the infrastructure provider, where the application is finally deployed is not giving the performance expected from its description, the quality-load-requirements relationship expressed by Quality Rules must adapted to cover the new kind of instances. Following the machine learning approach introduced before, the quality-load-requirements model could be recalculated at runtime using the monitoring data once the application is deployed. Applying a on-line supervised learning algorithm, we could update the Quality

Rule within a short processing time. Applying this update mechanism, we could reduce the effect of possible estimation problems and a better application adaptation.

Another part of improvement could be the Semantic Resource Allocation and Infrastructure Interoperability reasoning. In the thesis, we have seen reasoning algorithms are capable to solve current type of problems with the current amount of data and the complexity of the reasoning. However, the current trends in computer architecture is to have more and more small computing resources instead of having bigger resources and, on the side of applications, the trend is to have applications which combines more and more components specially with the appearance of new paradigms such as Internet of Things [134]. Therefore, we could have much more data to evaluate and the processing capabilities for traditional semantic reasoning algorithms could not grow with the same scale. In the near future, we will require adapting the semantic reasoning algorithms to work with big amount of data. This is a problem that big data researchers are tackling, they are proposing efficient distributed storage techniques and new frameworks for in-memory computation, so another important topic for future research is to improve the reasoning performance by applying big data techniques to the semantic reasoning computation.

The last part where we see a window for future research is the area of image management. Current infrastructure providers just offer a set of pre-defined images or allow the users to import an image. In the thesis, we have proposed a model to describe these images and how the application components are installed, configured and executed, comparing the image description with the application components description assigned to each resource, we can extract the required modification and apply them at deployment time by using dev-ops tools. The main drawback of this model is that installation steps can require too much time to apply them at deployment time. An initial deployment time do not have too much effect to the application performance, but the adaptation mechanism requires fast deployment mechanisms. To achieve it, we should apply the large changes before deployment, but until now, the only way to do it was to modify the image by hand and upload the modified image to the provider and recent research project like Optimis [32] or ASCETiC [135] are working on providing services to perform image changes at both construction an deployment time. This image management service could be added to the provider description in order to enable the Infrastructure Interoperability framework to infer deployment workflows with a more efficient image management.

Besides the possible improvements, distributed computing platforms are continuously evolving and our proposed system should take into account these new proposals. One of the latest proposals for managing the application resources is the introduction of software containers such as Docker [136] or the Google Container Engine [137]. As mentioned in

the previous paragraph, the deployment time is very important for application adaptation because the reaction to short peak of computational load requires a fast deployment of new VMs. Depending on the duration of the peak, the deployment of VMs could be slow because, for a complete VM deployment, the cloud middleware has to move and boot images with a complete Operating System. The container solutions have improved the deployment time by making an efficient system boot and keeping just the interesting capabilities of virtualization for the application execution. For this reason, the container approach is substituting the traditional VMs for providing an elastic application resource management. To integrate container providers with our system, we should map containers as a subclass of resources and describe the interface to manage containers with the providers' models. Due to the similarities with VMs, this task should not be too complicated. A more complicated integration could be the case of the image management. Despite the image description could be reused, the VMs and containers perform a different image management; so, further investigation is required in this field.

Bibliography

[1] Ian Foster and Carl Kesselman, editors. *The Grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. 1.1

[2] Thomas Erl. Service-oriented architecture. *Concepts, Technology, and Design*, 2004. 1.1

[3] Michael A Rappa. The utility business model and the future of computing services. *IBM Systems Journal*, 43(1):32–42, 2004. 1.1

[4] L. Vaquero, et al. A break in the clouds: Towards a cloud definition. *ACM SIGCOMM Computer Communications Review*, 39(1):50–55, 2009. 1.1

[5] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001. 1.3

[6] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning: theory & practice*. Elsevier, 2004. 1.3

[7] Jacques Ferber. *Multi-agent systems: an introduction to distributed artificial intelligence*, volume 1. Addison-Wesley Reading, 1999. 1.3, 6.1

[8] Open Nebula.
Web page at: http://opennebula.org/,
Date of last access: September 2015. 2.1

[9] Open Stack.
Web page at: https://www.openstack.org/,
Date of last access: September 2015. 2.1

[10] Apache Cloud Stack.
Web page at: https://cloudstack.apache.org/,
Date of last access: September 2015. 2.1

[11] The Open Grid Forum.
Web page at: http://www.ogf.org/,
Date of last access: September 2015. 2.1.1

[12] Open Cloud Computing Interface.
Web page at: http://occi-wg.org/,
Date of last access: September 2015. 2.1.1

[13] vCloud API Specification v1.0.
Available at: http://communities.vmware.com/docs/DOC-12464/,
Date of last access: September 2015. 2.1.1

[14] Distributed Management Task Force.
Web page at: http://www.dmtf.org,
Date of last access: September 2015. 2.1.1

[15] Apache jCloulds.
Web page at: http://jclouds.apache.org/,
Date of last access: September 2015. 2.1.1

[16] Apache Delta-Cloud .
Web page at: https://deltacloud.apache.org/,
Date of last access: September 2015. 2.1.1

[17] The Dasein Cloud API.
Web page at: http://dasein-cloud.sourceforge.net/,
Date of last access: September 2015. 2.1.1

[18] Heroku.
Web page at: http://www.heroku.com/,
Date of last access: September 2015. 2.2

[19] Cloud Control.
Web page at: https://www.cloudcontrol.com/,
Date of last access: September 2015. 2.2

[20] Spring MVC Framework.
Web page at: https://spring.io/,
Date of last access: September 2015. 2.2

[21] Django Web Framework.
Web page at: https://www.djangoproject.com/,
Date of last access: September 2015. 2.2

[22] Ruby on Rails Web Framework.
Web page at: http://rubyonrails.org/,
Date of last access: September 2015. 2.2

[23] Google App Engine.
Web page at: https://cloud.google.com/appengine,
Date of last access: September 2015. 2.2

[24] Microsoft Azure App Service.
Web page at: https://azure.microsoft.com/services/app-service/,
Date of last access: September 2015. 2.2

[25] Amazon Cloud Formation.
Web page at: http://aws.amazon.com/cloudformation/,
Date of last access: September 2015. 2.2

[26] Cloud Foundry.
Web page at: http://cloudfoundry.org,
Date of last access: September 2015. 2.2

[27] Cloudify.
Web page at: http://getcloudify.org/,
Date of last access: September 2015. 2.2

[28] Cloud Pier.
Web page at: http://www.opencloudpier.org,
Date of last access: September 2015. 2.2

[29] N. Loutas, et al. Towards a Reference Architecture for Semantically Interoperable Clouds. In *Int. Conf. on Cloud Computing Technology and Science*, pages 143–150, 2010. 2.2

[30] Distributed Management Task Force. Open Virtualization Format Specification. DSP0243, 2013. 2.3

[31] B. Rochwerger, et al. The Reservoir model and architecture for open federated cloud computing. *Journal of Research and Development*, 53(4):535–545. 2.3

[32] Ana Juan Ferrer, Francisco Hernández, Johan Tordsson, Erik Elmroth, Ahmed Ali-Eldin, Csilla Zsigri, RaüL Sirvent, Jordi Guitart, Rosa M Badia, Karim Djemame, et al. Optimis: A holistic approach to cloud service provisioning. *Future Generation Computer Systems*, 28(1):66–77, 2012. 2.3, 7.1

[33] Venus-C project.
Web page at: http://www.venus-c.eu/,
Date of last access: September 2015. 2.3

[34] mOSAIC Cloud project.
Web page at: http://www.mosaic-cloud.eu/,
Date of last access: September 2015. 2.3, 5.2

[35] Remics Project.
Web page at: http://www.remics.eu/,
Date of last access: September 2015. 2.3

[36] PaaSage Project.
Web page at: http://www.paasage.eu/,
Date of last access: September 2015. 2.3

[37] MODA Clouds Project.
Web page at: http://www.modaclouds.eu/,
Date of last access: September 2015. 2.3

[38] Francesco Moscato, Rocco Aversa, Beniamino Di Martino, Dana Petcu, Massimiliano Rak, and Salvatore Venticinque. An ontology for the cloud in mosaic. *Cloud Computing: Methodology, Systems, and Application*, pages 467–486, 2011. 2.3

[39] Cloud ML.
Web page at: http://cloudml.org/,
Date of last access: September 2015. 2.3

[40] Dereck Palma and Thomas Spatzier. Topology and Orchestration Specification for Cloud Applications Version 1.0. *Organization for the Advancement of Structured Information Standards(OASIS)*, 2013. 2.3

[41] Robert L Henderson. Job scheduling under the portable batch system. In *Job scheduling strategies for parallel processing*, pages 279–294. Springer, 1995. 2.4

[42] Load Sharing Facility.
Web page at: http://www.ibm.com/systems/platformcomputing/products/lsf,
Date of last access: September 2015. 2.4

[43] Maui Cluster Scheduler.
Web page at: http://www.adaptivecomputing.com/products/open-source/maui/,
Date of last access: September, 2015. 2.4

[44] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997. 2.4

[45] Dietmar W. Erwin and David F. Snelling. *Euro-Par 2001 Parallel Processing*, chapter UNICORE: A Grid Computing Environment, pages 825–834. Springer, 2001. 2.4

[46] E. Laure, S. M. Fisher, A. Frohner, C. Grandi, P. Kunszt, A. Krenek, O. Mulmo, F. Pacini, F. Prelz, J. White, M. Barroso, P. Buncic, F. Hemmer, A. Di Meglio, and A. Edlund. Programming the Grid with gLite. *Computational Methods in Science and Technology*, 12(1):33–45, 2006. 2.4

[47] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. In *Job Scheduling Strategies for Parallel Processing*, pages 62–82. Springer, 1998. 2.4

[48] Eduardo Huedo, Ruben S. Montero, and Ignacio M. Llorente. A framework for adaptive execution in grids. *Software: Practice and Experience*, 34(7):631–651, 2004. 2.4

[49] Volker Hamscher, Uwe Schwiegelshohn, Achim Streit, and Ramin Yahyapour. Evaluation of job-scheduling strategies for grid computing. In *Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, pages 191–202. Springer-Verlag, 2000. 2.4

[50] Miron Livny, Jim Basney, Rajesh Raman, and Todd Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP journal*, 11(1):36–40, 1997. 2.4

[51] Ivan Rodero, Francesc Guim, Julita Corbalan, Liana Fong, and S Masoud Sadjadi. Grid broker selection strategies using aggregated resource information. *Future Generation Computer Systems*, 26(1):72–86, 2010. 2.4

[52] Farag Azzedin and Muthucumaru Maheswaran. Towards trust-aware resource management in grid computing systems. In *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 452–452. IEEE, 2002. 2.4

[53] Rich Wolski, James S Plank, John Brevik, and Todd Bryan. Analyzing market-based resource allocation strategies for the computational grid. *International Journal of High Performance Computing Applications*, 15(3):258–281, 2001. 2.4

[54] S. Venugopal, R. Buyya, and L. J. Winton. A Grid service broker for scheduling e-Science applications on global data Grids. *Concurrency and Computation: Practice and Experience*, 18(6):685–699, 2006. 2.4

[55] Luis Tomás, Agustín C. Caminero, Carmen Carrión, and Blanca Caminero. Network-aware Meta-scheduling in Advance with Autonomous Self-tuning System. *Future Generation Computer Systems*, 27(5):486–497, 2011. 2.4

[56] Javier Conejero, Blanca Caminero, Carmen Carrión, and Luis Tomás. From volunteer to trustable computing: Providing qos-aware scheduling mechanisms for multi-grid computing environments. *Future Generation Computer Systems*, 34:76–93, 2014. 2.4

[57] Anton Beloglazov and Rajkumar Buyya. Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints. *Parallel and Distributed Systems, IEEE Transactions on*, 24(7):1366–1379, 2013. 2.4

[58] Íñigo Goiri, Josep Ll Berral, J Oriol Fitó, Ferran Julià, Ramon Nou, Jordi Guitart, Ricard Gavaldà, and Jordi Torres. Energy-efficient and multifaceted resource management for profit-driven virtualized data centers. *Future Generation Computer Systems*, 28(5):718–731, 2012. 2.4

[59] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future generation computer systems*, 28(5):755–768, 2012. 2.4

[60] Eyal Bin, Ofer Biran, Odellia Boni, Erez Hadad, Eliot K Kolodner, Yosef Moatti, and Dean H Lorenz. Guaranteeing high availability goals for virtual machine placement. In *Proceedings of the 31st International Conference on Distributed Computing Systems*, pages 700–709. IEEE, 2011. 2.4

[61] Fumio Machida, Masahiro Kawato, and Yoshiharu Maeno. Redundant virtual machine placement for fault-tolerant consolidated server clusters. In *Proceedings of the IEEE Network Operations and Management Symposium*, pages 32–39. IEEE, 2010. 2.4

[62] David De Roure, Nicholas R Jennings, and Nigel Shadbolt. The semantic grid: A future e-science infrastructure. *Grid Computing-Making the Global Infrastructure a Reality*, pages 437–470, 2003. 2.5

[63] EU OntoGrid project.
Web page at: http://www.ontogrid.net. 2.5

[64] Oscar Corcho, Pinar Alper, Ioannis Kotsiopoulos, Paolo Missier, Sean Bechhofer, and Carole Goble. An overview of s-ogsa: A reference semantic grid architecture. *Web Semantics: Science, Services and Agents on the World Wide Web*, 4(2):102–115, 2006. 2.5

[65] Hongsuda Tangmunarunkit, Stefan Decker, and Carl Kesselman. Ontology-based resource matching in the grid–the grid meets the semantic web. In *The Semantic Web-ISWC 2003*, pages 706–721. Springer, 2003. 2.5

[66] NV Neela and S Kailash. Resource matchmaking in grid-semantically. In *Proceedings of the 9th International Conference on Advanced Communication Technology*, volume 3, pages 2051–2055. IEEE, 2007. 2.5

[67] Zoi Kaoudi, Iris Miliaraki, Matoula Magiridou, Erietta Liarou, Stratos Idreos, and Manolis Koubarakis. Semantic grid resource discovery in atlas. In *Knowledge and Data Management in Grids*, pages 185–199. Springer, 2007. 2.5

[68] SPARQL Query Language for RDF.
Web page at: http://www.w3.org/TR/rdf-sparql-query,
Date of last access: September 2015. 2.5

[69] Balachandar R Amarnath, Thamarai Selvi Somasundaram, Mahendran Ellappan, and Rajkumar Buyya. Ontology-based grid resource management. *Software: Practice and Experience*, 39(17):1419–1438, 2009. 2.5

[70] Thamarai Selvi Somasundaram, Kumar Rangasamy, and Kannan Govindarajan. Intelligent semantic discovery in virtualized grid environment. In *Proceeding of the International Conference on Trends in Information Technology*, pages 644–649. IEEE, 2011. 2.5

[71] Thamarai Selvi Somasundaram, Kannan Govindarajan, Usha Kiruthika, and Rajkumar Buyya. Semantic-enabled care resource broker (secrb) for managing grid and cloud environment. *The Journal of Supercomputing*, 68(2):509–556, 2014. 2.5

[72] Grid Interoperabilty Project.
Web page at: http://www.grid-interoperability.eu/. 2.5

[73] John Brooke, Donal Fellows, Kevin Garwood, and Carole Goble. Semantic matching of grid resource descriptions. In *Grid Computing*, pages 240–249. Springer, 2004. 2.5

[74] Ontology Web Language (OWL).
Web page at: http://www.w3.org/TR/owl-features. 2.5

[75] Amir Vahid Dastjerdi, Sayed Gholam Hassan Tabatabaei, and Rajkumar Buyya. An effective architecture for automated appliance management system applying ontology-based cloud discovery. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 104–112. IEEE, 2010. 2.5

[76] Le Duy Ngan and Rajaraman Kanagasabai. Owl-s based semantic cloud service broker. In *Proceedings of the IEEE 19th International Conference on Web Services*, pages 560–567. IEEE, 2012. 2.5

[77] C Pittaras, C Papagianni, A Leivadeas, P Grosso, J van der Ham, and S Papavassiliou. Resource discovery and allocation for federated virtualized infrastructures. *Future Generation Computer Systems*, 42:55–63, 2015. 2.5

[78] Yu Deng, Michael R Head, Andrzej Kochut, Jonathan Munson, Anca Sailer, and Hidayatullah Shaikh. Introducing semantics to cloud services catalogs. In *Proceedings of the IEEE International Conference on Services Computing*, pages 24–31. IEEE, 2011. 2.5

[79] Teodor-Florin Fortiş, Victor Ion Munteanu, and Viorel Negru. Towards an ontology for cloud services. In *Proceedings of the 6th International Conference on Complex, Intelligent and Software Intensive Systems*, pages 787–792. IEEE, 2012. 2.5

[80] Beniamino Di Martino, Giuseppina Cretella, and Anna Esposito. Towards a unified owl ontology of cloud vendors' appliances and services at paas and saas level. In *Proceedings of the 8th International Conference on Complex, Intelligent and Software Intensive Systems*, pages 570–575. IEEE, 2014. 2.5

[81] Eleni Kamateri, Nikolaos Loutas, Dimitris Zeginis, James Ahtes, Francesco DâĂŹAndria, Stefano Bocconi, Panagiotis Gouvas, Giannis Ledakis, Franco Ravagli, Oleksandr Lobunets, et al. Cloud4soa: A semantic-interoperability paas solution for multi-cloud platform management and portability. In *Service-Oriented and Cloud Computing*, pages 64–78. Springer, 2013. 2.5

[82] I. Foster, N. Jennings, C. Kesselman. Brain meets brawn: Why grid and agents need each other. In *Porceedings of the 3rd International Conference on Autonomous Agents and Multiagent Systems*, 2004. 2.6, 6.6

[83] A. Chavez, A. Moukas, and P. Maes. Challenger: A multi-agent system for distributed resource allocation. In *Proceeding of the 1st International Conference on Autonomous Agents*, 1997. 2.6, 6.6

[84] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B.A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent and Grid Systems*, 1(3):169–182, 2005. 2.6, 6.6

[85] Y. Chevaleyre, U. Endriss, S. Estivie, and N. Maudet. Welfare engineering in practice: On the variety of multiagent resource allocation problems. *Engineering Societies in the Agents World V*, pages 335–347, 2005. 2.6, 6.6

[86] S. Parsons and M. Wooldridge. Game theory and decision theory in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 5(3):243–254, 2002. 2.6, 6.6

[87] S.S. Fatima and M. Wooldridge. Adaptive task resources allocation in multi-agent systems. In *The 5th International Conference on Autonomous Agents*, 2001. 2.6, 6.6

[88] SS Manvi, MN Birje, and B. Prasad. An agent-based resource allocation model for computational grids. *Multiagent and Grid Systems*, 1(1):17–27, 2005. 2.6, 6.6

[89] CatNets Project.
Web page at: http://www.catnets.org,
Date of last access: September 2015. 2.6, 6.6

[90] Oscar Ardaiz, Pau Artigas, Torsten Eymann, Felix Freitag, Leandro Navarro, and Michael Reinicke. The catallaxy approach for decentralized economic-based allocation in grid resource and service markets. *Applied Intelligence*, 25(2):131–145, 2006. 2.6

[91] Sorma Project.
Web page at: http://sorma-project.org/,
Date of last access: September 2015. 2.6, 6.6

[92] Salvatore Venticinque, Rocco Aversa, Beniamino Di Martino, Massimilano Rak, and Dana Petcu. A cloud agency for sla negotiation and management. In *Euro-Par 2010 Parallel Processing Workshops*, pages 587–594. Springer, 2011. 2.6

[93] Gopal Kirshna Shyam and SunilKumar S Manvi. Resource allocation in cloud computing using agents. In *Advance Computing Conference (IACC), 2015 IEEE International*, pages 458–463. IEEE, 2015. 2.6

[94] Jakub Gąsior and Franciszek Seredyński. A decentralized multi-agent approach to job scheduling in cloud environment. In *Intelligent Systems' 2014*, pages 403–414. Springer, 2015. 2.6

[95] Fernando De la Prieta, Sara Rodríguez, Javier Bajo, and Juan M Corchado. + cloud: A virtual organization of multiagent system for resource allocation into a cloud computing environment. In *Transactions on Computational Collective Intelligence XV*, pages 164–181. Springer, 2014. 2.6

[96] Mahmoud Al-Ayyoub, Yaser Jararweh, Mustafa Daraghmeh, and Qutaibah Althebyan. Multi-agent based dynamic resource provisioning and monitoring for cloud computing systems infrastructure. *Cluster Computing*, 18(2):919–932, 2015. 2.6

[97] Ken Wenzel and Heiner Reinhardt. Mathematical Computations for Linked Data Applications with OpenMath. In *Proceedings of the 24th Workshop on OpenMath*, pages 38–48, 2012. 3.2.2

[98] PuppetLabs.
Web page at: https://puppetLabs.com,
Date of last access: September 2015. 3.2.3, 5.5.1

[99] Large Scale Unix Configuration System.
Web page at: http://www.lcfg.org,
Date of last access: September 2015. 3.2.3

[100] B. Motik, et al. OWL 2 Web Ontology Language. W3C Recommendation, 2012. 3.4, 5.5

[101] I. Horrocks, et al. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission, 2004. 3.4

[102] E. Sirin, et al. Pellet: A practical owl-dl reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007. 3.4

[103] Enric Tejedor and Rosa M. Badia. COMP Superscalar: Bringing GRID Superscalar and GCM Together. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid*, pages 185–193, 2008. 3.4.1

[104] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 3.4.1

[105] The Message Passing Interface standard.
Web page at: http://www-unix.mcs.anl.gov/mpi/,
Date of last access: September 2015. 3.4.1

[106] The OpenMP Homepage.
Web page at: http://www.openmp.org/drupal/,
Date of last access: September 2015. 3.4.1

[107] András Micsik, Péter Pallinger, and Dávid Siklósi. Scaling a plagiarism search service on the bonfire testbed. In *Proceedings of the 5th International Conference on Cloud Computing Technology and Science*, volume 2, pages 57–62, 2013. 3.4.1

[108] R. Royo, J. López, D. Torrents, and J.L. Gelpi. A BioMoby-based workflow for gene detection using sequence homology. In *Proceeding of the International Supercomputing Conference*, 2008. 3.4.1

[109] BLAST Home Page. http://www.ncbi.nlm.nih.gov/BLAST/. 3.4.1

[110] Ewan Birney, Michele Clamp, and Richard Durbin. Genewise and genomewise. *Genome research*, 14(5):988–995, 2004. 3.4.1

[111] S. Andreozzi, et al. GLUE Schema Specification. Open Grid Forum Recomendation GFD-147, 2009. 4.2

[112] Distributed Management Task Force. Common Information Model v.3.0. DSP0004, 2013. 4.2, 5.2

[113] Grid Resource Ontology.
Web page at: http://www.unigrids.org/, Date last access: September 2015. 4.2

[114] B. Lithgow Smith, C. van Aart, M. Wooldridge, S. Paurobally, T. Moyaux, and V. Tamma. An Ontological Framework for Dynamic Coordination. In *Proceedings of the Fourth International Semantic Web Conference*, 2005. 4.2

[115] Apache Jena Semantic Web Framework.
Web page at: https://jena.apache.org/, Date last access: September 2015. 4.4.2, 5.5, 6.5

[116] Charles L Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19(1):17–37, 1982. 4.4.2

[117] J. Ejarque, et al. Using semantics for resource allocation in computing service providers. In *Proceeding of the IEEE International Conference on Services Computing*, volume 2, pages 583–587, 2008. 5.2

[118] D. Martin, et al. OWL-S: Semantic markup for web services. *W3C member submission*, 2004. 5.2

[119] C Michael Sperberg-McQueen and Eric Miller. On mapping from colloquial xml to rdf using xslt. In *Proceedings of Extreme Markup Languages*, 2004. 5.2.1

[120] Converters to RDF .
Web page at: http://www.w3.org/wiki/ConverterToRdf,
Date of last access: September, 2015. 5.2.1

[121] Web Service Description Language.
Web page at: http://www.w3.org/TR/wsdl,
Date of last access: September 2015. 5.2.1

[122] Web Application Description Language.
Web page at: http://www.w3.org/Submission/wadl,
Date of last access: September 2015. 5.2.1

[123] J. Kopeckỳ, K. Gomadam, T. Vitvar. hrests: An html microformat for describing restful web services. In *Proceedings of the International Conference on Web Intelligence and Intelligent Agent Technology*, 2008. 5.2.1

[124] Massimo Paolucci, Naveen Srinivasan, Katia Sycara, and Takuya Nishimura. Towards a semantic choreography of web services: from wsdl to daml-s. In *Proceedings of the First International Conference on Web Services*, pages 22–26, June 2003. 5.2.1

[125] M. Hert, G. Reif, H. C. Gall. Personal Knowledge Mapping with Semantic Web Technologies. In *Proceedings of the International Workshop on Personal Knowledge Management*, 2009. 5.3

[126] Anthony Barrett and Daniel S Weld. Partial-order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67(1):71–112, 1994. 5.4

[127] Planning4J - Java API for AI planning.
Web page at: http://code.google.com/p/planning4j,
Date of last access: September 2015. 5.5

[128] Bernhard Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001. 5.5

[129] Y. Chevaleyre, et al. Issues in Multiagent Resource Allocation. *Informatica*, 30:3–31, 2006. 6.1

[130] A.S. Rao and M.P. Georgeff. BDI agents: From theory to practice. In *Proceedings of the 1st International Conference on Multi-agent Systems*, 1995. 6.1

[131] RG Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, 100(29):1104–1113, 1980. 6.4

[132] Jadex active components.
Web page at: http://www.activecomponents.org. 6.5

[133] Java Agent DEvelopment Framework.
Web page at: http://jade.tilab.com/,
Date of last access: September 2015. 6.5

[134] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010. 7.1

[135] Karim Djemame, Django Armstrong, Richard Kavanagh, Ana Juan Ferrer, David Garcia Perez, David Antona, Jean-Christophe Deprez, Christophe Ponsard, David Ortiz, Mario Macias, et al. Energy efficiency embedded service lifecycle: Towards an energy efficient cloud computing architecture. In *CEUR Workshop Proceedings*, volume 1203, pages 1–6. CEUR Workshop Proceedings, 2014. 7.1

[136] Docker.
Web page at: https://www.docker.com,
Date of last access: September 2015. 7.1

[137] Google Container Engine.
Web page at: https://cloud.google.com/container-engine/,
Date of last access: September 2015. 7.1