UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
DIPARTIMENTO DI INFORMATICA

DOTTORATO DI RICERCA IN INFORMATICA
SCUOLA DI DOTTORATO IN INFORMATICA, XXVI CICLO

Dependability in Cloud Computing

TESI DI DOTTORATO DI RICERCA DI
Ravi Jhawar

RELATORE
Prof. Vincenzo Piuri

CORRELATORE
Prof. Pierangela Samarati

DIRETTORE DELLA SCUOLA DI DOTTORATO
Prof. Ernesto Damiani

Anno Accademico 2012/13

Abstract

The technological advances and success of Service-Oriented Architectures and the Cloud computing paradigm have produced a revolution in the Information and Communications Technology (ICT). Today, a wide range of services are provisioned to the users in a flexible and cost-effective manner, thanks to the encapsulation of several technologies with modern business models. These services not only offer high-level software functionalities such as social networks or e-commerce but also middleware tools that simplify application development and low-level data storage, processing, and networking resources. Hence, with the advent of the Cloud computing paradigm, today's ICT allows users to completely outsource their IT infrastructure and benefit significantly from the economies of scale.

At the same time, with the widespread use of ICT, the amount of data being generated, stored and processed by private companies, public organizations and individuals is rapidly increasing. The in-house management of data and applications is proving to be highly cost intensive and Cloud computing is becoming the destination of choice for increasing number of users. As a consequence, Cloud computing services are being used to realize a wide range of applications, each having unique dependability and Quality-of-Service (Qos) requirements. For example, a small enterprise may use a Cloud storage service as a simple backup solution, requiring high data availability, while a large government organization may execute a real-time mission-critical application using the Cloud compute service, requiring high levels of dependability (e.g., reliability, availability, security) and performance. Service providers are presently able to offer sufficient resource heterogeneity, but are failing to satisfy users' dependability requirements mainly because the failures and vulnerabilities in Cloud infrastructures are a norm rather than an exception.

This thesis provides a comprehensive solution for improving the dependability of Cloud computing – so that – users can justifiably trust Cloud computing services for building, deploying and executing their applications. A number of approaches ranging from the use of trustworthy hardware to secure application design has been proposed in the literature. The proposed solution consists of three inter-operable yet independent modules, each designed to improve dependability under different system context and/or use-case. A user can selectively apply either a single module or combine them suitably to improve the dependability of her applications both during design time and runtime. Based on the modules applied, the overall proposed solution can increase dependability at three distinct levels. In the following, we provide a brief description of each module.

The first module comprises a set of *assurance techniques* that validates whether a given service supports a specified dependability property with a given level of assurance, and accordingly, awards it a machine-readable certificate. To achieve this, we define a hierarchy of dependability properties where a property represents the dependability characteristics of the service and its specific configuration. A model of the service is also used to verify the validity of the certificate using runtime monitoring, thus complementing the dynamic nature of the Cloud computing infrastructure and making the certificate usable both at discovery and runtime. This module also extends the service registry to allow users to select services with a set of certified dependability properties, hence offering the basic support required to implement dependable applications. We note that this module directly considers services implemented by service providers and provides awareness tools that allow users to be aware of the QoS offered by potential partner services. We denote this passive technique as the solution that offers first level of dependability in this thesis.

Service providers typically implement a standard set of dependability mechanisms that satisfy the basic needs of most users. Since each application has unique dependability requirements, assurance techniques are not always effective, and a pro-active approach to *dependability management* is also required. The second module of our solution advocates the innovative approach of offering *dependability as a service* to users' applications and realizes a framework containing all the mechanisms required to achieve this. We note that this approach relieves users from implementing low-level dependability mechanisms and system management procedures during application development and satisfies specific dependability goals of each application. We denote the module offering dependability as a service as the solution that offers second level of dependability in this thesis.

The third, and the last, module of our solution concerns *secure application execution*. This module considers complex applications and presents advanced resource management schemes that deploy applications with improved optimality when compared to the algorithms of the second module. This module improves dependability of a given application by minimizing its exposure to existing vulnerabilities, while being subject to the same dependability policies and resource allocation conditions as in the second module. Our approach to secure application deployment and execution denotes the third level of dependability offered in this thesis.

The contributions of this thesis can be summarized as follows.

- With respect to assurance techniques our contributions are: *i)* definition of a hierarchy of dependability properties, an approach to service modeling, and a model transformation scheme; *ii)* definition of a dependability certification scheme for services; *iii)* an approach to service selection that considers users' dependability requirements; *iv)* definition of a solution to dependability certification of composite services, where the dependability properties of a composite service are calculated on the basis of the dependability certificates of component services.

- With respect to offering dependability as a service our contributions are: *i)* definition

of a delivery scheme that transparently functions on users' applications and satisfies their dependability requirements; *ii)* design of a framework that encapsulates all the components necessary to offer dependability as a service to the users; *iii)* an approach to translate high level users' requirements to low level dependability mechanisms; *iv)* formulation of constraints that allow enforcement of deployment conditions inherent to dependability mechanisms and an approach to satisfy such constraints during resource allocation; *v)* a resource management scheme that masks the affect of system changes by adapting the current allocation of the application.

- With respect to security management our contributions are: *i)* an approach that deploys users' applications in the Cloud infrastructure such that their exposure to vulnerabilities is minimized; *ii)* an approach to build interruptible elastic algorithms whose optimality improves as the processing time increases, eventually converging to an optimal solution.

Acknowledgements

I would like to use the occasion of this thesis to thank all the people who have helped and supported me, in different ways, during my PhD.

First and foremost, I would like to sincerely thank my advisors, Prof. Vincenzo Piuri and Prof. Pierangela Samarati. I am really grateful to them for introducing me to scientific research and for their constant presence, guidance, and support over the years. I would like to express them my gratitude for giving me all the opportunities and for graciously taking care of me not just academically but also personally. They always compassionately forgave my mistakes and motivated me in all situations. I consider myself very privileged to be one of their PhD students. Their passion for scientific research, work ethics, hard work and attention to details provide excellent role models for me.

I would like to thank Prof. Sabrina De Capitani di Vimercati, Prof. Ernesto Damiani, Dr. Claudio Ardagna and Dr. Sara Foresti for their answers to all my questions, for precious advices, and for encouraging me in all situations.

I would like to sincerely thank Prof. Sushil Jajodia for giving me the opportunity to visit the Center for Secure Information Systems, George Mason University, VA, USA. I am very grateful to him for his support and advices, and for providing a stimulating an enjoyable working atmosphere.

I will always be indebted to Dr. Kamal Jajodia for all the kindness, generosity, caring and concern that she has shown me. I would like to sincerely thank her for all that she has done for me.

I would like to thank Prof. Massimiliano Albanese and Dr. Marco Santambrogio for their help on the various aspects of the work presented in this thesis.

I am grateful to Prof. Vijay Atluri, Prof. Javier Lopez and Prof. Laurence Yang, the referees of this thesis, for having dedicated their precious time to review the thesis and whose valuable comments helped improving the presentation of this work.

I am grateful to my family for giving me the opportunity to pursue a PhD, and for their constant love and support. Their teachings have always been and will be a fundamental reference point for me.

Finally, I would like to thank my friends Mohammad Aktaruzzaman, Paolo Arcaini, Ruggero Donida Labati, Angelo Genovese, Giovanni Livraga and Massimo Rivolta.

Contents

List of Figures

List of Tables

1 Introduction

Individuals and enterprises are increasingly resorting to Cloud computing services for storage, processing, and management of their data and applications. This practice of relying on utility-based service-oriented computing is effective only when the users have some assurance that the dependability of their data and applications is not at risk. In this context, fault tolerance, high availability and security become of paramount importance for the users, and become pressing issues that need to be addressed.

This thesis is concerned with the definition of a comprehensive solution for improving the dependability of users' applications that leverage Cloud computing services. In the remainder of this chapter, we give the motivation and outline of this thesis.

1.1 Motivation

The increasing demand for flexibility in obtaining and releasing computing resources in a cost-effective manner has resulted in a wide adoption of the Cloud computing paradigm. This computing model offers several benefits to the users (application and data owners) with respect to traditional in-house management. First, users are relieved from buying expensive software licenses and hardware, and recruiting skilled personnel to administer and maintain computing resources, thus providing significant economic savings. Second, users can access their data and applications using any device providing Internet connectivity. Third, the availability of an extensible pool of resources provide incentives for the users to deploy applications with high scalability and processing requirements.

In general, the Cloud computing architecture consists of a stack of services at three distinct layers. While the services in the bottommost Infrastructure-as-a-Service (IaaS) layer provision low-level computational resources and allow users to store, execute, and manage their data and applications, the services belonging to the Platform-as-a-Service (PaaS) layer provide a set of middleware tools that simplify application development and deployment. The topmost Software-as-a-Service (SaaS) layer makes a wide range of high-level web services available to the users which, when coupled with standard XML-based protocols, allow for the design and dynamic composition of applications and business

processes. However, this architecture has significantly changed the dimension of risks in two respects. First, failures and vulnerabilities in the physical infrastructure, or services at a given layer, affect services not only in the same layer but also in the layers above it. For example, a failure in the storage service may impact several PaaS and SaaS services. This implies that a given solution must now take into account the risks across all the layers in the Cloud. Second, failures and vulnerabilities are typically outside the control scope of the users' organization. This results in an increasing concern among users about the risks that may affect the functional and non-functional properties of their applications, and a given solution must allow users to rely on techniques that address their dependability concerns at different levels.

This thesis concerns with the definition of a comprehensive solution for reducing the risks of using Cloud computing services. In particular, our goal is to improve the dependability of Cloud computing, so that the use of SaaS, PaaS and IaaS services for building, deploying, and executing applications can justifiably be trusted. A number of approaches ranging from the design of trustworthy hardware to secure application development have been proposed in the literature. Our approach consists in designing three inter-operable yet independent modules, each making different assumptions on the system context, use cases and desired dependability features. Based on the modules applied, the proposed solution allows users to improve the dependability of their applications progressively at three distinct levels. A brief outline of different modules of the overall solution is provided in the following.

- *Dependability assurance of services.* As the first step, we consider Cloud computing services in their present form (where dependability mechanisms are implemented by service providers), and provide a set of *assurance techniques* that validate, verify and *certify* dependability properties of services. Such assurance techniques allow users to be *aware* of the Quality of Service (QoS) offered by a given service, and increases users' confidence regarding the dependability of their partner services. This module allows users to select services satisfying their dependability requirements, and consequently, provides the basic support necessary to realize dependable applications. *Assurance techniques* denote the *first level* of dependability solution offered in this thesis (see Chapter 3). We note that awareness tools are necessary in our context because, at present, there is neither a clear solution that allows verification of SLAs at runtime nor a unified framework that selects services based on dependability properties.

- *Dependability management.* Service providers typically implement a standard set of dependability mechanisms that satisfy the basic needs of most users. However, a Cloud computing service may host a wide range of applications, each with unique dependability requirements. This implies that simple assurance techniques may not be sufficient and a more pro-active approach that satisfies specific dependability goals of individual applications is necessary.

Based on the above observation, we put forward an innovative notion of offering *dependability as a service*, allowing users to apply desired dependability properties for each application without having to implement low-level mechanisms. We achieve this by translating high-level users' requirements to low-level dependability policies and resource allocation conditions, and integrating such policies and conditions in the resource management process. This module provides higher level of dependability when compared to assurance techniques since Cloud services are managed to satisfy specific dependability goals of each application. Our techniques offering *dependability as a service* denote the *second level* of dependability solution offered in this thesis (see Chapters 4, 5). We note that this *flexible solution* also overcomes users' difficulties in implementing dependable applications particularly arisen due to high complexity and abstraction layers of Cloud computing.

- *Secure application execution.* As the third step, we consider complex applications, and present advanced resource management schemes that deploys applications with improved optimality when compared to algorithms at second level. In particular, the aspect of *secure application execution* improves dependability of a given application by minimizing its exposure to existing vulnerabilities, while being subject to same dependability policies and resource allocation conditions as in the second level. This module denotes the *third level* of dependability solution offered in this thesis (see Chapter 6). We note that a solution to secure application deployment is necessary because: existing security solutions either design the infrastructure using network hardening tools or develop applications using security measures, but fail to take into account the complex interdependencies between the infrastructure, users' applications, and residual vulnerabilities in the system. This implies that it is not possible to remediate all existing vulnerabilities and applications have to be executed on the infrastructure containing multiple and interdependent vulnerabilities.

The three aforementioned modules together provide a comprehensive solution for improving the dependability of Cloud computing. Users can rely on solutions corresponding to each aspect either individually or complement them with one another, based on their specific dependability goals. For example, a user can simply implement the functional aspects of her application and rely on the techniques offering dependability as a service to improve its reliability and availability. She can rely on techniques for secure application execution to further improve dependability and robustness. In contrast, a SaaS provider can implement her service using a set of dependability mechanisms by relying on services satisfying her dependability requirements, based on assurance techniques.

There are many real-world scenarios that need our dependability solution to safely benefit from Cloud computing. We outline two examples of great practical relevance.

E-commerce applications. A major factor in the evolution of the Web has been the widespread diffusion of electronic commerce (e-commerce) applications. Such applications

comprise online shopping, banking, and other financial services that foster the ability of customers to purchase, sell, and distribute goods and services over the Internet. Traditionally, the primary concern in the development of e-commerce was to provide a dependable infrastructure through solutions for secure and reliable data exchange and systems for prevention of unauthorized accesses. However, with the advent of Cloud computing, the focus has shifted from mere server-side static solutions to development of approaches that allow users in selectively adopting dependability tools that address their specific business requirements.

- If the e-Commerce business owner do not have confidence that its dependability requirements are not satisfied in the envisioned Cloud, they may not shift their critical business processes to external Cloud computing services.

- Failures and vulnerabilities in the Cloud infrastructure may become a bottleneck for the e-Commerce business owner, and requires flexible solutions of acquiring desired dependability properties for its applications.

Scientific and mission-critical applications. Generally, scientific applications involve processing intensive calculations (e.g., matrix multiplications, image and signal processing) and require highly available systems. Traditionally, these needs have been addressed by using high-performance computing solutions and installed facilities such as clusters and super computers, which are difficult to setup, maintain, and operate. On the other hand, Cloud computing provides scientists a new extensible model of utilizing compute, storage, and middleware resources, while changing the dimension of risks for applications. This situation requires new solutions for selecting dependable services, flexibly acquiring fault tolerance properties, and securely operating applications in the Cloud infrastructure. Such requirements directly apply also for mission-critical applications.

From the above description, it is straightforward to see that dependability problems envisioned for e-Commerce systems apply also for scientific and mission-critical applications scenario (with varying intensity), and demands user-centric solutions and technologies for guaranteeing dependability.

1.2 Contributions of the Thesis

The contributions of this thesis are centered on three related topics:

1. Definition of dependability certification scheme for Cloud computing services;

2. Definition of an innovative approach to dependability management, where users' applications can obtain desired dependability properties from an additional service, offered by a third party;

3. Definition of an application-centric framework that integrates vulnerability and resource management schemes to improve dependability of complex applications.

The remainder of this section discusses our original contributions in detail.

1.2.1 Dependability Certification of Services

The definition of assurance techniques increasing users' confidence that a service complies with their dependability requirements becomes of utmost importance in the context of Cloud computing and web services. A suitable technique to address this concern is based on certification [31]. Originally certification schemes targeted static and monolithic software, and produced human-readable certificates to be used at installation time [128, 55]. In this thesis, we define an approach that supports machine-readable certificates that verify dependability properties of services with a given level of assurance and prove service robustness against a set of failures. The main characteristics of our certification scheme can be summarized as follows.

Definition of a hierarchy of dependability properties. We start by defining a hierarchy of dependability properties that are the target of our certification scheme. First, dependability attributes that represent the general purpose dependability characteristics of the service under certification are considered. Then, a concrete property enriches the dependability attributes with a set of attributes that refer to the threats the service proves to handle, the mechanisms used to realize the service, or to a specific configuration of the mechanisms that characterizes the service to be certified. Finally, according to the attribute type and its value, a partial or total order relationship is defined.

Service modeling. The complete modeling of the service under certification represents a fundamental step to realize dependability certification, and serves as the basis to carry out the validation process. We present a modeling approach that allows generation of a model of the service as a Symbolic Transition System (STS), based on *i)* the dependability property to be certified and *ii)* the information released by the service provider in the Web Service Description Language (WSDL) interface and/or the Web Service Conversation Language (WSCL) document. The service model succinctly represents the functional and dependability properties of the service, and serves as the starting point to generate a certification model in the form of a discrete-time Markov chain. The certification model is then used to validate whether the service supports a given dependability property with a given level of assurance.

Certification scheme. We define an approach to property validation and certification as a two phase process. The first phase validates the dependability properties of the service before it is actually deployed in the Cloud, and issues a certificate

based on the initial validation results. The second phase monitors the certified properties of the service at runtime, and updates the certificate based on real validation results.

Dependability driven service selection. Our certification scheme provides a solution where services satisfying user's requirements can be searched and selected at runtime based on their dependability certificates. In particular, we extend standard service registries to incorporate the dependability metadata in the form of certificates and to support the service selection process.

Certification of composite services. A user typically implements her applications as a composition of different services, provided by different suppliers. In this thesis, we extend the modeling approach and certification scheme for single services to give a solution for the runtime certification of dependability properties of composite services.

1.2.2 System-Level Dependability Management

We advocate a new dimension where users' applications deployed using IaaS can obtain required dependability properties from a third party. The main contributions of the thesis in the direction of realizing the notion of dependability as a service are as follows.

Dependability management framework. Our approach consists in realizing general dependability mechanisms as independent modules such that each module transparently functions on the users' applications. We enrich each module with a set of metadata that characterizes its dependability properties, and the metadata is used to select mechanisms that satisfy users' requirements. This approach allows users to implement highly available and reliable applications without having to implement low level dependability mechanisms. Building on this approach, we design a framework that encapsulates all the components necessary to offer dependability as a service to the users, and integrates easily with existing Cloud infrastructures.

Analysis of dependability mechanisms. To offer the dependability service effectively, we provide an approach to measure the effectiveness of each dependability module in different configurations. In particular, we estimate the level of reliability and availability that can be obtained using each dependability module by taking into account the failure behavior of the infrastructure components, the correlation between individual failures, and the impact of each failure on the user's applications. This analysis allows us to define a search algorithm that identifies mechanisms satisfying users' requirements.

Resource allocation satisfying dependability constraints. We first categorize and formalize several constraints that users and service providers may want to specify with respect to security, reliability and availability of the service. Such constraints

allow enforcement of deployment conditions inherent to the dependability mechanism selected according to the users' requirements. We then address the satisfaction of such constraints in the overall problem of resource allocation in the Clouds offering IaaS.

Adaptive resource management. Cloud computing environment is highly dynamic and requires runtime monitoring of the delivered service. When system changes affect the desired dependability output for a given users' application (e.g., reduction in the availability due to a server crash), the affect of such changes are masked using algorithms that adapt the current allocation of the application, ensuring delivery of a solution that maintains users' requirements also during runtime.

1.2.3 Secure Application Execution in IaaS Clouds

We present a solution wherein, first, the current vulnerability distribution of the Cloud is considered and users' applications are deployed so as to minimize their exposure to vulnerabilities. Then, network hardening techniques are applied to protect deployed applications from possible cyber-attacks. The following is our contribution in this respect.

Secure application deployment as a task allocation problem. We provide an approach that enables deployment of users' applications in the Cloud infrastructure such that their exposure to network vulnerabilities is minimized. First, we model the secure application deployment problem as a task allocation problem with an application-centric, security-oriented objective, that is subject to dependability constraints. Then, we provide a state-space search algorithm that finds near-optimal allocations in time-efficient manner. We also introduce a heuristic to improve the performance of the search algorithm without significantly degrading the quality of the solutions.

Elastic resource management. In contrast to existing resource management algorithms that must run for at least a given period of time to provide a feasible solution, we present an approach to build elastic algorithms that *i)* provide an initial sub-optimal result rather quickly and their optimality improves as the processing time increases, *ii)* converge to an optimal solution eventually, and *iii)* can be interruptible. Such algorithms are useful in many real-world scenarios, particularly involving mission-critical applications, where the processing time available for resource management algorithms is very limited and often varying. In this direction, we present a resource management scheme that consists of two elastic algorithms, each operating in a different phase with respect to application deployment and execution. First, for a given user request, our elastic task allocation algorithm computes an execution plan for the user's application so as to minimize the application's exposure to existing vulnerabilities. Second, our elastic redeployment algorithm is invoked when the monitoring system detects an anomaly. The redeployment algorithm, instead of computing an allocation from scratch, adapts the application's current allocation so as to

minimize the impact of system changes. It also dynamically adjusts the optimality level of the allocation solution based on the magnitude of system changes.

1.3 Organization of the Thesis

In this chapter, we discussed the motivation and the main objectives of our work, and described the major contributions of this thesis. The remaining chapters are structured as follows.

Chapter 2 discusses the state of the art of fault tolerance and security techniques in the context of Cloud computing, relevant to the objectives of the thesis. It presents the current proposals for improving dependability in Cloud computing, web services and service oriented architectures, and the problem of resource management in IaaS Clouds.

Chapter 3 presents our dependability certification scheme for web services and business processes. This chapter illustrates some basic concepts, our approach to service modeling, and the certification process. It also describes how to integrate our system within existing service-based infrastructures in order to match services satisfying users' requirements.

Chapter 4 illustrates an approach and the conceptual framework that allows provisioning of dependability as a service to users' applications deployed in IaaS Clouds.

Chapter 5 provides a set of techniques involving the analysis of dependability mechanisms and the Cloud infrastructure, search algorithms that allow users to apply desired dependability properties on their applications only by providing high level requirements, and some system management schemes that together realize the notion of dependability as a service.

Chapter 6 formulates the secure application deployment problem as a task allocation problem and presents a state-space search scheme to solve it. This chapter also discusses an approach to transform best-fit search algorithms to elastic algorithms, particularly in the context of application deployment and adaptive resource management in Cloud computing.

Chapter 7 summarizes the contributions of this thesis and outlines future work.

Appendix A reports a list of publications related to the work illustrated in this thesis.

2

Related Work

This chapter discusses some preliminary concepts and state-of-the-art approaches proposed in the literature related to the areas of dependability in Cloud computing. Section 2.1 discusses important features of widely used Cloud computing managers in order to develop an understanding on the underlying system architecture and relevant technical challenges. Section 2.2 then discusses dependability approaches adopted in the context of Cloud computing. In particular, we investigate the failure behavior of various system components, and dependability techniques that exploit virtualization technology for software rejuvenation and management of crash and byzantine faults. Section 2.3 surveys resource management schemes that aim at improving dependability of users' applications. We note that our approach of offering dependability as a service (Chapters 4, 5) and secure application deployment (Chapter 6) is based on the virtualization and resource management techniques. Finally, Section 2.4 summarizes existing solutions that test, evaluate, and certify dependability properties of services.

2.1 Open Source Cloud Computing Solutions

The development of Cloud computing solutions has brought several technical challenges to application developers. These challenges are generally grouped in three main areas: negotiation, decision, and operation. The challenges related to how application developers interface with the Cloud belong to the negotiation area. It also includes description of Cloud offerings, and the definition of the programmability level that a given Cloud solution offers. The decision area copes with the main problem posing realization of any Cloud solution. For example, it concerns with how virtual resources are being scheduled in order to meet users' requirements. Finally, the operation area is associated with the enforcement of decisions and communication between various Cloud elements. In the following, we discuss main characteristics of three most widely used open source Cloud computing managers in order to develop an understanding on the Cloud architectures and corresponding technical challenges.

Eucalyptus. Eucalyptus [45] is the most widely used open source Cloud computing framework that provides resources for experimental instrumentation and study. According to [98], the Eucalyptus project incorporates four characteristics that differentiates it from other Cloud computing solutions: *i)* Eucalyptus is designed to be simple and intuitive, and does not require dedicated resources; *ii)* Eucalyptus is designed to encourage third-party extensions through modular software framework and language-agnostic communication mechanisms; *iii)* The external interface of Eucalyptus is based on the Amazon's API, in particular, Amazon EC2; and *iv)* Eucalyptus provides a virtual network overlay that not only isolates network traffic of different users but also allows clusters to appear to be part of the same local network.

Eucalyptus architecture is hierarchical, containing four high level components, where each component is implemented as a stand-alone web service.

- *Node Controller (NC):* runs on every node that is destined for hosting virtual machine instances. An NC is responsible to query and control the system software (operating system and hypervisor) and for conforming requests from its respective Cluster Controller. The main task of the node controller is to collect essential information, such as node's physical resources (e.g., number of cores and available disk space) and the state of virtual machine instances, and forward this information to its Cluster Controller (CC). NC is also responsible for assisting CC to control virtual machine instances on given a node, verifying authorization, confirming resource availability and executing requests with the hypervisor.

- *Cluster Controller (CC):* generally executes on a cluster front-end machine, or any machine that has network connectivity to two nodes: one running NCs and another running the Cloud Controller (CLC). A CC is responsible to collect/report information about and schedule VM execution on specific NCs and to manage virtual instance network overlay.

- *Storage Controller (Walrus):* is a data storage service that provides mechanisms for storing and accessing virtual machine images and user data. Walrus is based on web services technologies and compatible with Amazon's Simple Storage Service (S3) interface.

- *Cloud Controller (CLC):* is the entry-point for the users into the Cloud. Its main goal is to offer and manage the underlying virtualized resources. CLC is responsible for querying node managers for resources' information, making scheduling decisions, and implementing them by requests to CC. This component is composed by a set of web services which can be grouped into three categories, according their roles: resource services, data services, and interface services.

Nimbus. Nimbus[97] is licensed under the terms of the Apache License to turn clusters into Infrastructure as a Service (IaaS) Cloud, and mainly focuses on scientific applica-

tions. This solution allows users to allocate and configure remote resources by deploying virtual machines, known as Virtual Workspace Service (VWS), where a VWS is the virtual machine manager that different front-ends can invoke. To deploy applications, Nimbus offers a "Cloudkit" configuration that consists of a manager service hosting and an image repository. The workspace components are as follows.

- *Workspace service:* is a web service that supports two front-ends: Amazon EC2 and WSRF. It provides security by means of GSI authentication and authorization.

- *Workspace control:* is responsible for controlling virtual machine instances, managing and reconstructing images, integrating a virtual machine to the network and assigning IP and MAC addresses. The workspace control tools operates with the Xen and KVM hypervisors.

- *Workspace resource management:* manages different virtual machines, but it can be replaced by other technologies such as OpenNebula.

- *Workspace pilot:* provides virtualization with a few changes in the cluster operation, handles signals and administration tools.

OpenNebula. OpenNebula [94] is an open-source toolkit used to build private, public and hybrid Clouds. It has been designed to be integrated with networking and storage solutions and to fit into existing data centers. The OpenNebula architecture is based on three basic technologies that enable provisioning of services on a distributed infrastructure: virtualization, storage and network. All resource allocation is done based on predefined *policies*. The Cumulus Project [135] is an academic proposal based on OpenNebula that intends to provide virtual machines, virtual applications and virtual computing platforms for scientific applications. The Cumulus design is a layered architecture with three main entities: Cumulus front-end, OpenNebula front-end, and OS Farm, and focuses on reaching scalability and autonomy of data centers.

- *Cumulus front-end:* is the access point for a Cumulus system and is responsible for handling virtual machine requirements.

- *OpenNebula front-end:* provides an interface to manage the distributed blade servers and the resources for virtual machines deployment. Cumulus uses Network Information System to administer a common user system and Network File System to manage shared directory. OpenNebula was merged with secure infrastructure solutions, such as Lightweight Directory Access Protocol and Oracle Cluster File System.

- *OS Farm:* is a tool for virtual machine template management that operates to generate and to store Xen VM images and virtual appliances.

2.2 Dependability Approaches in Cloud Computing

Dependability concept usually consists of three parts: the threats affecting dependability, the attributes of dependability, and the means by which dependability is achieved. The threats identify the errors, faults, and failures that may affect a system. The attributes integrate different aspects of dependability and include the basic concepts of availability, reliability, safety, confidentiality, integrity, and maintainability. Other attributes such as security are included by combining various attributes (e.g., confidentiality, integrity and availability). Finally, the means define the categories of mechanisms, such as fault prevention, fault tolerance, and fault removal, that can be used to achieve system dependability.

In general, a failure represents the condition in which the system deviates from fulfilling its intended functionality or the expected behavior. A failure happens due to an error; that is, due to reaching an invalid system state. The hypothesized cause for an error is a fault which represents a fundamental impairment in the system. The notion of faults, errors and failures can be represented using the following chain:

$$\ldots \text{Fault} \rightarrow \text{Error} \rightarrow \text{Failure} \rightarrow \text{Fault} \rightarrow \text{Error} \rightarrow \text{Failure} \ldots$$

Fault tolerance is the ability of the system to perform its function even in the presence of failures, and serves as a means to improve dependability. It is utmost important to clearly understand and define what constitutes the correct system behavior so that specifications on its failure characteristics can be provided and consequently a fault tolerant system be developed. In this section, we first discuss the fault model in Cloud computing environments in order to develop an understanding on the numbers as well as the causes behind recurrent system failures (Section 2.2.1). Then, we discuss state-of-art dependability techniques particularly based on the virtualization technology (Section 2.2.2).

2.2.1 Failure Characteristics of Cloud Environment

Vishwanath et al. [132] and Gill et al. [51] use data mining techniques to describe the failure behavior of server and network components respectively. Their studies are based on the statistical information derived from large-scale data center failure logs. The key findings of their works is summarized below.

Failure behavior of servers. Vishwanath et al. [132] studied the server failure and hardware repair behavior using a large collection of servers (approximately 100,000 servers) and corresponding data on part replacement such as details about server configuration, when a hard disk was issued a ticket for replacement, and when it was actually replaced. Key observations derived from this study are as follows:

- 92% of the machines do not see any repair events but the average number of repairs for the remaining 8% is 2 per machine (20 repair/replacement events contained in 9

machines were identified over a 14 months period). The annual failure rate (AFR) is therefore around 8%.

- For an 8% AFR, repair costs that amount to 2.5 million dollars are approximately spent for 100,000 servers.

- About 78% of total faults/replacements were detected on hard disks, 5% on RAID controllers and 3% due to memory failures. 13% of replacements were due to a collection of components (not particularly dominated by a single component failure). Hard disks are clearly the most failure-prone hardware components and the most significant reason behind server failures.

- About 5% of servers experience a disk failure in less than 1 year from the date when it is commissioned (young servers), 12% when the machines are 1 year old, and 25% of the servers sees hard disk failures when it is 2 years old.

Interestingly, based on the Chi-squared automatic interaction detector methodology, none of the following factors: age of the server, its configuration, location within the rack and workload run on the machine were found to be a significant indicator for failures. Comparison between the number of repairs per machine (RPM) against the number of disks per server in a group of servers (clusters) indicates that *i)* there is a relationship in the failure characteristics of servers that have already experienced a failure, and *ii)* the number of RPM has a correspondence to the total number of disks on that machine. It can be inferred using these statistics that, robust fault tolerance mechanisms must be applied to improve the reliability of hard disks (assuming independent component failures) to substantially reduce the number of failures. Furthermore, to meet the high availability and reliability requirements, applications must reduce utilization of hard disks that have already experienced a failure (since the probability of seeing another failure on that hard disk is higher).

Failure behavior of the network. Similarly to the study on failure behavior of servers, Gill et al. [51] performed a large scale study on the network failures. A link failure happens when the connection between two devices on a specific interface is down and a device failure happens when the device is not routing/forwarding packets correctly (e.g., due to power outage or hardware crash). Key observations derived from this study are as follows:

- Among all the network devices, load balancers (LBs) are least reliable with failure probability of 1 in 5 and Top of Rack switches (ToRs) are most reliable with a failure rate of less than 5%. The root causes for failures in LBs are mainly the software bugs and configuration errors (as opposed to the hardware errors for other devices). Moreover, LBs tend to experience short but frequent failures. This observation indicates that low-cost commodity switches provide sufficient reliability.

- The links forwarding traffic from LBs have highest failure rates; links higher in the topology (e.g., connecting access routers) and links connecting redundant devices have second highest failure rates.

- The estimated median number of packets lost during a failure is 59K and median number of bytes is 25MB (average size of lost packets is 423Bytes). Based on prior measurement studies (that observe packet sizes to be bimodal with modes around 200Bytes and 1,400Bytes), it is estimated that most lost packets belong to the lower part (e.g., ping messages or ACKs).

- Network redundancy reduces the median impact of failures (in terms of number of lost bytes) by only 40%. This observation is against the common belief that network redundancy completely masks failures from applications.

The overall data center network reliability is therefore about 99.99% for 80% of the links and 60% of the devices.

Failure behavior of various system components can also be analyzed based on models defined using fault trees and Markov chains (see Chapter 5). The rationale behind such modeling is twofold: *i)* to capture the user's perspective on component failures, that is, understand the behavior of user's applications that are deployed in the virtual machine instances under system component failures and *ii)* to define the correlation between individual failures and the boundaries on the impact of each failure.

2.2.2 Dependability Techniques in Cloud Computing

In general, the faults can be classified in different ways depending on the nature of the system. Since, in this section, we are interested in typical faults that appear as failures to the end users, we classify the faults into two types similarly to other distributed systems: *i) crash faults* that cause the system components to completely stop functioning or remain inactive during failures (e.g., power outage, hard disk crash), and *ii) byzantine faults* that leads the system components to behave arbitrarily or maliciously during failure, causing the system to behave unpredictably incorrect.

The most widely adopted methods to achieve fault tolerance against crash faults and byzantine faults are as follows:

- *Checking and monitoring:* The system is constantly monitored at runtime to validate, verify and ensure that correct system specifications are being met. This technique, while simple, plays a key role in failure detection and subsequent reconfiguration.

- *Checkpoint and restart:* The system state is captured and saved based on pre-defined parameters (e.g., after every 1024 instructions or every 60 seconds). When the system undergoes a failure, it is restored to the previously known correct state using the latest checkpoint information (instead of restarting the system from start).

- *Replication:* Critical system components are duplicated using additional hardware, software and network resources in such a way that a copy of the critical components is available even after a failure happens. Replication mechanisms are mainly used in two formats: active and passive. In active replication, all the replicas are simultaneously invoked and each replica processes the same request at the same time. This implies that all the replicas have the same system state at any given point of time (unless designed to function in an asynchronous manner) and it can continue to deliver its service even in case of a single replica failure. This method is also called as hot standby. In passive replication, only one processing unit (the primary replica) processes the requests while the backup replicas only save the system state during normal execution periods. Backup replicas take over the execution process only when the primary replica fails. This method is called as cold standby. The $N + M$ technique of adding M standby hosts to spares for N working hosts to accommodate up to M failures is the most popular solution.

In the following, we discuss dependability techniques that exploit attributes of the virtualization technology to handle system failures (e.g., [29, 78, 116, 123, 122]). We discuss this aspect in detail because our solution to offer dependability as a service uses virtualization to transparently operate on the user's applications (see Chapter 4). In particular, we discuss three lines of research: virtualization-based software rejuvenation, dependability against crash faults, and dependability against byzantine faults.

Software rejuvenation. Software rejuvenation is a proactive fault management technique that aims at cleaning the system's internal state so as to prevent the occurrence of severe failures due to the phenomena of software aging or transient failures [125]. A number of solutions have applied software rejuvenation techniques to Cloud Computing and virtualization. Thein et al. [123] propose a technique that can increase the availability of application servers using virtualization, clustering, and software rejuvenation. Their solution uses analytical models to analyze multiple design choices when a single physical server and dual physical servers are used to host multiple virtual machines. The results of their study demonstrate that integration of virtualization, clustering, and software rejuvenation increases availability, manageability and savings from server consolidation, without significantly decreasing the uptime of critical services. Moura Silva et al. [116] propose a similar approach based on automated self-healing techniques that claims to induce zero downtime for most of the cases. In their solution, software aging and transient failures are detected through continuous monitoring of system data and performability metrics of the application server. In [125], Trivedi et al. are propose stochastic models that help to detect software aging and determine optimal times to perform rejuvenation. Models are constructed using workload and resource usage data collected from the UNIX operating system over a period of time. Their measurement-based models help in development of strategies for software rejuvenation triggered by actual measurements.

Thein et al. [122] propose another rejuvenation technique for application servers using stochastic models and the virtualization technology. In particular, the authors present stochastic modeling of a single physical server, that is used to host multiple virtual machines, configured with the specified technique. Their modeling scheme is intended as a general model which captures various characteristics of the application server, failure behavior, and performability measures. Kourai et al. [78] present a technique called warm-VM reboot for fast rejuvenation of VMMs that enables efficiently rebooting only a VMM by suspending and resuming VMs without accessing the memory images. This technique is based on two mechanisms, on-memory suspend/resume of VMs and quick reload of VMMs. Warm-VM reboot technique reduces downtime and prevents the performance degradation due to cache misses after the reboot.

Dependability against crash faults. A number of solutions that improve the dependability of users' applications using virtualization are based on checkpoint and restart, monitoring, and dynamic replication schemes [103, 49, 133, 46, 29, 119, 92]. For instance, the Google File System [49] creates new file "chunks" when the number of available copies is below a specified threshold, and commercial tools such as VMWare High Availability [133] allows a virtual machine on a failed host to be reinstantiated on a new machine. Nagarajan et al. [92] present a proactive fault tolerance technique for Message Passing Interface (MPI) applications by exploiting the Xen hypervisor's live migration mechanism. In particular, their approach consists in migration of MPI tasks from a health-deteriorating node to a healthy one without stopping the MPI task during most of the migration. Experimental results demonstrate that live migration hides migration costs and limits the overhead to a few seconds. Mishra et al. [89] provide a high-level approach for autonomic management of system availability. Their approach involves real-time evaluation monitoring and management, and the monitoring data collected during operation is used to populate a set of analytical models. However, their approach focuses on static system architectures and assumes that the underlying availability models are built manually at system design time. Finally, Freiling et al. [46] introduce an extensible grammar that classifies the states and transitions of VM images. The grammar can also be used to create rules for recovery and high availability, by exploiting virtualization for simplified fault tolerance.

Similarly to the above solutions, the approach of leveraging virtualization to handle system failures and to improve the dependability of users' applications is central to the work presented in this thesis. In particular, our work is based on the schemes that can be applied independently to the application and underlying hardware, offering high levels of transparency and generality. Here, a detailed description of a technique (called Remus [29]) that combines checkpointing, live migration, and replication schemes to tolerate crash faults in the Cloud is discussed as an example.

In Remus, the system or user application that must be protected from failures is first encapsulated in a VM (say active VM or the primary), and the following operations are performed at the VM level to obtain paired servers that run in active-passive configuration.

1. Checkpoint the changed memory state at the primary and continue to next epoch of network and disk request streams.

2. Replicate system state on the backup.

3. Send checkpoint acknowledgement from the backup when complete memory checkpoint and corresponding disk requests have been received.

4. Release outbound network packets queued during the previous epoch upon receiving the acknowledgement.

Remus achieves high-availability by frequently checkpointing and transmitting the state of the active VM on to a backup physical host. The VM image on the backup is resident in the memory and may begin execution immediately after a failure in the active VM is detected. The backup only acts like a receptor since the VM in the backup host is not actually executed during fail-free periods. This allows the backup to concurrently receive checkpoints from VMs running on multiple physical hosts (in an N-to-1 style configuration) providing a higher degree of freedom in balancing resource costs due to redundancy. In addition to generality and transparency, seamless failure recovery can be achieved i.e., no externally visible state is lost in case of a single host failure and recovery happens rapidly enough that it appears only like a temporary packet loss. Since the backup is only periodically consistent with the primary replica using the checkpoint-transmission procedure, all network output is buffered until a consistent image of the host is received by the backup, and the buffer is released only when the backup is completely synchronized with the primary. Unlike network traffic, the disk state is not externally visible but it has to be transmitted to the backup as part of a complete cycle. To address this, Remus asynchronously sends the disk state to the backup where it is initially buffered in the RAM. When the corresponding memory state is received, complete checkpoint is acknowledged, output is made visible to the user, and buffered disk state is applied to the backup disk.

Remus is built on Xen hypervisor's live migration machinery [25]. Live migration is a technique using which a complete VM can be relocated onto another physical host in the network (typically a LAN) with a minor interruption to the VM. While Remus provides an efficient replication mechanism, it employs a simple heartbeat based failure detection technique that is directly integrated within the checkpoint stream. Experiments shown that the protocol is practically deployable but not well suited for applications that are very sensitive to network latencies.

Dependability against byzantine faults. Byzantine Fault Tolerance (BFT) protocols are powerful approaches to obtain highly reliable and available systems. Zhang et al. [140] propose the following system architecture. A request made to the Cloud system will consist of a flow through which the request will be served and returned to the requester. This flow

consists of primary selection, replica selection, request execution, primary updating, and replica updating, operations. The primary selection is a selection of the primary node or a resource to which the initial request will go to. This depends on the QoS requirements of the request which in turn depends on how critical the request is. After this, the replica selection is performed where, up to $3f+1$ replicas are selected by the primary from a pool which is sorted with the best QoS rating. Next, the requests are executed by the primary and all the replicas and the results are sent to the primary, who performs an analysis on the result and decide whether to perform primary or replica update.

Wood et al. [137] identify that despite numerous efforts, most BFT systems are too expensive for practical use (no commercial data centers have employed BFT techniques), and argue that the dominant costs are due to the hardware performing service execution and not due to running the agreement protocol. For instance, a toy application running null requests with the Zyzzyva BFT approach [77] exhibits a peak throughput of 80K requests/second while a database service running the same protocol on comparable hardware exhibits almost three times lower throughput. Based on this observation, ZZ, an execution approach that can be integrated with existing BFT state machine replication (SMR) and agreement protocols is presented in [137]. The prototype of ZZ is built on the BASE implementation [26] and guarantees BFT while significantly reducing resource consumption costs during fail-free periods.

The design of ZZ is based on the virtualization technology and targeted to tolerate byzantine faults while reducing the resource provisioning costs incurred by BFT protocols during fail-free periods. The cost reduction benefits of ZZ can be obtained only when BFT is used in the data center running multiple applications so that sleeping replicas can be distributed across the pool of servers and higher peak throughput can be achieved when execution dominates the request processing cost and resources are constrained. These assumptions make ZZ a suitable scheme to be applied in a Cloud computing environment.

The BFT execution protocol reduces the replication cost from $2f+1$ to $f+1$ based on the following principle:

- A system designed to function correctly in an asynchronous environment will provide correct results even if some of the replicas are outdated.
- A system designed to function correctly in the presence of f byzantine faults will, during fault-free period, remain unaffected even if up to f replicas are turned off.

The second observation is used to commission only $f+1$ replica to actively execute requests. The system is in a correct state if the response obtained from all $f+1$ replica is the same. In case of a failure (i.e., when responses do not match), the first observation is used to continue system operation as if the f standby replicas were slow but correct replicas. To correctly realize this design, the system requires an agile replica wake-up mechanism. To achieve this, the system exploits virtualization technology by maintaining additional replicas (VMs) in a "dormant" state, which are either pre-spawned but paused VMs or

the VM that is hibernated to a disk. There is a trade-off in adopting either method. Pre-spawned VM can resume execution in very short span (in the order of few milliseconds) but consumes memory higher resources, whereas, VMs hibernated to disks incur greater recovery times but occupy only storage space.

2.3 Dependability-Oriented Resource Management Schemes

Traditionally, to ensure resource availability, QoS and dependability to hundreds of applications in the Cloud under fluctuating workloads, server failures, and network congestion, dedicated servers were allocated to applications and server capacity was often over-provisioned. However, the use of dedicated hardware not only lead to poor energy usage, but also made it difficult to react to system changes. Furthermore, the growing number of under-utilized servers increased operating costs such as system management, energy consumption of servers, and network and cooling infrastructure costs.

In the last decade, the virtualization technology has emerged as a very effective approach to address these issues by de-coupling physical servers from the resources needed by applications. In particular, virtualization provides an efficient way to insulate and partition server's resources so that only a portion of them can be utilized by an application. It also provides a greater flexibility and control over resource management, allowing for dynamic adjustment of CPU and memory usage, and live migration of virtual machines among physical servers (e.g., [54]). Most Cloud computing solutions create virtualized environments for application execution on distributed data centers.

Deploying virtualized services in Cloud computing create new resource management problems, such as optimal placement of virtual machines. Existing solutions focus mainly on placing and adaptive managing virtual machines in order to *i)* reduce energy consumption costs and maximize profits for the data center owner, and *ii)* improve the performance and QoS of applications. Only a few approaches are available in the literature to improve dependability of applications. In this section, we investigate resource management schemes that improve dependability and performance of applications in Cloud data centers.

Resource management for Cloud computing is often modeled as a placement problem in which virtual machines are allocated on the data center's hosts [54], having been the applications mapped on the appropriate virtual machine templates available in the data center environment. Existing solutions, particularly for services that provision on-demand resources to users, primarily focus on making virtual machine placement decisions at two distinct levels: *i)* initial virtual machine allocation and *ii)* runtime adaption of current virtual machine allocation [88]. Based on user's requirements and failure characteristics of the envisioned data center, a set of dependability and performance constraints are specified (e.g., constraint specifying that replicas of the user's application be allocated on two different physical hosts to avoid single points of failure). The initial resource

allocation process identifies the physical hosts on which the requested virtual machines can be allocated such that all the placement constraints are satisfied. Once the required virtual machines are created and delivered to the user, the runtime adaption process monitors the system and resizes virtual machines or migrates them to other physical hosts in order to meet the predefined goals (e.g., energy conservation), while satisfying the placement constraints. While the objective of most resource management algorithms in this context has been to maximize the service provider's goals (e.g., through resource consolidation, load balancing, satisfaction of SLAs), we will provide a broader perspective which encompasses both the provider's and the users' views, balancing all needs in a comprehensive way.

In this section, first, we will study resource management schemes for placing applications in the Cloud computing environment at the initial deployment (see Section 2.3.1). Then, we will discuss dynamic adaptation of the applications placement to deal with changing working status of the architecture components, balancing dependability and performance of users' applications (see Section 2.3.2).

2.3.1 Initial Allocation of Virtual Machines

Given the NP-Hardness of the allocation problem, existing solutions either use Constraints Programming (CP) solvers (e.g., [22]) or design heuristics (e.g., [114]) to obtain the placement solutions. In general, while the goal is to maximize the goals of the data center owner, each solution takes a different approach in modeling the context of the system and, consequently, defines different objective functions and placement constraints. In the following, we discuss four representative solutions to understand different dimensions in which the overall problem has been studied.

Bin et al. [22] combine the Hardware Predicted Failure Analysis alerts (HwPFA) and live migration techniques to provide a high availability solution. On predicting hardware failure alerts, a trigger to the cluster management system is provided so as to move the virtual machines from the failing host to other working hosts. Depending on the allowed response time, either a complete live relocation of the virtual machine is performed so that continuous operation of the applications is ensured, or a cold relocation is performed by starting a new virtual machine on a working host with a small interruption. The goal of their solution is to provide k-resiliency to users' applications while reducing the resource consumption costs. We note that k-resiliency allows a given application or virtual machine to tolerate up to k host failures. In general, to ensure k-resiliency, a feasible solution should dedicate at least k hosts for the given virtual machine (in addition to the virtual machine itself). The proposed approach introduces the notion of shadow virtual machines that denotes the location or host where a virtual machine can be evacuated (i.e., a shadow serves as a placeholder) and aim to construct shadow placement constraints so to reduce the overall resource requirements to a value less than $(k + 1)$. To achieve this, they transform the placement problem with k-resiliency constraints into a constraint

satisfaction problem including the notion of shadow virtual machines, and solve it using a constraint programming engine. All the shadows of a given virtual machine and the virtual machine itself are anti-colocated. In addition, they employ a scheme of numbering shadows and failures in a way that identifies the possible overlaps of actual virtual machine evacuations. In particular, each failing host is assigned a unique index (1 through k) and each shadow of a virtual machine is assigned a unique index. Upon failure of a host indexed i, the virtual machines on that host are evacuated to the location of their i-th shadow. The placement constraints are defined to specifically numbered shadows and virtual machines that may overlap following host failures, thus reducing the number of backup hosts required. For example, virtual machines that are placed on different hosts cannot be evacuated together to shadows with the same index (as each host would be assigned a different failure index); therefore, their shadows with same index can overlap.

Machida et al. [83] consider consolidated server systems and present a method to redundant configuration of virtual machines, in anticipation of host server failures for online applications. They estimate the requisite minimum number of virtual machines according to the performance requirements of the given application, and compute the virtual machine placement solution so that the configuration can survive k host server failures. The overall problem is defined as a combinatorial optimization problem and a greedy algorithm for determining the placement solution is provided with the aim of minimizing the number of required hosting servers. Their method performs better than the conventional $N+M$ redundant configuration in terms of the number of hosting servers required.

Shi et al. [114] formulate the problem of virtual machine placement as an Integer Linear Programming (ILP) problem and provide a twofold solution. First, they use solvers to obtain optimal results. Second, since the scalability of this approach is limited, they also provide a modified version of the first fit decreasing heuristic to generate sub-optimal results. In particular, they classify the requests for virtual machine placements into different categories and satisfy the following constraints using the first fit decreasing heuristic, in the form of a multidimensional vector packing problem: *i)* the full deployment constraint that ensures either all the virtual machines requested by the user are allocated or none; *ii)* the anti-colocation constraint requiring all the virtual machines to be placed on different physical hosts; and *iii)* the security constraint requiring a physical host only be assigned virtual machines from the same user request and not be assigned any virtual machines from other requests.

The three aforementioned resource allocation techniques consider only fault tolerance and security constraints. The solution by Jayasinghe et al. [60] that also take into account various performance attributes while performing initial allocation of virtual machines. In particular, they propose a structural constraints-aware virtual machine placement approach to improve the performance and availability of applications deployed in the data centers. They integrate the structural information of users' applications within the algorithm for initial placement of virtual machines by means of three constraints: *i)* demand

constraint, that defines the lower bound of resource allocations that each virtual machine requires from the service to meet its SLA; *ii)* availability constraint, that improves the overall availability of given applications using a combination of anti-collocation/collocation constraints; and *iii)* communication constraint, that represents the communication requirement between two virtual machines. The objective of the proposed algorithm is to minimize the communication cost while satisfying both the demand and availability constraints. Their solution uses the divide-and-conquer technique which involves the following steps: *i)* the group of virtual machines requested by the user is divided into a set of smaller virtual machine groups and the upper bound of the virtual machine group size is determined by the average capacity of a server rack; *ii)* a suitable server rack is identified for each virtual machine group such that the mapping minimizes the total communication cost and guarantees the satisfaction of availability constraints; *iii)* a physical host satisfying the demand constraint is identified for each virtual machine.

2.3.2 Runtime Adaption of Virtual Machine Allocation

Cloud computing is highly dynamic in terms of task activation, bandwidth availability, component failures and recovery. This implies that static deployment strategies for virtual machines that perform only initial allocation may not provide satisfactory results at runtime and application's dependability and performance requirements may not be satisfied all along their lives. A naive approach is to re-compute the allocation from scratch each time system changes affect an application. However, since this method may not scale well during runtime, a number of solutions have been proposed in the literature to adapt the current allocation of applications using fewer actions. In the following, we present state-of-art adaptive resource management schemes.

The placement of application replicas to achieve dependability becomes especially challenging when they consist of communicating components (e.g., multi-tier web applications). Recent works on performance optimization of such applications (e.g., [30], [70]) address the performance impact of resource allocation, but does not combine performance modeling with availability requirements and dynamic regeneration of failed components.

The trade-off between availability and performance is considered in the literature on dependability since increasing availability (by using more redundancy) typically increases response time. In fact, the well-known Brewer's theorem states that consistency, availability, and partition tolerance are the three commonly desired properties by a distributed system, but it is impossible to achieve all three [50]. Examples of work that explicitly address this issue include [115, 74, 65, 70]. Among these solutions, [115] considers the problem of when to invoke a (human) repair process to optimize various metrics of cost and availability defined on the system. The optimal policies that specify when the repair should be invoked (as a function of system state) are computed off-line via Markov decision process models of the system. Similarly, Jung et. al [70] study how virtualization can be used to provide enhanced solutions to the classic problem of ensuring high availabil-

ity while maintaining performance of multi-tier web services. Software components are restored whenever failures occur and component placement is managed using information about application control flow and performance predictions.

Addis et al. [1] devise a resource allocation policy for virtualized Cloud computing environments aiming at identifying performance and energy trade-off, with a priori availability guarantees for the users. They model the problem as a mixed integer non-linear programming problem and propose a heuristic solution based on non-linear programming and local-search techniques. In particular, the availability requirements are introduced as constraints, and the objective is to determine a resource allocation that allows improving the total profit (the difference between the revenues from SLA contracts and the total costs). Their solution defines the following four actions to take into account availability constraints: *i)* increase/decrease the working frequency for each server according to the application loads (a frequency change compatible with the new application loads does not affect availability); *ii)* switch a server to low power sleep state and allocate all applications of that server on the other available servers (some applications from the overloaded servers can be duplicated and allocated on another active server to satisfy availability constraints); *iii)* reallocation of class-tier applications on a server with sufficient availability assurance to satisfy the performance constraints; *iv)* servers exchange is used to move all applications from a server which should be switched to low the power sleep state to a different active server if the availability of the new server guarantees the availability requests of all the moved applications.

Zheng et al. [143] consider the lack of maintenance to be the root cause for downtime events in a Cloud computing data center. To address this issue, they first present a heuristics for resource provisioning under a given maintenance schedule and, then, build on the heuristics to solve the joint resource provisioning and maintenance scheduling problem. Yang et al. [139] propose an algorithm that uses Markov chain models to schedule tasks so that they get the best value of utility. A job being executed on the Cloud possesses the following factors: deadline, data, and reward factor for completing it on time. The reward is assigned to each task depending on the time it takes to complete the job with respect to its deadline. The earlier it completes, the higher is the reward. If a task fails during execution, then the time required for the task to recover is also added to the total time it takes to complete. The proposed algorithm includes reliability while calculating the value of the reward for each task. The impact of the amount of time which is spent in recovery from a failure and the useless waiting time in the task queue are added in this model. The proposed rules are: execute a job as soon as possible when the resources are available, and pausing a job in the queue till a resource is available or assigning a new task to free resources. This approach has been tested against well-known task scheduling algorithms: if there are not system failures results have quality similar to other conventional scheduling techniques, while in the case of system failures the proposed algorithm produces better efficiency and stability.

2.4 Web Services Dependability Evaluation

Our certification solution has multi-disciplinary roots involving areas of SOA, system modeling, testing, and certification. In this section, we discuss some related work corresponding to these areas.

An important line of research concerns definition of modeling approaches for automatic generation of test cases and, validation of functional and QoS parameters for services and service compositions. Salva and Rabhi [108] propose an approach to test the robustness of a service by automatically generating test cases from its WSDL interface. Frantzen et al. [48] define an approach to model service coordination using an STS and automatically generate run-time test cases. Salva et al. [107] propose a testing method based on STSs and security rules for stateful web services. They first define security rules using the Nomad language, then translate rules into an STS, and finally generate test cases from STS to check whether security rules are satisfied. Ding et al. [42] model failure behavior using non-homogeneous Poisson process and compute the overall system reliability through the reliability of partner link, port type, and operation. Riccobene et al. [105] use architecture- and path-based reliability models to predict the reliability of an SCA-ASM component model, and of the SCA assembly modeling a service orchestration, by considering failures specific to the nature of ASMs. Bentakouk et al. [21] propose a testing solution that uses an SMT solver to check the conformance of a composite service against its specifications. In contrast to the above works, we use formal modeling to validate and certify dependability properties of services. Mateescu and Rampacek [85] define an approach for modeling business processes and web services described in BPEL using process algebraic rules. Betakauk et al. [20] propose a framework for testing service orchestrations. The orchestration specification is translated into an STS, and then, a Symbolic Execution Tree (SET) is computed. SET supports retrieval of STS execution semantics and, for a given coverage criterion, generation of a set of execution paths which are run by a test oracle against the orchestration implementation. Pathak et al. [100] define a framework that models service compositions as STSs starting from UML state machines. Similarly to the above approaches, we model services as state automata (using STSs) and apply a derivative approach to generate the models for service compositions. However, our solution generates dependability certificates for service compositions using certificates of individual services.

Definition of service certification schemes is the most relevant aspect to our work. Kourtesis et al. [79] present a solution using Stream X-machines to improve the reliability of SOA environments; they evaluate if the service is functionally equivalent to its specifications, and award a certificate to it. Anisetti et al. [10, 8] proposes a model-driven test-based security certification scheme for (composite) services. This solution allows clients to select services on the basis of their security preferences [9]. The work in [23] presents a test-based reliability certification scheme for services that provides an a priori validation of services based on reliability patterns, and a posteriori testing using a set of metrics. Our

work, instead, quantitatively evaluates dependability properties of (composite) services and supports run-time validation of certificates.

The approach of probabilistically estimating the reliability of a software using Markov chains has been studied in the past. Mustafiz et al. [91] propose an approach to identify reliable ways of designing the interactions in a system and assigning probability values to each interaction measuring its success. Cheung [27] claims that reliability of a software depends on the reliability of its components and the probabilistic distribution of their utilization. A Markov model is then used to measure the reliability and effects of failures on the system with respect to a user environment. Other Markovian model-based approaches to evaluate system reliability have been proposed (e.g., [57, 90]). In contrast to Markov models, Petri nets are often employed to verify the feasibility and correctness of business process configurations (e.g., [129]). Although these models are effective in establishing functional requirements of service compositions, they seemingly overkill our goals of validating and monitoring dependability properties of individual services and add complexity in determining assurance level values.

2.5 Chapter Summary

In this chapter, we focused on solutions known in the literature for solving problems related to dependability in Cloud computing. We discussed the architecture and important features of widely used Cloud computing managers and investigated failure characteristics of various system components. Based on this analysis, we surveyed dependability approaches that leverage the virtualization technology and resource management schemes that perform initial allocation and runtime adaption of users' applications. Finally, we provided related work in the area of dependability certification of services.

In the following of this thesis, we will analyze more in depth the aspect of dependability in SOA and Cloud computing, proposing a new approach to dependability certification of services, and fault tolerance and security management in Cloud infrastructures.

3

Dependability Certification of Services

Trustworthiness of services become a critical factor for the users when applications are implemented by composing a set of Cloud computing services. In this context, there is a need for awareness tools that report the Quality of Service (QoS) offered by a given service, and assurance techniques that increase users' confidence that a given service complies with their dependability requirements.

This chapter defines a certification scheme that allows users to verify the dependability properties of services and business processes. The certification scheme relies on discrete-time Markov chains and awards machine-readable dependability certificates to services, whose validity is then continuously verified using runtime monitoring. This solution also extends the traditional discovery and selection process with dependability requirements and certificates in order to support a dependability-aware application composition. In other words, our certification scheme provides the basic support necessary for the users in building dependable applications by selecting and integrating only those external services that satisfy their dependability requirements. We note that the certification scheme is a passive method that is applied on a given Cloud computing service in its present form, and denotes the first level of dependability support offered in this thesis.

3.1 Introduction

The availability of a range of services published by different providers, coupled with standard XML-based protocols, allows for the design and dynamic composition of applications and business processes across organization boundaries [99]. Traditionally, web services provided only high-level software functionality such as travel reservation, and shipment and accounts management. With the advent of Cloud computing, services are now being offered by following the SaaS and IaaS models, that provide both high-level and low-level functionalities to the users, thus increasing the opportunities to build the overall system in a service-oriented manner.

In general, service providers and users negotiate a set of guarantees on the performance and availability of the service, in the form of SLAs. However, the violation of SLAs has become a norm rather than an exception and, as a result, users are increasingly concerned about service failures that may affect the functional and non-functional properties of their applications. In this context, trustworthiness of services is a critical factor for the users, and raises the need for the definition of an assurance technique that increases users' confidence that a given service complies with their dependability requirements.

A suitable technique to address the above concerns is based on certification [31]. The certification schemes originally targeted static and monolithic software, and produced human-readable certificates to be used at deployment and installation time [128, 55]. However, Cloud computing and SOA paradigms require the definition of certification schemes that can address the issues introduced by a service-based ecosystem and its dynamics. In particular, there is a need for the definition of approaches that support machine-readable certificates that can be integrated within the service discovery and selection process. The fundamental requirement for such certification scheme is the ability to verify dependability properties of services and their compositions with a given level of assurance, and prove service robustness against a set of failures to the users.

This chapter defines a certification scheme that, starting from a model of the service as a Symbolic Transition System (STS), generates a certification model in the form of a discrete-time Markov chain. The certification model is used to validate whether the service supports a given dependability property with a given level of assurance. The result of property validation and certification is a machine-readable certificate that represents the reasons why the service supports a dependability property and serves as a proof to the users that appropriate dependability mechanisms have been used while building it. To complement the dynamic nature of service-based infrastructures, the certificate validity is continuously verified using runtime monitoring, making the certificate usable both at discovery and run time. The certification scheme discussed here allows users to select services with a set of certified dependability properties, and supports dependability certification of composite services.

Given the goal of this thesis is to improve the dependability of Cloud computing, particularly by reducing the risks of using IaaS and SaaS services, our dependability certification scheme provides the first level of the overall solution where an assertion of dependability and QoS is performed. It allows users to implement dependable applications and allows providers to ensure negotiated dependability properties of their services at runtime, thus building trustworthy relationship between users and service providers.

3.1.1 Chapter Outline

This chapter is organized as follows. Section 3.2 describes a reference scenario and some basic concepts on dependability certification. Section 3.3 illustrates an approach to STS-based service modeling. Section 3.4 presents the certification model based on discrete-

time Markov chains and defines the concept of assurance level. Section 3.5 describes the two-phase dependability certification process for single services. Section 3.6 discusses how to integrate the proposed certification process within service-based infrastructures to effectively match service certificates with user's requirements. Section 3.7 presents how the certification process can be extended to certify composite services. Finally, Section 3.8 provides concluding remarks.

3.2 Reference Scenario and Basic Concepts

This section describes a reference scenario and some basic concepts on dependability certification.

3.2.1 Reference Scenario

Consider a highly dynamic and distributed service-based infrastructure that involves the following main parties: *i)* a trusted *certification authority* that certifies the dependability properties of services; *ii)* a *service provider* that implements a service and engages with the certification authority to obtain a certificate for the service; *iii)* a *client* (business owner) that establishes a business relationship with one or more service providers and uses a set of certified services to implement its business process; *iv)* a *service discovery* that enhances a registry of published services to support the certification process and metadata. We note that the client can also be a service consumer searching for and selecting a single certified service. The term client refers to both the users as well as servers.

As an example, consider an online shopping service (*eShop*) that allows its customers to browse and compare available items, purchase goods, and make shipping orders over the Internet. The eShop business owner is the client of the certification framework, who implements eShop as a business process using *i)* three vendor services that offer a range of goods for trade to eShop, *ii)* two shipment services that deliver items to a customer address, and *iii)* an external storage service to store, retrieve, and update shopping information. The vendor and shipping services belong to SaaS layer and the external storage is an IaaS service. Table 3.1 summarizes the details about the operations of partner services.

When a query to browse available items is provided to eShop, a call to operation `browseItems` of all three vendor services is made. The result from each vendor is reported to the customer, say in a tabular form, to enable comparison. The customer can then choose to purchase an item from a specific vendor. In this case, first a call to operation `buyGoods` of that vendor service is made, and then operation `shipItems` of the shipment service with minimum freight cost is invoked. For each transaction, eShop stores the customer, vendor, and shipping specific data using operation `write` of the storage service. Shopping information is then accessed using operation `read` whenever necessary. Operations `write` and `read` are invoked after a successful login.

Table 3.1: Summary of operations realized by eShop partner services

Service	Operation	Description
Vendor	`browseItems`(*query*)	Allows customers to browse available items
	`buyGoods`(*itemID,data*)	Allows customers to buy an item
Shipping	`shipItems`(*itemID,addr*)	Allows customers to ship an item to an address
Storage	`login`(*usr,pwd*)	Provides password-based authentication and returns an authentication token
	`write`(*data,token*)	Stores data in the remote server
	`read`(*query,token*)	Provides access to data stored in the server
	`logout`(*token*)	Allows customers to log out

Intuitively, failures in the partner services may have an impact on eShop and, therefore, in addition to the functional properties, dependability properties of partner services become of paramount importance to eShop. For example, a failure in one of the vendor services may result in quality degradation of eShop, while a failure in the storage service may affect its overall reliability and availability. Hence, eShop must integrate only those external services that satisfy its dependability requirements. In this context, a dependability certificate can serve as an effective means of assurance to the eShop business owner, by providing a proof that its partner services support a given set of dependability properties. A service discovery that provides a selection approach based on dependability certificates can further serve as a means to search and integrate appropriate partner services. For simplicity, whenever not strictly required, we will use a simplified version of the motivating scenario and discuss the concepts in this chapter using the storage service.

3.2.2 Basic Concepts

A service provider implements its service using a set of dependability mechanisms, and engages with the certification authority in a process that certifies the dependability properties of the service. To realize this process, the certification authority must define: *i)* a hierarchy of dependability properties to be certified; *ii)* a model of the service to drive the certification process; and *iii)* a policy to assess and prove that a given property holds for the service.

Hierarchy of dependability properties

According to [17, 126], dependability concept usually consists of three parts: the *threats* affecting dependability, the *attributes* of dependability, and the *means* by which dependability is achieved. The threats identify the errors, faults, and failures that may affect a system. The attributes integrate different aspects of dependability and include the basic concepts of availability, reliability, safety, confidentiality, integrity, and maintainability. This chapter considers a subset of dependability attributes, that is, availability, reliabil-

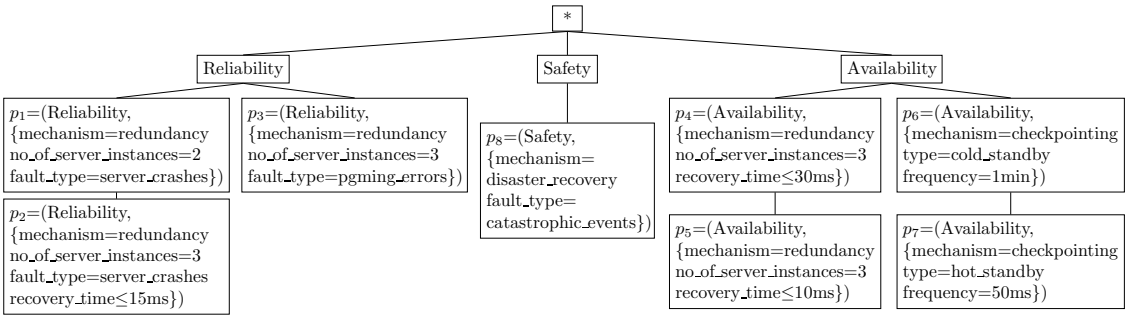


Figure 3.1: An example of a hierarchy of dependability properties

ity, and safety, which measure the ability of the service to be up and running, and to be resistant to failures. Finally, the means define the categories of mechanisms, such as fault prevention, fault tolerance, and fault removal, that can be used to achieve system dependability.

Starting from the above definition of dependability, the definition of a hierarchy of dependability properties that are the target of the certification scheme is provided. First, the dependability attributes (abstract properties below) are considered to represent the general purpose dependability characteristics of the service under certification. Then, a concrete property $p{=}(\hat{p}, A)$ enriches the abstract property $p.\hat{p}$ with a set $p.A$ of attributes that refer to the threats the service proves to handle, the mechanisms used to realize the service, or to a specific configuration of the mechanisms that characterizes the service to be certified. We note that the mechanisms represent specific implementations of dependability means. For each attribute $attr{\in}A$, according to its type, a partial or total order relationship $\preceq_{attr}$ can be defined on its domain Ω_{attr}, and $v(attr)$ represents the value of $attr$. If an attribute does not contribute to a property configuration, its value is not specified. In general, attributes represent a service provider's claims on the dependability of its service. For instance, when $attr$ is fault_type, $v(attr)$ can be *crash failure*, *programming error*, or *byzantine failure*.

The hierarchical ordering of dependability properties can be defined by a pair $(\mathcal{P}, \preceq_P)$, where $\mathcal{P}$ is the set of all concrete properties, and $\preceq_P$ is a partial order relationship over $\mathcal{P}$. We note that an abstract property corresponds to a concrete property with no attributes specified. Given two properties $p_i{,}p_j{\in}\mathcal{P}$, $p_i{\preceq_P}p_j$ if *i)* $p_i.\hat{p}{=}p_j.\hat{p}$ and *ii)* $\forall attr{\in}A$ either $v_i(attr)$ is not specified for p_i or $v_i(attr){\preceq_{attr}}v_j(attr)$. The relation $p_i{\preceq_P}p_j$ means that p_i is weaker than p_j and a service certified for p_j also holds p_i. Figure 3.1 shows an example of a hierarchy of dependability properties. Each node represents a concrete property $p{=}(\hat{p}{,}A)$. Each child node of a given node represents a stronger property and takes precedence in the hierarchy. For instance, $p_1{\preceq_P}p_2$, $p_4{\preceq_P}p_5$, and $p_6{\preceq_P}p_7$. $p_1{\preceq_P}p_2$ since p_2 specifies additional guarantees on the recovery time). We note that some properties are

incomparable despite the same abstract property (e.g., p_4 and p_6).

Symbolic Transition System (STS)

The service model must succinctly represent the functional and dependability properties of the service. To this aim, it must represent the different states of the service, the dependability mechanisms, and their specific configurations. Following the approaches in [48, 73, 124], services, interactions within a service, and communications between different services are modeled using STSs. An STS extends a finite state automaton with variables, actions, and guards to capture the complex interactions in a system. It can be defined as follows.

Definition 3.2.1 (Symbolic Transition System)**.** *A symbolic transition system is a 6-tuple* $\langle \mathcal{S}, s_1, \mathcal{V}, \mathcal{I}, \mathcal{A}, \overset{a,g,u}{\rightarrow} \rangle$*, where* $\mathcal{S}$ *is the set of states in the service,* $s_1 \in \mathcal{S}$ *is the initial state,* $\mathcal{V}$ *is the set of (location) internal variables specifying data dependent flow,* $\mathcal{I}$ *is the set of interaction variables representing operation inputs and outputs,* $\mathcal{A}$ *is the set of actions, and* $\overset{a,g,u}{\rightarrow}$ *is the transition relation. Each transition* $(s_i, s_j) \in \overset{a,g,u}{\rightarrow}$ *between two states* $s_i, s_j \in \mathcal{S}$ *is associated with an action* $a \in \mathcal{A}$ *that encapsulates a guard* g*, defining the conditions on transition, and an update mapping* u*, providing new assignments to the variables in* $\mathcal{V}$*.*

In the following, when possible, the transition relation is simply referred with $\rightarrow$. Differently from [48, 73, 124], the modeling approach discussed here can also be used when the real implementation of the service (and its dependability mechanisms) is available. This approach permits to generate fine-grained test cases that can be used to generate the certification model of the certification solution, and validate the dependability properties against various threats (see Sections 3.4, 3.5, and 3.7).

Policy

A certification scheme must verify and prove that a dependability property p is supported by the service. Proving p is equivalent to validating the implementation of dependability mechanisms used by the service to counteract a given (set of) threat. Based on this observation, a policy $Pol(p) \rightarrow \{c_1, \ldots, c_m\}$ contains all conditions $c_1, \ldots, c_m$ on dependability mechanisms necessary to prove that p holds for the service. We note that while the threats specified in the property drive the certification and testing processes (e.g., by defining a given fault injection model), they are not considered in policy specification. This is due to the fact that policies only include conditions that can be quantitatively measured to validate a given mechanism. Hence, a policy can be defined corresponding to each property configuration, where each $c_i \in Pol(p)$ defines a relationship derived by the attributes in $p.A$ and mechanisms used to implement the service. For instance, for property p_2 in Figure 3.1, a policy can be defined with conditions: c_1:no_of_server_instances$\geq$3 and c_2:recovery_time$\leq$15ms. In this context, a dependability certificate is granted to the

service when it satisfies the policy proving that it holds a property p with a given level of assurance against a given set of threats (see Section 3.5).

3.3 Service Modeling

The complete modeling of the service under certification represents a fundamental step to realize dependability certification, and serves as the basis for the certification authority to carry out its validation process. This section presents a modeling approach that allows the certification authority to generate an STS-based model of a service, based on *i)* the dependability property to be certified and *ii)* the information released by the service provider in the Web Service Description Language (WSDL) interface and/or the Web Service Conversation Language (WSCL) document.

3.3.1 WSDL-based Model

The WSDL interface is the least set of information that a service provider has to release to publish its service, and specifies the set of service operations and the methods of accessing them. In the certification scheme here, the WSDL interface is used to define the WSDL-based model of the service, as follows.

Definition 3.3.1 (WSDL-based model)**.** *Let $\mathcal{M}$ be the set of STS-based service models, a WSDL-based model $M_{wsdl}\in\mathcal{M}$ of a service* ws *is an STS that consists of a set $\{m_{wsdl}\}$ of connected components, each one modeling a single service operation. Each m_{wsdl} is in turn modeled as an STS $\langle \mathcal{S}, s_1, \mathcal{V}, \mathcal{I}, \mathcal{A}, \rightarrow \rangle$ (see Definition 3.2.1), where the number of actions modeling operation input and output is equal to two ($|\mathcal{A}|$=2) and the number of states is at least equal to three ($|\mathcal{S}|\geq$3).*

Intuitively, each $m_{wsdl}\in M_{wsdl}$ always includes three states modeling the operation interface as follows: *i)* $s_1\in\mathcal{S}$ is the initial state when no input has been received by the service operation, *ii)* $s_2\in\mathcal{S}$ is the intermediate state when the input is received while the output is not yet generated (i.e., when the operation is being performed), and *iii)* $s_3\in\mathcal{S}$ is the final state when the output has been generated and correctly returned to the client. The set $\mathcal{S}$ of states in m_{wsdl} can be extended to represent the *stateful* implementation of the service operation when its source code is available. In this case, the intermediate state s_2 consists of a number of sub-states as described in Example 3.3.1 at the end of this section. Guards g at state transitions model the functional correctness of the service and the specific configuration of dependability mechanisms.

3.3.2 WSCL-based Model

The WSCL document defines the service conversation as the communication protocol between the clients and the service, and the interactions/ordering between various operations

within the service. Given a service conversation, the aim is to define the WSCL-based model of the service in the form of an STS. Hence, the certification authority considers the connected components $m_{wsdl} \in M_{wsdl}$ as building blocks, and defines a set of modeling operators $op: \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$ that take as input two service models and return as output their combination. According to the operators typically defining the WSCL conversation, first the set $\mathcal{O}=\{\odot, \otimes\}$ of modeling operators op is defined, where $\odot$ is the sequence operator and $\otimes$ is the alternative operator. Then, the modeling operators are recursively applied on the WSCL-based model, using the connected components m_{wsdl} as basic elements, to derive M_{wscl} as follows (see Example 3.3.1).

$$M_{wscl} = m_{wsdl} \mid M_{wscl} \odot M_{wscl} \mid M_{wscl} \otimes M_{wscl}$$

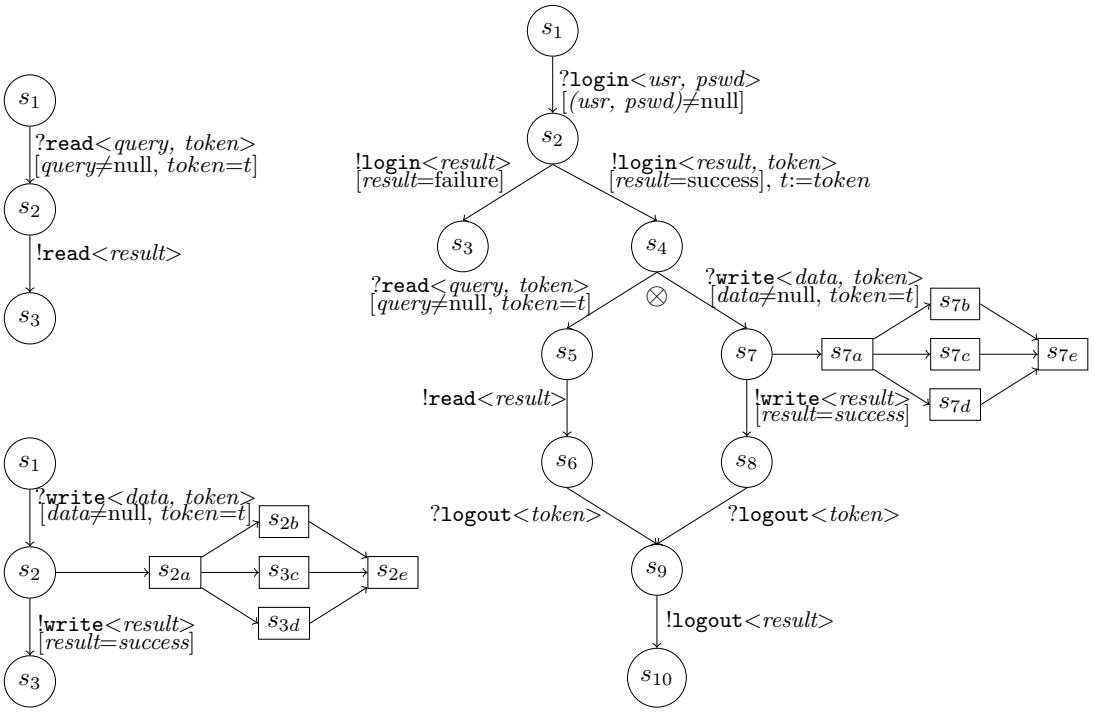
The WSCL-based model can be defined as follows.

Definition 3.3.2 (WSCL-based model)**.** *Let $\mathcal{M}$ be the set of service models, a WSCL-based model $M_{wscl} \in \mathcal{M}$ of a service* ws *is an STS $\langle \mathcal{S}, s_1, \mathcal{V}, \mathcal{I}, \mathcal{A}, \rightarrow \rangle$ (see Definition 3.2.1), where $\mathcal{S}$ is the union of all the states of component WSDL-based models integrated using the operators in $\mathcal{O}=\{\odot, \otimes\}$, and $\mathcal{A}$ represents the set of service operations involved in the conversation.*

Given two WSCL-based models M_1 and M_2 combined using $\otimes$, the initial states of the two models can be unified and represented using a single state (e.g., State s_4 in Figure 3.2(b) is the common state between operations `read` and `write` combined by operator alternative). Similarly, when the two WSCL-based models are combined using $\odot$, the final state of the first model M_1 and the initial state of the second model M_2 can be represented using a single common state (e.g., State s_4 in Figure 3.2(b) is the combination of the final state of operation `login` and the initial state of the choice between the operations `read` and `write`). We note that the WSCL-based model resulting from the application of these modeling operators can be further refined to derive a more clear while equivalent representation. For example, the final state of operation `login` in Figure 3.2(b) can be represented with two branches, where state s_3 is reached as a result of an internal trigger on login failure, and state s_4 represents the final correct state of operation `login`. Similarly to the WSDL-based model, the set $\mathcal{S}$ of states in M_{wscl} can be extended when the source code of the service operations is available. The implementation states of M_{wsdl} are included in M_{wscl} when corresponding interface states are involved in the conversation.

Example 3.3.1. *Figure 3.2 shows an example of an STS-based model of the storage service. The input actions are denoted as* ?`operation`<parameters>*, while the corresponding output actions are denoted as* !`operation`<results>*. The interface states are represented as circles and stateful implementation states as rectangles.*

Figure 3.2(a) shows two connected components of the WSDL-based model of the storage service modeling operation `read` *with no implementation states (i.e., $|\mathcal{S}|=3$) and operation* `write` *with implementation states (i.e., $|\mathcal{S}|>3$). We note that, while not shown in*



(a) WSDL-based model (b) WSCL-based model

Figure 3.2: An example showing the STS-based service models of the storage service

Figure 3.2(a), the model also includes connected components corresponding to operations `login` *and* `logout`*. Let us now consider property* p_2 *in Figure 3.1 as the property to be certified for operation* `write`*. Operation* `write` *starts at state* s_1 *waiting for an input. When the input is received and verified to be valid using the guard* data≠*null and* token=t*, a transition to state* s_2 *happens (functional correctness is verified). State* s_2 *contains five sub-states representing stateful implementation of the dependability mechanism, where data, metadata, and index are stored across three redundant storage servers. In particular, state* s_{2a} *denotes the state when input is provided to servers, states* s_{2b}*,* s_{2c} *and* s_{2d} *represent the state in which data are stored in three different servers, and state* s_{2e} *performs the output check. A transition from* s_{2a} *to* $s_i \in \{s_{2b}, s_{2c}, s_{2d}\}$ *is observed when the* i*-th server is up and running (i.e., guard [status(server*$_i$*)*=ok*] is verified), and a transition from* s_i *to* s_{2e} *when the* i*-th server returns* success *(i.e., guard [result(server*$_i$*)*=success*] is verified). For the sake of clarity, guards validating dependability mechanisms in transitions between states* s_{2a} *and* s_{2e} *are not shown in Figure 3.2. Transition from state* s_2 *to the final state* s_3 *happens when* success *is returned by all storage servers.*

Figure 3.2(b) illustrates the WSCL-based model of the conversation that allows the

client to access service operation `read` *or* `write`*, after it has been authenticated using operation* `login`*, and then disconnect using operation* `logout`*.* M_{wscl} *is generated by applying modeling operators in* $\mathcal{O}=\{\odot,\otimes\}$ *on the connected components in* M_{wsdl}*. The components representing operations* `read` *and* `write` *(see Figure 3.2(a)) are combined using* $\otimes$*, and then connected to operation* `login` *using* $\odot$*. Finally, operation* `logout` *is appended to operations* `read` *and* `write`*. The service starts in state* s_1 *where it receives the login credentials; if the authentication is successful, it transits to state* s_4 *and the update mapping assigns the login* token *to the internal variable* $t\in\mathcal{V}$*. In state* s_4*, the client can call either operation* `read` *or* `write` *with relevant parameters, and perform its task. The client can request to logout from the service in state* s_6 *or* s_8 *to reach the final state* s_{10}*. Set* $\mathcal{I}=\{$usr, pswd, result, token, query, data$\}$ *comprises the state interaction variables.*

A service model represents the functional and dependability properties of a service in the form of an STS, and serves as a building block to dependability certification for two reasons. First, it is at the basis of the certification model used to validate a dependability property with a given assurance level. Second, it is used, together with the threats specified in property p to be certified, to generate a set of test cases [130] (service requests) that are used to evaluate dependability behavior of the service and to award a certificate. We note that the more detailed the service model, the more complete and effective the generated test cases, and in turn the higher the certification quality.

3.4 Certification Model

The certification model is a Markov model representation of the service, that enables the certification authority to quantitatively measure the compliance of the service to a property p, and accordingly award a dependability certificate to the service. Section 3.4.1 presents the process that receives as input a service model and produces as output a certification model. Section 3.4.2 defines the concept of assurance level and an approach to calculate it.

3.4.1 Markov-based Representation of the Service

The Markov model representation of the service is generated by performing three activities: *i)* prune, *ii)* join states, map policy, and integrate absorbing states, and *iii)* add probabilities.

Prune. The prune activity applies a projection π on the service model, over dependability property p, to generate a projected model M^{π} that *i)* contains all Most Important Operations (MIOs) with respect to p, that is, operations that are needed to certify p, and *ii)* ensures that the projection is consistent with the specifications in the WSDL interface and WSCL document. To obtain the projected model, a labeling function $\lambda{:}\mathcal{S}^I\rightarrow\{0,1\}$ is introduced, where $\mathcal{S}^I\subseteq\mathcal{S}$ is the set of states in the interface part of the service model,

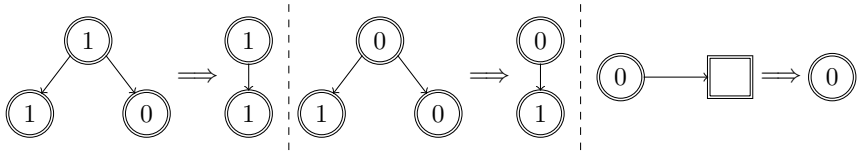


Figure 3.3: Pruning rules for interface and implementation states of M^{λ}

that marks each state $s \in \mathcal{S}^I$ with a binary value $\{0, 1\}$. The application of such labeling function results in a *labeled service model* M^{λ}, which extends the service model M by annotating each state $s \in \mathcal{S}^I$ corresponding to a MIO with 1 and all other states with 0. A state s shared by two or more operations (e.g., State s_4 in Figure 3.4) is labeled 1 if at least one of these operations is a MIO.

Using labeled service model M^{λ}, a set of *pruning rules* that are used to generate projected model M^{π} are defined as follows (see Figure 3.3; each three-state component of the service model is denoted with a circle, and implementation states with a rectangle).

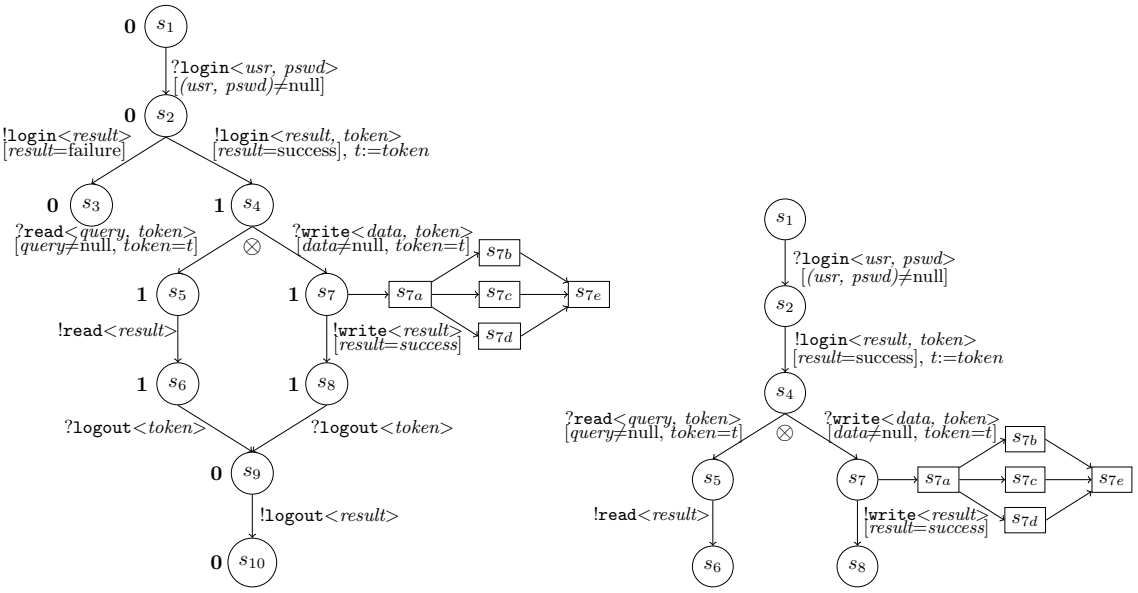
- *Pruning rule for interface states*: It operates recursively on the leaf states in the interface part of the labeled service model and removes those states for which $\lambda(s)=0$. To maintain consistency, if a state s_i has a descendant state s_j for which $\lambda(s_j)=1$, state s_i is not removed even if $\lambda(s_i)=0$.

- *Pruning rule for implementation states*: It removes all implementation states associated with an interface state s for which $\lambda(s)=0$.

We note that, when a WSDL-based model is considered, the pruning rules are applied on each connected component $m_{wsdl} \in M_{wsdl}$ independently, and the component is either removed or taken as it is. The *pruning* activity can then be viewed as a function π that takes a labeled service model M^{λ} as input, applies the pruning rules, and generates the projected model $M^{\pi} \in \mathcal{M}$ of the service as output. We note that $M^{\pi} = \langle \mathcal{S}^{\pi}, s_1^{\pi}, \mathcal{V}^{\pi}, \mathcal{I}^{\pi}, \mathcal{A}^{\pi}, \rightarrow^{\pi} \rangle$ is a sub-model of $M = \langle \mathcal{S}, s_1, \mathcal{V}, \mathcal{I}, \mathcal{A}, \rightarrow \rangle$, such that $\mathcal{S} \subseteq \mathcal{S}^{\pi}$, $\mathcal{V} \subseteq \mathcal{V}^{\pi}$, $\mathcal{I} \subseteq \mathcal{I}^{\pi}$, $\mathcal{A} \subseteq \mathcal{A}^{\pi}$, $\rightarrow \subseteq \rightarrow^{\pi}$.

Example 3.4.1. *Let p=(reliability, {mechanism=redundancy, no_of_server_instances=3,fault_type=server_crashes,recovery_time≤15ms}) be the property to be certified for the storage service and M in Figure 3.2(b) the WSCL-based model of the service. First the labeling function λ is applied on M to obtain the labeled service model M^{λ} illustrated in Figure 3.4(a). We note that the states corresponding to operations* `login` *and* `logout` *are marked* 0*, since they are not MIOs for the certification of p. Then, following the pruning rules, the projected service model M^{π} in Figure 3.4(b) is obtained. Here, operation* `logout` *has been removed, while a part of operation* `login` *has been maintained since it has at least a descendant state s such that $\lambda(s)=1$.*

Join states, map policy, and integrate absorbing states. This activity applies a transformation $\bowtie$ on the projected service model M^{π}, over dependability property p, to



(a) Labeled service model M^{λ} (b) Projected model M^{π}

Figure 3.4: An example of pruning activity applied on the model in Figure 3.2(b)

generate a new model $M^{\bowtie}$. It is composed of two steps, *i) join states and map policy*, and *ii) integrate absorbing states*, as follows.

The *join states and map policy* step is a manual process that depends on property p, policy $Pol(p)$, service model M, and implemented dependability mechanisms. It first joins the implementation states representing the dependability mechanisms of each service operation in M^{π}, to model the successful execution flow of service operations. It then uses guards and update mapping on transitions in M^{π} involving the joined states and, according to policy $Pol(p)$, produces the conditions that regulate transitions in $M^{\bowtie}$. It finally modifies the interface part of M^{π} by specifying, for each state transition, a set of conditions derived from g and u. In the following, state transitions are denoted as $\xrightarrow{c_{ij}}$, where c_{ij} are the conditions on transitions.

The *integrate absorbing states* step inserts two absorbing states C and F, representing the state of correct output and failure, respectively, and connects them to each leaf in the interface part of the model. From the certification point of view, state C is reached when the service satisfies the policy conditions in $Pol(p)$, while state F is reached in case of a policy violation.

The two steps of this activity can be viewed as a function $\bowtie$ that *i)* takes the projected service model M^{π} as input, *ii)* applies join states and map policy, and integrate absorbing states steps, and *iii)* generates a new model of the service $M^{\bowtie}=\langle \mathcal{S}^{\bowtie}, s_1^{\bowtie}, C, F, \xrightarrow{c_{ij}}^{\bowtie} \rangle$.

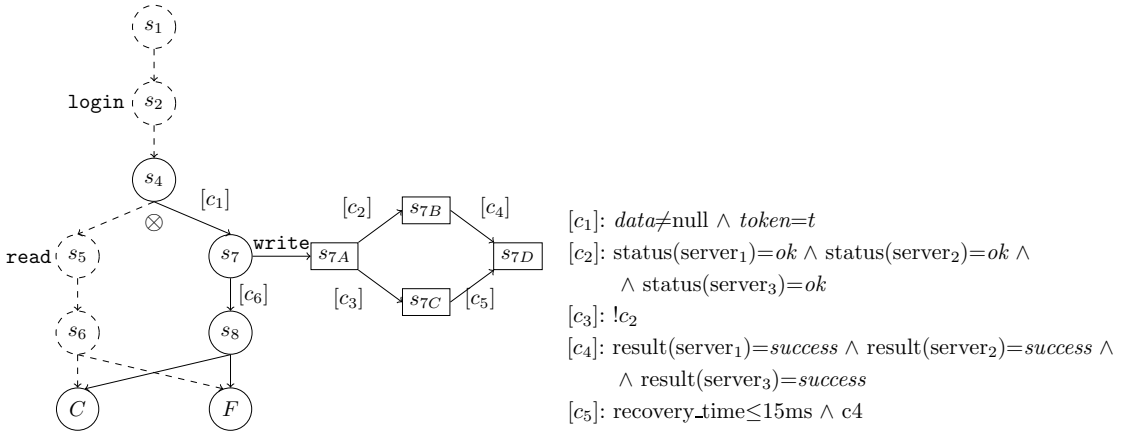


Figure 3.5: An example of model $M^{\bowtie}$ of the storage service, obtained after applying join states, map policy, and integrate absorbing states to operation `write` of model M^{π} in Figure 3.4(b)

Example 3.4.2. *Let M^{π} in Figure 3.4(b) be the projected model and p=(reliability, {mechanism=redundancy,no_of_server_instances=3,fault_type=server_crashes, recovery_time≤15ms}) the dependability property. For simplicity, in this example, consider the operation* `write` *(states s_4, s_7, s_8) only. The portion of model $M^{\bowtie}$ in Figure 3.5 referring to operation* `write` *(black lines) is generated as follows. The join states and map policy step is applied on M^{π} over p, to embed $Pol(p)$ within the model. To this aim, first M^{π} is modified using the implementation states of the dependability mechanism i.e., redundancy with three server instances (states $s_{7a} - s_{7e}$) and the states of the new model are generated as follows:* i) *state s_{7a} of M^{π} corresponds to state s_{7A} in $M^{\bowtie}$,* ii) *states s_{7b},s_{7c},s_{7d} are represented with states s_{7B} and s_{7C}, and* iii) *state s_{7e} is mapped to state s_{7D}. Then, the policy conditions in $Pol(p)$ are integrated with guards and update mappings in M^{π} to produce conditions [c_2]–[c_6]. Here, when the service reaches state s_7, first an implicit transition to the sub-state s_{7A} happens, where it sends the request to three replicated storage servers. The service then moves to state s_{7B} when all servers are fail-free (following condition [c_2]); otherwise, when one or more server crashes are detected, it transits to state s_{7C} ([c_3]). In s_{7C}, the service starts the recovery process and moves to state s_{7D} if recovery is performed in less than 15ms and all servers return* success *([c_5]). A transition from s_{7B} to s_{7D} is observed if* success *is returned by all the storage servers ([c_4]). The service moves to the output state s_8 following condition [c_6] in $M^{\bowtie}$, which is an aggregate of conditions applied at each sub-state of state s_7 ([c_2] to [c_5]). Finally, the absorbing states C and F are integrated and connected to the final states in the interface part of the model. We note that there is an implicit transition between each state s_i and F (denoted as (s_i, F)) if some unexpected errors or policy violation happen. We also note*

that if C is reached, a fail-free operation/conversation have been executed.

Add probabilities. The last activity extends $M^{\bowtie}=\langle \mathcal{S}^{\bowtie}, s_1^{\bowtie}, C, F, \overset{c_{ij}}{\rightarrow}^{\bowtie} \rangle$ with probability values to generate the Markov chain used in this chapter as the certification model M_{cert}. M_{cert} specifies probabilities Pr_{ij} to satisfy the conditions corresponding to each state transition $\overset{c_{ij}}{\rightarrow}$, and probability R_i to remain fail-free corresponding to each state s_i. In this context, similar to [27], R_iPr_{ij} represents the probability that execution of a service in state s_i will produce the correct results, and transfer the control to state s_j. The transition from the final state s_k to the correct state C, having probability R_k, is observed if the service satisfies relevant conditions in $Pol(p)$ (with no failures). We note that there is an implicit transition of probability $1-\sum_{j=1}^{k} R_iPr_{ij}$ from each state $s_i \neq s_k$ to F representing a failure or violation of the condition in that state. The transition from s_k to F has probability $1-R_k$. We also note that there can be multiple final states s_k in a WSCL conversation (e.g., s_6 and s_8 in Figure 3.5). In this case, the Markov model converges all final states to a single node that is then connected to C and F. The transition from one state to another is assumed to follow the Markov property, regardless of the point at which the transition occurs. The certification model M_{cert} then consists of a state-based, discrete-time Markov chain that combines the failure behavior and system architecture of the service to validate and certify a given set of dependability properties. M_{cert} can be defined as follows.

Definition 3.4.1 (Certification model)**.** *The certification model M_{cert} of a service is a 6-tuple $\langle \mathcal{S}, s_1, C, F, \overset{c_{ij}}{\rightarrow}, R_iPr_{ij} \rangle$, where: $\mathcal{S}=\mathcal{S}^{\bowtie}$ is the set of all states in the model, $s_1=s_1^{\bowtie}$ is the initial state; C is the final correct state; F is the final failure state; $\overset{c_{ij}}{\rightarrow}=\overset{c_{ij}}{\rightarrow}^{\bowtie}$ represents a transition relation between pairs of states (s_i, s_j) labeled with a condition c_{ij} derived from $Pol(p)$, g, and u; and R_iPr_{ij} is the probability that the service execution provides the correct results, satisfies the conditions in state s_i, and moves to state s_j.*

In case a WSDL-based model is used, M_{cert} is produced for each MIO in the service, while a single M_{cert} is generated for a WSCL-based model.

3.4.2 Assurance Level

This section presents an interpretation of the Markov model of the service as the assurance level used to quantitatively validate a dependability property p. The certification model

can be represented by a transition matrix Q' as follows.

$$Q' = \begin{array}{c} \\ C \\ F \\ s_1 \\ s_2 \\ \vdots \\ s_k \end{array} \begin{array}{c} \begin{array}{cccccc} C & F & s_1 & s_2 & \dots & s_k \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 1-\sum_{j=1}^{k} R_1Pr_{1j} & R_1Pr_{11} & R_1Pr_{12} & \dots & R_1Pr_{1k} \\ 0 & 1-\sum_{j=1}^{k} R_2Pr_{2j} & R_2Pr_{21} & R_2Pr_{22} & \dots & R_2Pr_{2k} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ R_k & 1-R_k & 0 & 0 & \dots & 0 \end{pmatrix} \end{array}$$

We note that, for the sake of clarity, conditions on transitions are not reported in Q'. The certification authority can use the matrix Q' to estimate the extent to which the service satisfies dependability property p by following the approach presented in [27]. Let Q be a matrix obtained from Q' by deleting rows and columns corresponding to C and F; μ is a matrix such that

$$\mu = I + Q + Q^2 + Q^3 \cdots = \sum_{x=0}^{\infty} Q^x = (I-Q)^{-1}$$

where I is the identity matrix with same dimension as Q. Here, the assurance level of the service is defined as follows.

Definition 3.4.2 (Assurance level)**.** *Assurance level L of a service deployed in a specific environment is the probability that it satisfies a policy $Pol(p)$ and holds a dependability property $p \in \mathcal{P}$ for a given rate of service executions.*

We note that the assurance level characterizes the extent to which a service communication that starts from the initial state s_1 will reach the final execution state s_k, and transit from s_k to the final correct state C. Assurance level L of a service can be estimated using $L = \mu_{1,k} * R_k$, where $\mu_{1,k}$ represents the probability value at 1^{st} row and k^{th} column of the matrix μ, and R_k is the probability of the final execution state to be fail-free. $\mu_{1,k}$ can also be computed using

$$\mu_{1,k} = (-1)^{k+1} \frac{|Q|}{|I-Q|}$$

where $|Q|$ and $|I-Q|$ represent the determinant of Q and $I-Q$, respectively.

When the certification model is generated using the WSDL-based model, assurance level L is calculated for each operation individually. Instead, when the certification model is generated using the WSCL-based model, assurance level of the overall conversation is calculated as a single value.

Example 3.4.3. *Figure 3.6 illustrates the certification model corresponding to operation* `write` *of the storage service in Figure 3.2(a). This model is obtained after applying*

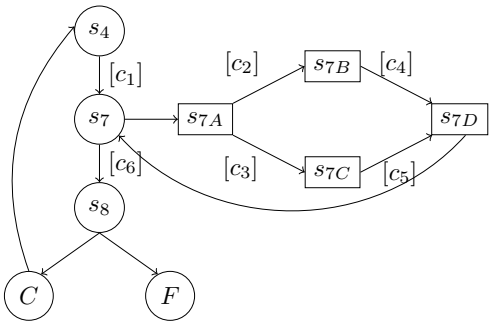


Figure 3.6: An example of a certification model for operation `write` in Figure 3.2(a)

prune, join states, map policy, integrate absorbing states and add probabilities to the service model. We note that conditions [c_1]–[c_6] are the ones discussed in Example 3.4.2. Let the fail-free probabilities of states s_4, s_7, s_8 computed by the certification authority be R_4=0.99, R_7=0.94, and R_8=0.97, and transition probabilities between nodes be Pr_{47}=0.93 and Pr_{78}=0.95. An approach to derive the probability values is discussed in Section 3.5. For simplicity, in this example, the probabilities of internal states s_{7A}–s_{7D} are not specified, while they are used to deduce probability R_7Pr_{78}. The corresponding transition matrix Q' and matrix $\mu=(I-Q)^{-1}$ are:

$$Q' = \begin{array}{c} \\ C \\ F \\ s_4 \\ s_7 \\ s_8 \end{array} \begin{array}{c} \begin{array}{ccccc} C & F & s_4 & s_7 & s_8 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0.0793 & 0 & 0.9207 & 0 \\ 0 & 0.1070 & 0 & 0 & 0.893 \\ 0.9700 & 0.0300 & 0 & 0 & 0 \end{pmatrix} \end{array} \qquad \mu = \begin{array}{c} \\ s_4 \\ s_7 \\ s_8 \end{array} \begin{array}{c} \begin{array}{ccc} s_4 & s_7 & s_8 \end{array} \\ \begin{pmatrix} 1 & 0.9207 & 0.8222 \\ 0 & 1 & 0.8930 \\ 0 & 0 & 1 \end{pmatrix} \end{array}$$

Here, $\mu_{4,8}$=0.8222. The probability that operation `write` *of the storage service satisfies a property p is L=0.8222∗0.97=0.7975.*

3.5 Dependability Certification Process

The certification scheme is designed as a two-phase process due to the highly dynamic nature of the Cloud computing and SOA environment. The first phase validates the dependability properties of the service before it is actually deployed in a system, and issues a certificate to the service provider based on the initial validation results (*offline phase* in Section 3.5.1). The second phase monitors the certified properties of the service at run-time, and updates the certificate based on real validation results (*online phase* in Section 3.5.2). This certification process results in a dependability certificate life-cycle, which represents the possible states of certificates (Section 3.5.3). For simplicity, in the

following, consider the certification process that proves and awards certificates with a single dependability property.

3.5.1 Offline Phase

The offline phase starts when a service provider requests the certification authority to issue a certificate to its service for dependability property p. The certification authority first generates the service model based on p and the specifications released by the service provider. After verifying that the service model conforms to the real service implementation, the certification authority derives the certification model M_{cert} as discussed in Section 3.4. In this solution, the service model is used to generate service executions (test cases) that are used with M_{cert} to perform property validation. A validation function f is defined to verify that the service satisfies policy $Pol(p)$ using M_{cert}, assuming that each condition c_i in M_{cert} is specified as a boolean valued predicate. The service proves to hold p when the conditions in $Pol(p)$ are satisfied with a given level of assurance. The validation function is defined as follows.

Definition 3.5.1 (Validation function)**.** *The validation function* $f{:}(ws, p, M_{cert}, k) \rightarrow \{true, false\}$ *takes service* ws *under evaluation, dependability property* p *to be validated, certification model* M_{cert} *integrating* $Pol(p)$*, and an index* k *referring to the service execution that triggers policy verification as input, and returns* $true$ *when relevant conditions in* $Pol(p)$ *are satisfied with respect to* M_{cert}*,* $false$ *otherwise, as output.*

In other words, the validation function verifies if a given service execution reaches state C (success) or state F (failure) of the certification model in Definition 3.4.1. We note that the certification model of the service and the dependability property remain constant, while the index k may change over time. We also note that the service is static, while its context changes. In particular, k is an index referring to the service executions (i.e., the validation tests) used by the certification authority to verify the dependability property of the service under different contexts (e.g., using fault injection).

Example 3.5.1. *Consider the certification model for operation* `write` *in Figure 3.6. The certification authority validates property* p*=(reliability, {mechanism=red-undancy, no_of_server_instances=3, fault_type=server_crashes, recovery_time*$\leq$*15ms}) by performing a sequence of tests driven by the service model of operation* `write` *in Figure 3.2(a). For each test iteration, the validation function* f *returns* $true$ *if all conditions in the execution path are satisfied and the service reaches state* C*; it returns* $false$ *and moves to state* F *from any of its states, otherwise.*

The results of the validation function are used by the certification authority to estimate the values of $R_i Pr_{ij}$ in Definition 3.4.1, and L in Definition 3.4.2. To this aim, a *frequency log* that maintains f's results is introduced. A frequency log is a list of triplets $(k, \{v_k\}, s_i)$, where: k is the index of the test request in Definition 3.5.1; $\{v_k\}$ represents the attribute

value(s) causing a transition to F; and s_i is the state of the certification model in which a transition to F is observed. We note that $\{v_k\}$ and s_i are empty if f returns $true$. We also note that state s_i is identified by observing the fault returned by the service under certification and by accessing the state of the service model in which the execution is currently blocked, since a guard that is linked to one or more policy conditions in the certification model is violated. Each probability R_iPr_{ij} can then be calculated, using the frequency log, as the number of successful transitions (s_i,s_j) over the total number of test requests reaching s_i. The total number of requests is such that each path in the model is tested for a given number of times. As an example, let us consider the certification model in Figure 3.6 and property p in Example 3.5.1, and suppose that the service fails to recover from a server crash in state s_{7C}; the frequency log registers (k,{no_of_server_instances=2,recovery_time>15ms},s_{7C}). On the basis of the values of R_iPr_{ij} estimated using the validation function, assurance level L in Definition 3.4.2 is quantified by performing the matrix operations described in Section 3.4, and used to characterize the dependability property of a service.

When M_{wsdl} is used, each most important operation (MIO) is validated individually, and assurance level is calculated for each MIO independently. Let o_i be a MIO and L_{o_i} be its assurance level. A dependability certificate is issued to the service if the assurance level of each operation L_{o_i} is greater than or equal to a predefined threshold $\mathcal{T}\in[0,1]$. When M_{wscl} is used, the overall conversation is validated, and a certificate is issued to the service if assurance level L of the conversation is greater than or equal to $\mathcal{T}$. The dependability certificate is then of the form $\mathcal{C}(p, M, \{(o_i, L_{o_i})\})$, where: *i)* p represents the dependability property supported by the service; *ii)* $M\in\{M_{wsdl}, M_{wscl}\}$ is the service model; *iii)* $\{(o_i, L_{o_i})\}$ includes assurance level L_{o_i} with which each o_i supports p when a WSDL-based model is used; it contains a single pair (–,L) when a WSCL-based model is used. We note that a service certified using M_{wscl} is first certified using M_{wsdl}.

3.5.2 Online Phase

The online phase starts immediately after the service provider deploys its certified service. In this phase, the certification authority continuously verifies the validity of the certificate issued to the service, since, in complex digital ecosystems, dependability properties may change over time, resulting in outdated certificates. For example, certified reliability and availability properties of the service may change if a replica failure or network congestion happens. To this aim, *Evaluation Body* is introduced as a component that is owned by the certification authority and placed in the system where the certified service is deployed, to monitor its dependability property. In particular, when the WSCL-based model is available, overall conversation is monitored using M_{wscl}; otherwise, each MIO is monitored individually using connected components $m_{wsdl}\in M_{wsdl}$. The results obtained by monitoring are then reflected on the corresponding certification model(s) M_{cert} to verify $Pol(p)$, and to update the assurance level(s) in the certificate at run-time.

Since the certification model generates all possible states of the service, the number of states can be extremely large. However, to monitor and verify dependability properties of a service, we do not need the complete Markov model. Therefore a lightweight Markov model is derived by reducing the original one while maintaining its accuracy. We note that this reduction is applied on the certification model to improve the performance of the monitoring process, and reduce requirements on the platform deploying the service. A reduced certification model $\tilde{M}_{cert}$ is formally defined as follows.

Definition 3.5.2 (Reduced certification model)**.** *Let M_{cert} be a certification model, a reduced certification model $\tilde{M}_{cert}$ is of the form $\tilde{M}_{cert}=\langle \mathcal{S}, s_1, C, F, \stackrel{c_{ij}}{\rightarrow}, R_i Pr_{ij} \rangle$, such that, $|\tilde{M}_{cert}(\mathcal{S})|<|M_{cert}(\mathcal{S})|$. For all validation tests,* i) *$f(ws, p, \tilde{M}_{cert}, k)=f(ws, p, M_{cert}, k)$ and* ii) *the frequency logs for $\tilde{M}_{cert}$ and M_{cert} are consistent.*

The frequency logs are consistent if, for each entry $(k, \{v_k\}, s_i)$ generated using M_{cert}, there exists $(\tilde{k}, \{\tilde{v}_k\}, \tilde{s}_i)$ generated using $\tilde{M}_{cert}$, such that $k=\tilde{k}$, $\{v_k\}=\{\tilde{v}_k\}$, and $s_i=\tilde{s}_i$ or $\tilde{s}_i$ is a combination of states including s_i. For example, states s_{7B} and s_{7C} of M_{cert} in Figure 3.6 can be combined to a single state $\tilde{s}_{7BC}$ to obtain a reduced Markov model $\tilde{M}_{cert}$. In $\tilde{M}_{cert}$, service moves from s_{7A} to $\tilde{s}_{7BC}$ following a combination of $[c_2]$ and $[c_3]$, and from $\tilde{s}_{7BC}$ to s_{7D} following a combination of $[c_4]$ and $[c_5]$. Transition $(\tilde{s}_{7BC}, F)$ is a combination of transitions (s_{7B}, F) and (s_{7C}, F) in M_{cert}. The results of f using $\tilde{M}_{cert}$ are then the same as the ones obtained using M_{cert}.

At run-time, the evaluation body monitors service executions by obtaining real-attribute values using the service model, and maps them to $\tilde{M}_{cert}$. A dependability certificate issued to a service remains valid if its real-attribute values satisfy the conditions in the policy with a given level of assurance. For each service execution verified using $f(ws, p, \tilde{M}_{cert}, k)$, the probability values in the original model M_{cert} (and matrix Q') must be updated using the results of f, and the assurance level of the service must be recomputed in order to verify if $L_{o_i} \geq \mathcal{T}$ for each service operation o_i in case of WSDL-based model and $L \geq \mathcal{T}$ in case of WSCL-based model. To this aim, as in the offline phase, a *frequency log* that stores f's results is used within the evaluation body. We note that the source of failure or policy violation in M_{cert} can be precisely located using the service model, the frequency log, and $\tilde{M}_{cert}$. We also note that the update of matrix Q' is done periodically to preserve system performance.

The notion of assurance level is extended to support the validation process of the certification authority, and a random variable L^t is defined to characterize the dependability property of a service at run-time. For simplicity, in the remaining of this section, L^t is used to refer to the assurance level at time t for certification processes that rely on both WSDL-based and WSCL-based models. Given the time instant t at which the evaluation body starts the matrix update, L^t represents the assurance level of the service quantified by matrix Q', updated using the reduced Markov model $\tilde{M}_{cert}$ and the frequency log. Assurance level L^t observed by the evaluation body leads to the following conditions: *i)* $L^t \geq L^0$, where L^0 is the assurance level when the certificate was issued to the service in

the offline phase. This implies that $L^t \geq \mathcal{T}$, that is, the assurance value of the service at run-time is still greater than the predefined threshold value $\mathcal{T}$, and the dependability certificate of the service remains valid; *ii)* $L^t < L^0$, in this case, the evaluation body first checks whether $L^t \geq \mathcal{T}$. If true, the certificate remains valid; otherwise, the certification authority either updates the dependability certificate or revokes it. The definition of dependability certificate as $\mathcal{C}(p, M, \{(o_i, L^t_{o_i})\})$ is extended to comply with the dynamic changes in service dependability.

3.5.3 Dependability Certificate Life-cycle

The certificate life-cycle starts in the offline phase when the certification authority issues a certificate $\mathcal{C}$ to a service and marks it as *valid*. $\mathcal{C}$ is associated with a validity period t_e, where e is the expiration date of the certificate. During the online phase, the evaluation body monitors the service executions, checks the certificate validity, and updates the assurance level in the certificate using real-attribute values. As long as $L^t \geq \mathcal{T}$ and $t < t_e$, for property p and model M, certificate $\mathcal{C}$ remains valid. The following situations can occur when $L^t < \mathcal{T}$ and $t < t_e$.

- The certification authority builds a new certification model for a new property $p_i \preceq_P p$, by relaxing some policy conditions. For example, if the original policy condition is *no_of_server_instances* ≥ 3, the new certification model may relax the condition as *no_of_server_instances* ≥ 2, and consider a new property with two replicas. Based on the new certification model, validation tests are performed on the service by monitoring real service executions; if $L^t \geq \mathcal{T}$, a *downgraded* certificate with property p_i is issued to the service.
- When a downgraded certificate cannot be generated, $\mathcal{C}$ is *revoked*.

At a given point in time, if the service with a downgraded certificate resumes (part of) correct functionality and satisfies dependability property p with $L^t \geq \mathcal{T}$, using the original certification model (e.g., the model integrating policy condition *no_of_server_instances* ≥ 3), an *upgraded* certificate is issued to the service. We note that, while the assurance level in the upgraded certificate can be higher than the one in the original certificate, the property can be at most the one in the original certificate. Finally, based on the validity time t_e, certificate $\mathcal{C}$ can be renewed as follows. The certification authority starts a renewal process at time $t_i < t_e$, where the exact time t_i depends on the considered scenario, to re-validate dependability property p, and in turn the certificate, for the service. If $L^{t_i} \geq \mathcal{T}$ holds, a renewed *valid* certificate is offered to the service, with a new initial assurance level L^0 and a new validity time t_e. Otherwise, if $L^{t_i} < \mathcal{T}$ or t_e expires, the certificate becomes *invalid*. We note that, at any point in time, a certificate can be either valid, invalid, upgraded, downgraded, or revoked. In the following, when clear from the context, the valid, downgraded, and upgraded certificates are referred as simply valid certificates.

3.6 Dependability Certificate-Based Service Selection

The aim of the dependability certification scheme is to provide a solution where services can be searched and selected at run-time based on their dependability certificate and client's dependability requirements. To this aim, the service discovery component extends standard service registries *i)* to incorporate the dependability metadata in the form of certificates and *ii)* to support the matching and comparison processes described in the following of this section.

Let us consider a service registry that contains a set of services ws_j, each one having a dependability certificate $\mathcal{C}_j(p, M, \{(o_i, L^t_{o_i})\})$. A client can define its dependability requirements $Req(p, M, \{(o_i, L_{o_i})\})$ in terms of preferences on *i)* dependability property $Req.p$, *ii)* granularity of the service model used to validate and certify the service $Req.M \in \{M_{wsdl}, M_{wscl}\}$, and *iii)* assurance level $Req.\{(o_i, L_{o_i})\}$ for M_{wsdl} or $Req.(-, L)$ for M_{wscl}.

The matching process performs an automatic matching of client's requirements Req against valid dependability certificates $\mathcal{C}_j$ of services in the registry, and returns the set of services satisfying the specified requirements. The matching process implements a three-step process as follows [9].

- *Property match*: it selects services such that $Req.p \preceq_P \mathcal{C}_j.p$, using the hierarchy of dependability properties defined in Section 3.2.

- *Model match*: it selects services such that $Req.M \preceq_M \mathcal{C}_j.M$, that is, either $Req.M = \mathcal{C}_j.M$, or $Req.M = M_{wsdl}$ and $C_j.M = M_{wscl}$. The latter condition holds since a service certified for M_{wscl} is first certified for M_{wsdl}.

- *Assurance level match*: it selects services on the basis of the assurance level in the certificate. In case of WSDL-based model, a service is selected iff $Req.L_{o_i} \leq \mathcal{C}_j.L^t_{o_i}$ for each operation o_i. In case of WSCL-based model, a service is selected iff $Req.L \leq \mathcal{C}_j.L^t$.

The matching process returns a set WS of services compatible with client's preferences, according to property, model, and assurance level matches. The comparison process takes WS as input and transparently generates an ordering of services. The goal of this phase is to rank the shortlisted set of services in WS based on their certificates so as to facilitate the client in selecting the most appropriate service among the compatible ones. Given two services $ws_j, ws_k \in WS$ with certificates $\mathcal{C}_j$ and $\mathcal{C}_k$, respectively, the ordering of services is performed based on the hierarchical relationship among dependability properties, the model granularities, and the assurance level values. We note that, in some cases, there can be inconsistencies in the comparison (e.g., $\mathcal{C}_j.p \preceq_P \mathcal{C}_k.p$ and $\mathcal{C}_j.M \preceq_M \mathcal{C}_k.M$, but $\mathcal{C}_j.L^t \not\leq \mathcal{C}_k.L^t$). In this context, a default precedence rule in which the property is more important than the model, and the model is more important than the assurance level

Table 3.2: An example of dependability certificates and client's requirements

Dependability certificates

	$\hat{p}$	A	M	L^t
$\mathcal{C}_{st_1}$	Reliability	mechanism=redundancy no_of_server_instances=3 fault_type=server_crashes	M_{wscl}	L^t=0.98
$\mathcal{C}_{st_2}$	Reliability	mechanism=redundancy no_of_server_instances=4 fault_type=server_crashes recovery_time≤15ms	M_{wsdl}	$L^t_{\texttt{read}}$ =0.96 $L^t_{\texttt{write}}$=0.95
$\mathcal{C}_{st_3}$	Reliability	mechanism=redundancy no_of_server_instances=4 fault_type=server_crashes recovery_time≤15ms	M_{wscl}	L^t=0.90
$\mathcal{C}_{st_4}$	Availability	mechanism=redundancy recovery_time≤15ms	M_{wscl}	L^t=0.92

Client's requirements

	$\hat{p}$	A	M	L
Req	Reliability	mechanism=redundancy no_of_server_instances≥3 fault_type=server_crashes	M_{wscl}	L≥0.85

is assumed. We note that different precedence rules can also be used based on client's preferences [9]. A (partially) ordered set $\overline{WS}$ of services in WS is returned to the client as the output of the comparison phase.

Example 3.6.1. *Let us consider a client searching for a storage service at time t, and a service discovery with four storage services st_1, st_2, st_3, st_4, each one having a valid dependability certificate $\mathcal{C}_{st_1}$, $\mathcal{C}_{st_2}$, $\mathcal{C}_{st_3}$, $\mathcal{C}_{st_4}$. Table 3.2 presents the client's requirements Req and the four dependability certificates.*

Upon receiving request Req from the client, the service discovery starts the three-step matching process. First, it matches the client's requirement on dependability property Req.p against the dependability property in the certificates. Here, service st_4 is filtered out because $Req.p \not\preceq_P \mathcal{C}_4.p$. The service discovery then performs a match on the model used to certify services. In this step, st_2 is not selected since $Req.M \not\preceq_M \mathcal{C}_2.M$. Finally, the matching process considers the assurance level and returns WS=$\{st_1, st_3\}$*, since $Req.L \leq \mathcal{C}_1.L^t$ and $Req.L \leq \mathcal{C}_3.L^t$. The result of the matching process is the set of compatible services* WS *that is given as input to the comparison process. The comparison process then compares certificates C_{st_1} and C_{st_3}, and produces an ordered list* $\overline{\text{WS}}$=$\{st_3, st_1\}$ *since $C_{st_1}.p \preceq_P C_{st_3}.p$. Service st_3 is finally returned to the client as the most appropriate service that satisfies its preferences.*

The matching and comparison processes are extended to complement the certification scheme and, to provide a two-phase service selection solution. In the first phase, a static service selection is performed when the client sends a request to the service discovery. The second phase starts when the client selects a service $ws_j \in \overline{WS}$. In this phase, service discovery performs constant monitoring of the certificate status for ws_j. If a certificate is downgraded, revoked, or moves to invalid state, the service discovery triggers the matching and comparison processes and replaces the originally selected service with a new, compatible, service $ws_k \in \overline{WS}$. The second phase is transparent to the client and allows the certification solution to ensure clients requirements also during run-time.

3.7 Certifying Business Processes

A service provider can implement its business process as a composition of different services, provided by different suppliers. To this aim, it defines a *template* specifying the order in which service operations must be called, the data to be exchanged in each phase of the composite service workflow, and the conditions under which a given service instance must be integrated within the business process. This chapter considers Business Process Execution Language (BPEL), a de-facto standard for web service composition [6]. BPEL templates define executable processes using XML and mainly consider functionality requirements in the selection of component services to be integrated in a business process. The goal is then to extend the modeling approach and the certification scheme discussed in this chapter to give a solution to the run-time certification of dependability properties for composite services.

3.7.1 Modeling a Service Composition

Given the BPEL template and the WSDL-based model of partner services, the BPEL-based model M_{bpel} of the composition is defined, which is then used to certify the dependability property of the business process. To this aim, the set $\mathcal{O}$ of operators in Section 3.3.2 is extended with the parallel operator $\oplus$, that is, $\mathcal{O}=\{\odot, \otimes, \oplus\}$. The parallel operator is used to model processes involving the simultaneous invocation of different operations; for instance, in eShop, operation `browseItems` of three vendor services are invoked in parallel. Operators in $\mathcal{O}$ are recursively applied on the connected components m_{wsdl} of partner services to incrementally derive M_{bpel} as follows.

$$M_{bpel} = m_{wsdl} \mid M_{bpel} \odot M_{bpel} \mid M_{bpel} \otimes M_{bpel} \mid M_{bpel} \oplus M_{bpel}$$

The BPEL-based model can be formally defined as follows.

Definition 3.7.1 (BPEL-based model)**.** *Let $\mathcal{M}$ be the set of service models, BPEL-based model $M_{bpel} \in \mathcal{M}$ of a service is an STS $\langle \mathcal{S}, s_1, \mathcal{V}, \mathcal{I}, \mathcal{A}, \rightarrow \rangle$ (see Definition 3.2.1), where $\mathcal{S}$ is the union of all the states of the WSDL-based models of partner services integrated using*

the operators in $\mathcal{O}$=$\{\odot, \otimes, \oplus\}$, and $\mathcal{A}$ represents the set of service operations selected and integrated in the business process.

We note that given two BPEL-based models M_1 and M_2 composed using the parallel operator $\oplus$, the initial states of the two models are represented as a single state where the input is distributed to the two parallel flows; similarly, the final states of the two models are represented as a single state where the results of the two parallel executions are combined. For sequence $\odot$ and alternative $\otimes$ operators, the STS-based model is built as discussed in Section 3.3.2 for M_{wscl}. The conceptual difference between the WSCL-based and BPEL-based models is that the former considers operations of a single service, while the latter of different services. As for the WSCL-based model, the set $\mathcal{S}$ of states in M_{bpel} can also be extended when the source code of the service operations is available, and there is an implicit transformation from M_{wsdl} to M_{bpel} on the basis of the WSDL-based model of partner services and $\mathcal{O}$.

3.7.2 Certification Scheme for Business Processes

The dimension of failures in business processes is significantly different from monolithic services, since business process dependability is affected by the composition protocol and partner services. This implies that business process owner (the client in the certification framework) must utilize those partner services that not only satisfy its functional requirements, but also its requirements on dependability properties. The certification scheme therefore requires the client to: *i)* define the business process in the form of a BPEL template, *ii)* select dependability property p to be certified for its business process, *iii)* extend the BPEL template with a set of requirements Req_j on the dependability of each partner service ws_j to be integrated. In the following, for the sake of clarity, assume Req_j includes only requirements on property $Req_j.p$.

When requirements $Req_j.p$ are defined in the BPEL template, the client can use the certificate-based matching and comparison processes in Section 3.6 to select appropriate partner services. The chosen services can then be integrated and orchestrated to realize the business process (which is called BPEL *instance* here). We note that the BPEL instance produced in this manner will hold property p, or a stronger property, since the matching and comparison processes select services ws_j considering $Req_j.p$ as the lower bound. To this aim, assume a common ontology specifying rules for property composition. This ontology can drive the client in the specification of BPEL templates annotated with suitable requirements for the certification of specific properties of composite services.

The certification process starts when a client releases its BPEL instance, and requests the certification authority to certify property p for the business process. Differently from the certification of single services, the certification authority cannot determine assurance level and certify a composite service a priori during the offline phase, because the integration of component services is performed at run-time. However, to avoid downtimes in

service certification, first the assurance level L_{bpel} of a composition can be estimated at run-time, according to the following rules.

- When two operations o_i and o_j are composed in a sequence, the assurance level of the sequence is L_{ij}=$L^t_{o_i} * L^t_{o_j}$, and o_i and o_j are considered as a single operation o_{ij}. This rule is based on the assumption that partner services perform their operations independently of each other. For example, suppose operation `shipItems` of shipping service sh and operation `write` of storage service st are invoked in a sequence, the assurance level of their composition is $L^t_{sh} * L^t_{st}$.
- When two operations o_i and o_j are composed in a parallel or alternative, the assurance level is L_{ij}=min$(L^t_{o_i}, L^t_{o_j})$, and o_i and o_j are considered as a single operation o_{ij}. For example, when eShop invokes operation `browseItems` from three independent vendors v_1, v_2 and v_3 in parallel, it can perform its correct functionality (e.g., it generates a table of items returned from all three vendors) only if all three vendors behave correctly. If either vendors fail, the dependability of eShop is affected. Therefore, the assurance level of the composition is min$(L^t_{v_1}, L^t_{v_2}, L^t_{v_3})$.

The above rules are recursively applied by the certification authority to a BPEL instance as follows: *i)* all pairs in a sequence are considered; *ii)* when no sequences are left, an alternative or parallel is considered; *iii)* points *i)* and *ii)* are repeated until a single operation o is left. The certification authority then estimates assurance level L_{bpel} of the composition and awards a *temporary* dependability certificate $\mathcal{C}(p, M_{bpel}, L_{bpel})$, if $L_{bpel} \geq \mathcal{T}$. We note that, since dependability requirements $Req_j.p$ given as input to the matching and comparison processes represent lower bounds for service selection, the certification authority could include a stronger property p_j (i.e., $Req_j.p \preceq_P p_j$) in the certificate. As an example, consider a client specifying its requirements for two partner services in a sequence as $Req_j.p$=(reliability, {mechanism=redundancy, no_of_server_instances=2, fault_type=server_crashes}); then, suppose that the matching and comparison processes return two services with property p_j=(reliability,{mechanism=redundancy, no_of_server_instances=4, fault_type=server_crashes}), with $Req_j.p \preceq_P p_j$. Clearly, if $L_{bpel} \geq \mathcal{T}$, the composition is certified for property p_j.

After releasing the temporary certificate, the certification authority first produces the Markov-based certification model as discussed in Section 3.4.1. It then monitors the business process by observing executions of the BPEL instance, maintains the results in the frequency log, and updates the Q' matrix as discussed in Section 3.5.2. When a given (set of) quality metric has been satisfied by service executions (e.g., all the execution paths in the BPEL instance have been invoked a sufficient number of times or a given coverage of the service model has been achieved), the certification authority calculates the new assurance level L^t_{bpel} at time t using the approach in Section 3.5.2. If $L^t_{bpel} \geq \mathcal{T}$, the temporary certificate becomes a valid certificate with assurance level L^t_{bpel}, otherwise it

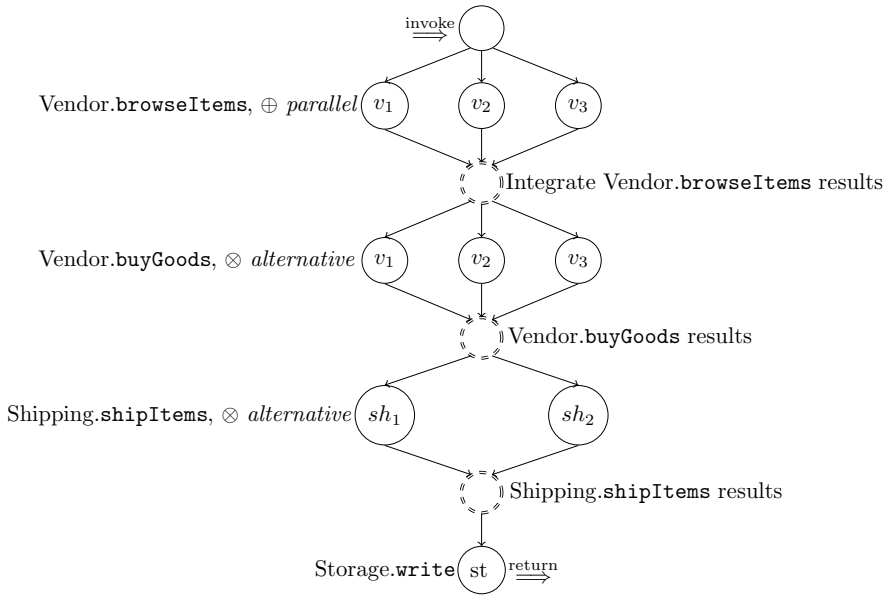


Figure 3.7: An example of a composition in the eShop business process

is revoked. It is important to note that in case a service ws_j in a composition becomes unavailable or its certificate violates $Req_j.p$, the selection process in Section 3.6 is executed to substitute ws_j with another candidate ws_k on the basis of the dependability requirement $Req_j.p$ in the BPEL template. As soon as ws_k has been selected and integrated, the process restarts with the generation of a temporary certificate.

Example 3.7.1. *Let us consider the reference scenario in which eShop composes three vendor services (v_1, v_2, v_3), two shipment services (sh_1, sh_2), and a single storage service (st). eShop defines a BPEL template and requires property Req.p=(reliability, {mechanism=redundancy, no_of_server_instances≥3, fault_type=server_crashes}) for all services to be composed. It then uses the matching and comparison processes in Section 3.6 to select them. Figure 3.7 illustrates the BPEL instance addressing the above requirements. Here, for simplicity, assume that each selected service (v_1, v_2, v_3, sh_1, sh_2, st) has been certified for Req.p, with M_{wsdl}, and with the same assurance level L^t for all its operations. The following values for L^t are considered: $L^t_{v_1}$=0.95, $L^t_{v_2}$=0.98, $L^t_{v_3}$=0.92, $L^t_{sh_1}$=1, $L^t_{sh_2}$=0.95, and L^t_{st}=0.96.*

The certification of the BPEL instance starts with the generation of a temporary certificate. Since there are no sequences, the certification authority first considers operation `browseItems` *from vendor services that are executed in parallel, and estimates the assurance level as* $L_{v_{123}}=\min(0.95, 0.98, 0.92)=0.92$. *The same process applies to operation* `buyGoods` *of vendor services composed in an alternative, where* $L_{v'_{123}}=\min(0.95, 0.98, 0.92)=0.92$. *The operations* `browseItems` *and* `buyGoods` *are further*

composed in a sequence, and the overall assurance level is $L_{v_{123},v'_{123}}$=0.92∗0.92=0.8464. *The shipment services are then invoked in an alternative, and the assurance level estimated as* $L_{sh_{12}}$=min(1, 0.95)=0.95. *Finally, eShop composes the vendor, shipment, and storage services in a sequence. The assurance level of eShop is estimated as* L_{bpel}=($L_{v_{123},v'_{123}}$∗$L_{sh_{12}}$)∗L_{st}=(0.8464∗0.95)∗0.96=0.772, *where the assurance level of the sequence between vendor and shipment services is first estimated, and the overall assurance level calculated as the sequence between the sequence of vendor and shipment services and the storage service. The certification authority generates a temporary certificate and issues it to eShop, if* $L_{bpel} \geq \mathcal{T}$.

After the release of a temporary certificate, eShop is continuously monitored by the certification authority and a valid certificate $\mathcal{C}(p, M_{bpel}, L^t_{bpel})$ *at time t is awarded to it, if it satisfies property p with assurance level* $L^t_{bpel} \geq \mathcal{T}$. *When one or more among* v_1, v_2, v_3, sh_1, sh_2, st *become unavailable or their certificates violate client's requirements, the certificate for the composition is revoked and the validation process resumes.*

3.8 Chapter Summary

This chapter presented a dependability certification scheme in which a machine-readable certificate is issued to the service after validating its dependability properties using Markov chains. The service is continuously monitored at runtime and the validity of the issued certificate is verified. We showed how the certification scheme can be integrated within existing service-based infrastructures, allowing users to select services with a given set of dependability properties and ensuring that users' requirements are addressed at runtime. Finally, building on the certificate-driven selection solution for monolithic services, a modeling and certification solution for business processes (service compositions) is presented.

Our certification scheme serves as *i)* an awareness tool that reports the QoS of a given service, and *ii)* an assurance tool that allows users to verify the compliance of a given service against a set of dependability properties. Given a service implemented using a set of dependability mechanisms (e.g., Cloud storage service) by the service provider, our certification scheme provides the basic support essential to assess the dependability that can be obtained by its use.

Service providers typically implement a standard set of dependability mechanisms that satisfy the basic needs of most users, whereas, each application has unique dependability requirements. This implies that certification techniques alone may not be sufficient and a more pro-active approach to satisfy specific dependability goals of individual applications is necessary. In the next chapter, we investigate an innovative approach of offering dependability as a service to users' applications. In particular, we consider the scenario where service providers can develop their services only by implementing the functional aspects, and obtain required dependability properties from a third-party as an additional service.

4

System-level Dependability Management

Our certification scheme presented in Chapter 3 validates dependability properties of services and awards machine-readable certificates. It is applied after a service provider implements her service with a set of dependability mechanisms (a posteriori) and allows service consumers to select and integrate services satisfying their dependability requirements. In other words, it serves as an awareness tool that offers the basic support necessary to build dependable applications. In contrast to the certification scheme, in this chapter, we aim to *relieve users from implementing low-level mechanisms*, and at the same time, obtain desired dependability properties for their services from a third party. This a priori approach significantly improves dependability when compared to the certification scheme and denotes the second level of dependability support offered in this thesis.

This chapter introduces an innovative, system-level, modular perspective on creating and managing dependability in Clouds. The proposed system allows users to build applications only considering its functional properties and obtain dependability support transparently from a third party. In particular, our system inserts a dedicated service layer that allows users to specify and apply desired level of dependability without needing any knowledge about the low level techniques that are available in the envisioned Cloud and their implementations. In this manner, our solution overcomes users' dependability issues with respect to the design of reliable and high available applications.

This chapter discusses a two-stage delivery scheme that offers dependability support to users' applications and a conceptual framework comprising all the components necessary to realize the notion of offering dependability as a service.

4.1 Introduction

The availability of an extensible pool of resources in Cloud computing provide an effective alternative for the users to deploy their applications with high scalability requirements. In general, the IaaS service provider builds its infrastructure by connecting large-scale

data centers, and delivers required amounts of computing resources to the users as an on-demand service over the Internet, using virtual machines (e.g., [44, 45]). This computing paradigm has changed the dimension of risks on user's applications, specifically because the failures (e.g., server overload, network congestion, hardware faults) in the data centers are outside the control scope of the user's organization [132, 56]. Nevertheless, these failures affect the applications deployed in the virtual machines and, as a result, there is an increasing need to address user's reliability and availability concerns.

The most typical way of achieving reliable and highly available software is to make use of dependability methods at procurement and development time. This implies that users must understand dependability techniques and tailor their applications by considering environment-specific parameters during the design phase. However, for the applications to be deployed using IaaS, it is difficult to design a holistic dependability solution that efficiently combines the failure behavior and system architecture of the application. This difficulty is due to *i)* high system complexity, and *ii)* abstraction layers of Cloud computing that release limited information about the underlying infrastructure to its users.

In contrast to the typical approach, the new dimension in which users' applications can obtain required dependability properties from a third-party might be beneficial. This approach of offering *dependability as a service* consists in realizing general dependability mechanisms as independent modules such that each module can transparently function on the user's application. Each module must be enriched with a set of metadata that characterizes its dependability properties, and the metadata be used to select mechanisms that satisfy users' requirements. This chapter presents a scheme that *i)* delivers a comprehensive dependability solution to user's application by combining selected dependability mechanisms and *ii)* ascertains the properties of a dependability solution by means of runtime monitoring. Based on this approach, the design of a framework that easily integrates with the existing Cloud infrastructure and facilitates a third-party in offering dependability as a service is also discussed.

4.1.1 Chapter Outline

The remainder of this chapter is organized as follows. Section 4.2 describes a motivating scenario and some basic concepts. Section 4.3 presents overview of an approach to resource management. Section 4.4 outlines a two-stage service delivery scheme that transparently offers dependability support to user's applications. Section 4.5 presents architectural details of the framework and Section 4.6 presents a chapter summary.

4.2 Motivating Scenario and Basic Concepts

This section describes a motivating scenario for the new dimension and presents some basic concepts on dependability.

4.2.1 Motivating Scenario

Consider a highly complex, service-oriented, and distributed infrastructure involving the following main stakeholders.

- *IaaS service provider:* builds a Cloud computing infrastructure, and realizes a service-oriented computing resources delivery scheme.

- *User:* deploys her applications using IaaS service provider's service. A user satisfies her reliability and availability requirements by leveraging the service offered by the dependability service provider.

- *Dependability service provider:* offers dependability support to users' applications based on a given set of requirements. We assume that the dependability service provider is trusted by both the IaaS service provider and the user.

For simplicity, when clear from the context, we refer the IaaS service provider as the *infrastructure provider*, and the dependability service provider as the *service provider*.

As an example, consider a user offering a web-based banking service which allows her customers to perform fund transfers and manage their accounts over the Internet. The user implements the banking service as a multi-tier application where: *i)* the data-tier uses the storage service offered by the infrastructure provider to store and retrieve her customer data, and *ii)* the application-tier uses the infrastructure provider's compute service to process operations and respond to customer queries. This system architecture allows the banking service to meet its varying business demands with respect to scalability and elasticity of computing resources. However, a failure in the infrastructure provider's system can have high implications on the reliability and availability of the banking service. Furthermore, a failure in the storage server may have a significantly higher impact than a failure in one amongst several compute nodes. This implies that each tier of the banking application requires different dependability properties, and the requirements may change over time based on the business demands. However, using existing methods, dependability properties of the banking service remains constant throughout its life-cycle. Therefore, in the user's perspective, it is easier to engage with the dependability service provider, specify her reliability and availability requirements based on the business needs, and transparently obtain desired dependability properties for her applications.

4.2.2 Basic Concepts

A user engages with the service provider to obtain dependability support for her applications. The goal of the service provider is to create a dependability solution based on the user's requirements such that a fine balance between the following factors is achieved.

- *Fault model:* It measures the granularity at which the dependability solution must handle errors and failures in the system. This factor is characterized by the mechanisms applied to achieve dependability, robustness of failure detection protocols, and strength of fail-over granularity.

- *Resource consumption:* It measures the amount and cost of resources that are required to realize a fault model. This factor is normally inherent with the granularity of the failure detection and recovery mechanisms in terms of CPU, memory, bandwidth, I/O, and so on.

- *Performance:* This factor deals with the impact of the dependability procedure on the end-to-end quality of service (QoS) both during failure and failure-free periods. This impact is often characterized using fault detection latency, replica launch latency and failure recovery latency, and other application-dependent metrics such as bandwidth, latency, and loss rate.

The most widely adopted strategy to tolerate failures in a system is based on the notion of redundancy. In redundancy based schemes, critical system components are duplicated using additional hardware, software, and network resources such that a copy of critical components is available after a failure happens. For example, the data-tier of the banking service can be replicated on several storage servers such that at least one copy of the data is always available to process customer queries. In general, a dependability algorithm that handles failures at a finer granularity, and offers high performance guarantees, consumes higher amount of resources. For instance, active replication methods in which all redundant components are simultaneously invoked, consume more resources than passive replication methods in which only one processing node handles the requests while other replicas are simple backups. However, passive replication techniques can only handle crash faults while active replication techniques using $3f+1$ replicas can be used to tolerate up to f arbitrary faults (e.g., [142, 26]). A detailed analysis of this behavior is provided in Section 5.2.4

It is clear that the dependability service provider must satisfy the following requirements to effectively realize its functionality and meet its business goals.

- The service provider must maintain a consistent view of the resources in the Cloud to efficiently deliver the dependability support to its users. To this aim, we must introduce a resource manager that is maintained by the dependability service provider in collaboration with the IaaS service provider (see Section 4.3).

- The service provider must develop: *i)* an approach to realize standard dependability algorithms that can extrinsically function on the users' applications, *ii)* a method to evaluate the dependability properties offered by a given mechanism and match it with users' requirements, and *iii)* a delivery scheme that can transparently enforce the desired dependability properties on users' applications (see Section 4.4).

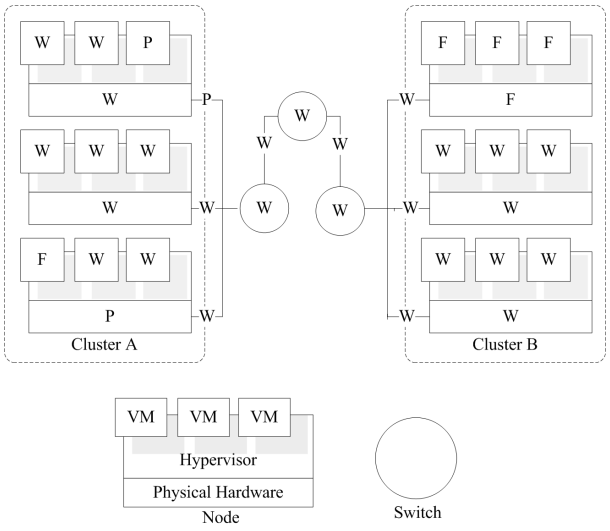


Figure 4.1: Graph generated by the Resource Manager

- The service provider must design a framework that can easily integrate with the existing Cloud infrastructure and meet service provider's goals (see Section 4.5).

4.3 Resource Manager

The dependability service provider must maintain a consistent view of all computing resources in the Cloud in order to efficiently allocate resources during each user request and avoid over provisioning during failures. In this context, a resource manager that continuously monitors the working state of physical and virtual resources, maintains a database of inventory and log information, and a graph representing the topology and working state of resources must be introduced by the dependability service provider in the IaaS provider's system.

The database of the resource manager must maintain the inventory information of each machine such as its unique serial number, composition of the machine (e.g., processor speed, number of hard disks and memory modules), date when the machine was commissioned (or decommissioned), location of the machine in the cluster, and so on. The runtime state of machines like memory used/free, disk capacity used/free and processor cores utilization must also be logged. On the other hand, a resource graph must represent the topology of resources in a system. Figure 4.1 represents the resource graph $G(\mathcal{H}, \mathcal{E})$ of a Cloud infrastructure where two clusters of three processing nodes each are connected by network switches. Here, the resource graph maintains details about the physical hosts, VM instances, and network links in the Cloud. W, P and F respectively represent the

working, partially faulty and completely faulty state of a resource. In the resource graph, each vertex represents a processing host $h \in \mathcal{H}$, and a network link between two hosts is represented as an edge $e \in \mathcal{E}$. A vertex also maintains information about the set of virtual machine (VM) instances hosted on that physical machine. In the service provider's point of view, a resource graph can represent the state of hosts and network links at different granularities. In a simple case, each host and link can be categorized in one of the three categories: working (W), partially faulty (P) and completely faulty (F). The resource manager marks the hosts (and links) with W when they exhibit a 'normal' condition i.e., operational with its full potential. A host (or link) is marked F if it has crashed or has incurred a major failure and cannot be recovered back to W. Partially faulty nodes, represented as P in the resource graph, are the ones where only a component of the host is not in use or is exhibiting a degraded performance (e.g., only the disk storage of the host is nonfunctional). Similarly, the working state of network links and VM instances must be maintained by the resource manager. We note that the database and the resource graph are essential for the service provider to ensure the correct behavior of dependability mechanisms. For example, a replication mechanism may have constraints on relative placement of individual replicas and requirements on resource characteristics of each replica which can be satisfied using the resource manager. We further note that the resource manager significantly contributes towards balancing the resource costs, performance, and fault model factors for the service provider. The above two aspects are discussed in detail in Sections 5.3.2, 5.4.2 and 5.4.3.

4.4 Dependability Delivery Scheme

The task of offering dependability as a service requires the service provider to implement general purpose dependability mechanisms in a way that the user's applications deployed in virtual machine instances can transparently obtain dependability properties. Let us define *dep_unit* as the fundamental module that applies a coherent dependability mechanism to a recurrent system failure at the granularity of a VM instance. The notion of dep_unit is based on the observation that the impact of hardware failures on user's applications can be handled by applying dependability mechanisms directly at the virtualization layer than the application itself (e.g., [29, 119]). For instance, dependability of the banking service can be increased by replicating the entire VM instance in which its application-tier is deployed on multiple physical nodes, and server crashes can be detected using well-known failure detection algorithms such as the heartbeat protocol. An example of a heartbeat protocol is depicted in Figure 4.2 where the primary and the backup components are run in VM instances independent to the banking service's application-tier. In this example, the primary component periodically sends a liveness request to all backup components and maintains a timer for each request. When a backup receives a liveness request, it immediately responds to the primary. If the backup fails (due to a server crash) to respond to the primary for

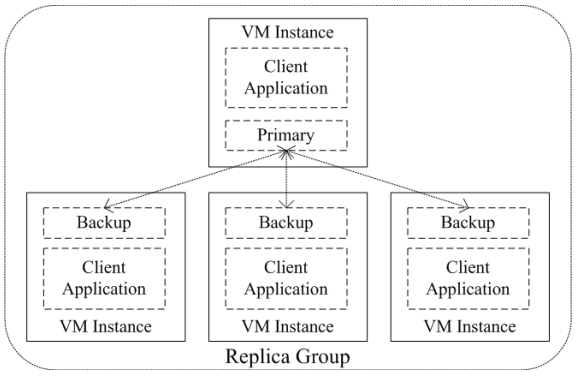


Figure 4.2: Dep_unit realizing the heartbeat based failure detection mechanism

N consecutive requests, each within a predefined timeout threshold, it is suspected to failure. In this context, we note that replication of the user's application (dep_unit1), and detection of node failures (dep_unit2) are performed without requiring any changes to the application's source code. This chapter assumes that the service provider implements a range of dependability mechanisms as dep_units, and based on this assumption, presents a two stage delivery scheme: design stage, and runtime stage, to transparently deliver high levels of dependability to user's applications using dep_units. This dependability delivery scheme is consistent with the dependability certification scheme discussed in Section 3.5.

4.4.1 Design Stage

The design stage starts when a user requests the service provider to offer dependability support to her application. In this stage, the service provider must first analyze the user's requirements, match them with available dep_units, and form a complete dependability solution using appropriate dep_units. We note that each dep_unit offers a unique set of dependability properties that can be characterized using its functional, operational, and structural attributes. Similarly to the dependability properties in the certification scheme (see Section 3.2), dependability property p of a dep_unit can be specified as p=(u, $\hat{p}$, A) where u represents the dep_unit, $\hat{p}$ denotes the abstract property, and A denotes a set of attributes that refers to the granularity at which u can handle failures, the benefits and limitations of using u, inherent resource consumption costs, and quality of service (QoS) parameters. For each attribute $attr \in A$, a partial (or total) order relationship can be defined on its domain $\mathcal{D}_{attr}$, and $v(attr)$ represents the value of $attr$. For instance, dependability property of a dep_unit u_1 can be denoted as p=(u_1, availability= 98.5%, {mechanism=active_replication, no_of_replicas=4, fault_model=node_crashes}). Therefore, by binding the abstract property $\hat{p}$ and the attributes set A to the dep_unit as its metadata, the service provider is facilitated in estimating the dependability properties

that can be obtained with its use. If the user's requirements are specified in terms of expected dependability properties p_c, the set of dep_units that matches p_c can be generated by including all dep_units for which $p_i.\hat{p}{=}p_c.\hat{p}$ and $v_i(attr){\geq}v_c(attr)$ for each attribute $attr_i{\in}A$ specified in p_i of u_i, that is, all dep_units that holds the properties desired by the user. We note that there is also an implicit hierarchy of dependability properties where $p_i{\preceq_P}p_j$ implies that p_j satisfies p_i. After shortlisting the dep_units that satisfy user's requirements, the task of the service provider is to compare each dep_unit within the shortlisted set and choose the one that best balances the fault model, resource costs, and performance with respect to p_c. As an example, let us consider that the service provider realizes three dep_units with properties: $p_1{=}(u_1$, {mechanism=heartbeat_test, timeout_period=50ms, $N{=}5$, fault_model=node_crashes, max_no_replicas=3}), $p_2{=}(u_2$, {mechanism=majority_voting, fault_model=programming_errors}), and $p_3{=}(u_3$, {mechanism=heartbeat_test, timeout_period=25ms, $N{=}3$, fault_model=node_crashes, max_no_replicas=5}) respectively. If the user requests a dependability support for her banking service with a more robust crash failure detection mechanism, the service provider first shortlists $S'{=}(u_1, u_3)$, then compares S' and finally makes use of u_3 dep_unit since u_3 is more robust than u_1. For simplicity, abstract properties are not specified in this example.

Although a dep_unit can serve as the fundamental dependability module for the service provider, a comprehensive dependability solution *dep_sol* that must be delivered to a user's application may be formed only by combining a set of dep_units in a specific execution logic. For example, a heartbeat test (dep_unit1) can be applied only after the user's application is replicated on multiple nodes (dep_unit2), and a recovery mechanism (dep_unit3) can be applied only after a failure is detected, that is, a comprehensive dependability solution that is finally delivered to the user is as follows:

dep_sol[
`invoke`:dep_unit(*VM-instances replication*)
`invoke`:dep_unit(*failure detection*)
`do`{
`execute`(*failure detection dep_unit*)
}`while`(*no failures*)
`if`(*failure detected*)
`invoke`:dep_unit(*recovery mechanism*)
]

By using dependability modules (dep_unit) to form a comprehensive solution, the dimension and intensity of the dependability support can be dynamically changed. In other words, the dependability properties applied on user's application can be adapted based on business needs to overcome the inflexibility of traditional dependability methods. For instance, a robust failure detection mechanism (such as u_3 in above example) can be replaced with a less robust one (u_1) in dep_sol. Furthermore, dep_units can flexibly and

extensively be reused for each user request saving significant amount of resources for the service provider, and by realizing dep_units to be configurable at runtime, resource consumption costs for users can be largely controlled. For example, by providing the parameters such as the number of replicas ($no\_of\_replicas$) for a dep_unit at runtime, the value $v(no\_of\_replicas)$=4 can be modified to $v(no\_of\_replicas)$<4 or $v(no\_of\_replicas)$>4 based on business demands. However, we note that a wide range of dep_units must be realized by the service provider to offer a higher quality of dependability support that precisely meets user's requirements.

4.4.2 Runtime Stage

The runtime stage starts immediately after the service provider forms a dep_sol and delivers it on the user's application. This stage is critical for efficient service delivery since the context and attribute values of a dependability solution may change at runtime due to the dynamic nature of the Cloud computing environment. In other words, the mutable behavior of dependability attributes requires the service provider to ascertain that the user's requirements are satisfied even during runtime. To achieve this, the service provider must first define a set $\mathcal{R}$ of rules over attributes $attr \in A$ and their values $v(attr)$ such that the validity of every rule $r \in \mathcal{R}$ establishes that property p is supported by the dependability solution and violation of a rule $r_i \in \mathcal{R}$ implies that p is invalid. For instance, for a comprehensive dependability solution s_1 that holds the property p_1=(s_1, availability=98%, {mechanism=active_replication, level=3, failure_detection=heartbeat_test, max_recovery_time=25ms}), a set of rules $\mathcal{R}$ that can sufficiently test the validity of p_1 must be defined, such as, r_1:no_of_server_instances$\geq$3, r_2:heartbeat_test_frequency=5ms, r_3:recovery_time$\leq$25ms. In this context, the task of the service provider is to continuously monitor the attribute values of each dependability solution delivered to the user's application at runtime, and verify the corresponding set of rules $\mathcal{R}$ to ensure that user's requirements are satisfied. We note that the service provider can obtain attribute values by periodically querying each dep_sol s delivered to a user's application. Here, a dependability property can be represented as $p^t=(s, \hat{p}, A^t)$ where t denotes the point of time at which the attribute value is queried, $v^t(attr)$ is the value of attribute at t, and s is the comprehensive dependability solution. The service provider can define a validation function $f(s, \mathcal{R})$ that takes s and the corresponding $\mathcal{R}$ as input and outputs $true$ if $v^t(attr) \geq v_i(attr)$ for each attribute $attr \in A$, that is, $f(s, \mathcal{R})$ verifies whether the dependability solution remains valid and satisfies the user's requirements at runtime. In case, $f(s, \mathcal{R})$ returns $false$, the service provider must either trigger the matching and comparison process of the design phase to select a new set of dep_units and form a new dep_sol that best matches user's requirements. Therefore, by constantly monitoring each dep_sol and by updating the attribute values, the service provider can deliver a dependability support that is valid throughout the life-cycle of the user's application (initially during request time and at runtime). Furthermore, a change in the user's requirements at

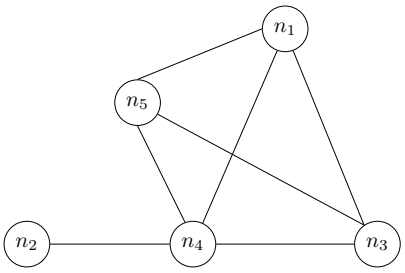


Figure 4.3: An example of resource graph generated by the Resource Manager

any stage also triggers the design phase to form a new dependability solution.

4.5 Dependability Manager: Architecture Framework

This section presents a conceptual framework, the Dependability Manager (DM), that provides the basis for the service provider to realize the delivery scheme presented in the previous section and hence to offer dependability as a service. This framework should be inserted as a dedicated service layer between the user's applications and the hardware, so that, it works directly on top of the virtual machine manager at the level of VM instances. The dependability manager must address the issue of heterogeneity in computing resources, fulfill the target of transparently providing dependability support to user's applications against node failures, and satisfy scalability and interoperability goals. To overcome these challenges, the dependability manager is built using the principles of service-oriented architectures, where each dep_unit is realized as an individual web service, and a dep_sol is formed using the business process execution language (BPEL) constructs [6]. A resource manager that coordinates with the cloud manager to produce the resource graph and the database discussed in Section 4.3 is included within the DM. The resource manager is realized in the form of a web service that provides a `status` operation which takes a resource (e.g., processing node, storage, memory) as input and outputs the state of that resource. Note that `status` operation can be run independently on each node and, using the `update` operation, state of the resource can be updated in the database and resource graph. As an example, let us consider that at the start of the service invocation, the service provider generates a profile of computing resources in the cloud infrastructure by identifying five processing nodes $\{h_1, \ldots, h_5\} \in \mathcal{H}$ whose resource graph is presented in Figure 4.3. A description of all the components in the framework is provided further in this section.

4.5.1 Client Interface

The service invocation process begins when a user requests the service provider to offer dependability support to its application with a desired set of properties. In this context, it is essential to include a client interface component within DM that provides a specification language which allows users to specify and define their requirements (e.g., [84], [76]). However, since the present day cloud computing systems require its users to manage their VMs while dealing with sophisticated system-level concerns, an automated configuration tool that requires users to simply select the application for which they wish to obtain dependability support, and correspondingly provide values of desired availability, reliability, response time, criticality of the application and cost can be beneficial. We note that an automated configuration tool can limit human error and save time by lessening the need for manual tedious configuration. Moreover, if the input can be provided in a high level format (like percentages, range and numbers), even users with a non-technical background can configure the desired properties with ease. The aspect of transforming high level metric values into dependability properties and standard dependability mechanisms, and an algorithm matching high level user requirements to available dep_sols in the system is discussed in Section 5.2.

4.5.2 DMKernel

The central computing component of the dependability manager is the DMKernel which is responsible for composing a dependability solution based on user's requirements using the web service modules (dep_units) implemented by the service provider, delivering the composed service on user's applications, and monitoring each service instance to ensure its QoS. DMKernel is composed of a service directory, a composition engine, and an evaluation unit.

- *Service Directory:* It is a registry of all dep_units realized by the service provider in the form of web services. A dep_unit applies a dependability mechanism as a self-contained, loosely coupled module, with a well-defined language-agnostic interface that *i)* describes its operations and input/output data structures (e.g., WSDL and WSCL), and *ii)* allows other dep_units to coordinate and assemble with it. In addition to the dep_units, this component also registers the metadata that represents the dependability property $p{=}(u, \hat{p}, A)$ of each dep_unit. When DM receives input from the client interface, this component first performs a matching between the user's preferences p_c, and properties p_i of each dep_unit in the service directory, to generate the set of dep_units that satisfy p_c. The set of services is then ordered based on user's preferences and provided to the composition engine. The service directory triggers the matching and comparison processes at runtime if the evaluation unit updates the metadata of a dep_unit. However, we note that the service provider

must perform an a priori validation of all its dep_units and estimate their properties p in the infrastructure provider's system as a prerequisite.

- *Composition Engine:* It receives an ordered set of dep_units from the service directory as input, and generates a comprehensive dependability solution dep_sol using the web services (dep_units) that best match user's preferences as output. In terms of service oriented architectures, the composition engine can be viewed as a web service orchestration engine that exploits BPEL constructs to build a composed dependability solution that is delivered to user's application using robust message exchanges protocols (e.g., [138]) as presented in Section 4.4.

- *Evaluation Unit:* It continuously monitors all composed dependability solutions at runtime using the validation function $f(s, \mathcal{R})$ and the set of rules $\mathcal{R}$ defined corresponding to each dep_sol. We note that the interface exposed by web services (e.g., WSDL and WSCL) allows the evaluation unit to validate all the rules $r \in \mathcal{R}$ during runtime monitoring. If $f(s, \mathcal{R})$ returns $false$, the evaluation unit updates the present attribute values in the metadata, otherwise, the service continues uninterrupted.

DMKernel can measure the overall reliability of the service provided to the user's application by comparing a set of metrics (such as, mean time between failure MTBF) between the real time operational data obtained from the resource database, and expected values of the metrics obtained from the input using client interface. For example, a user's request for 99% availability of its application implies that DM must ensure that the MTBF of the node where the application is deployed is at least $avail_{exp} = 0.99 * t$ for a given time period t. Since each node failure is logged in the database, the operational $avail_{real}$ value can be calculated, and the strength of the service provided by DM by measuring $avail_{exp} - avail_{real}$.

In addition to the resource manager, client interface, and DMKernel, we note that the DM must include a set of components that provide a complementary support to dependability mechanisms. These components significantly affect the quality of service offered by the service provider, and are essential to satisfy user's requirements and constraints. In particular, one must include the following components in DM. Figure 4.4 illustrates the overall architecture of the dependability manager.

- *Replication Manager:* It provides support to dep_units that realize replication mechanisms by managing the details regarding individual replicas of a user's application, their location, and synchronization process between them (see Section 4.5.3).

- *Fault Detection/Prediction Manager:* It provides support to techniques that either detect or predict failures among the nodes (see Section 4.5.4).

- *Fault Masking Manager:* It comprise modules which support techniques that are used to mask the presence of faults in the system. (see Section 4.5.5).

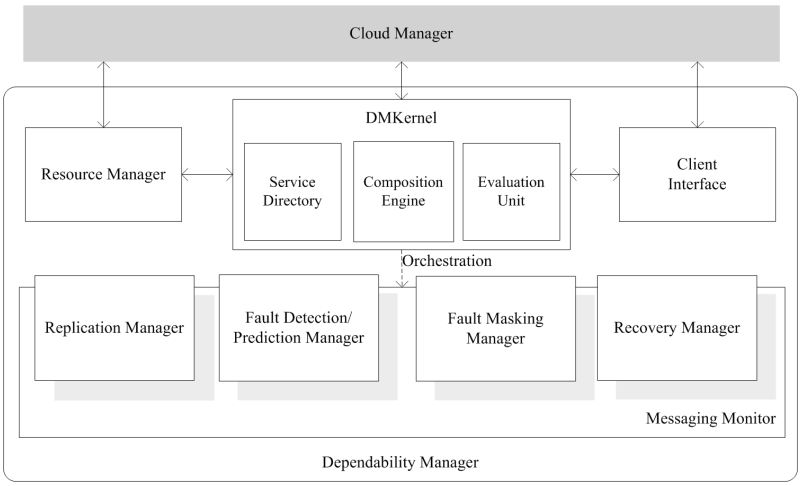


Figure 4.4: Architectural overview of the Dependability Manager

- *Recovery Manager:* It includes services that support dep_units that recovers a faulty node back to operational (see Section 4.5.6).
- *Messaging Monitor:* It provides the infrastructure necessary for communication among all the components of DM (see Section 4.5.7).

A detailed description of these components is provided in the following sections.

4.5.3 Replication Manager

This component supports the replication mechanisms by invoking replicas and managing their execution based on the user's requirements. Let us denote the set of VM instances that are controlled by a single implementation of a replication mechanism (dep_unit) as a replica group. Each replica within a group can be uniquely identified, and a set of rules $\mathcal{R}$ that must be satisfied by a replica group are specified. The task of the replication manager is to make the user perceive a replica group as a single service, and to ensure that the fault free replicas exhibit correct behavior during execution time.

Figure 4.5 provides an overview of various components within the replication manager and their interactions with each other. To support a replication mechanism, the replica invoker first contemplates the desired replication parameters such as the style of replication (active, passive, cold passive, hot passive), number of replicas, and constraints on relative placement of individual replicas, and forms the replica group. In other words, the replica invoker takes the reference of a user's application as input from DMKernel, analyzes the expected dependability properties, and interacts with the resource manager to obtain the location of each replica. The replica group manager then creates the replica group by

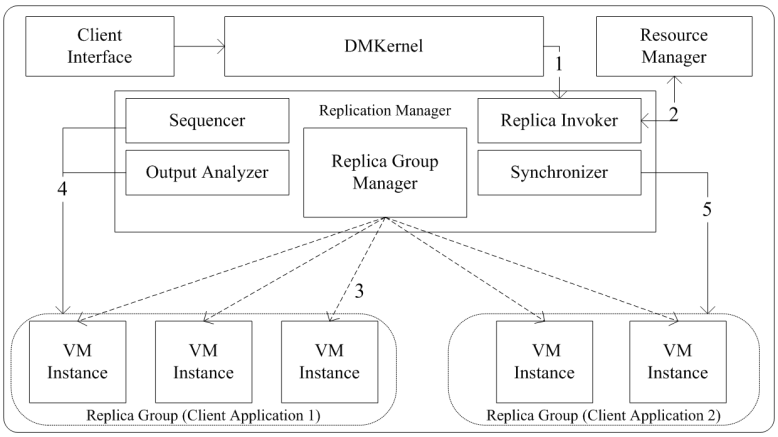


Figure 4.5: Architectural overview of the Replication Manager

The figure shows various components of the replication manager, and their interaction with other components of the framework. The straight lines with numbers describe the replication process; the operations at each step is correspondingly explained in Section 4.5.3. The dotted lines represent the interaction with the VM instances that are external to the framework.

invoking VM instances at those locations and managing their execution. The sequencer provides the input to application executing in the replica group by means of consensus protocol (e.g., [26, 93]) in order to ensure determinism among replicas. The output analyzer carries out majority voting on the responses obtained, and returns the chosen result to the user. The synchronizer includes techniques to update the state of backup replicas with that of the primary in a replica group. It also supports membership change and primary election algorithms when the primary node undergoes failure. We note that robustness of these procedures largely contribute to the consistency and reliability of the service.

Example 4.5.1. *Let us consider that DMKernel chooses a passive replication mechanism corresponding to the banking service's request where the following constraints must be satisfied:* i) *the replica group must contain one primary and two backup nodes at all times,* ii) *the node on which the primary executes must not be shared with any other VM instances, and* iii) *all the replicas must be located on different nodes. For the Cloud infrastructure depicted in Figure 4.3, the replication manager forms a replica group of the banking service's application by choosing the host h_1 for the primary, and hosts h_3 and h_4 respectively for backup replicas. We note that hosts h_3 and h_4 can deploy VM instances of other replica groups while only one VM instance can run on host h_1. The synchronizer of replication manager frequently checkpoints the primary and updates the state of backup replicas.*

4.5.4 Fault Detection/Prediction Manager

This component enriches the DM by providing failure detection support at two different levels. The first level is infrastructure-centric, and provides failure detection globally across all the nodes in the Cloud, whereas, the second level is application-centric, and provides support only to detect failures among individual replicas within a replica group. To realize failure detection at either levels, we note that this component must support several well known failure detection algorithms (e.g., the gossip based protocol, and heartbeat protocol) that are configured at runtime based on replication mechanism and user's requirements.

When the replication manager successfully creates a replica group, the composition engine invokes dep_units to detect/predict failures within the replica group. An example of a failure detection dep_unit (heartbeat protocol) is presented in Section 4.4. The main goal of the failure detection/prediction manager is to support DM in detecting faults immediately after their occurrence, and sending a notification about the faulty replica to the fault masking manager and the recovery manager. For infrastructure-centric failure detection, failure notifications are sent to the resource manager to update the resource state of the cloud that is utilized to predict failures in a proactive dependability approach. We note that most failure detection protocols that are exploited in a passively replicated system perform well in detecting major failures. However, to detect errors at smaller granularity resulting from a partially faulty node, active replication methods need to be deployed. For example, programming errors in user's application can be detected by applying a majority voting using the output analyzer of the replication manager on the output generated by each active replica.

Example 4.5.2. *For the replica group of the banking service's application described in Example 4.5.1, suppose that the service directory selects a dep_unit that realizes a proactive dependability mechanism. This implies that the failure detection/prediction manager must continuously gather the state information of hosts h_1, h_3, and h_4, and verify if all system parameter values are over a certain threshold (e.g., physical memory usage of a node allocated to a VM instance must be less than 70% of its total capacity).*

4.5.5 Fault Masking Manager

The goal of this component is to support dep_units that realize fault masking mechanisms so that occurrence of faults in the system can be hidden from users. When a failure is detected in the system, this component immediately applies masking procedures to prevent faults from resulting into errors. We note that the functionality of this component is critical to meet user's high availability requirements.

Example 4.5.3. *From Example 4.5.2, let us consider that the failure detection/prediction manager predicts a failure in host h_3 and immediately invokes the fault masking dep_unit.*

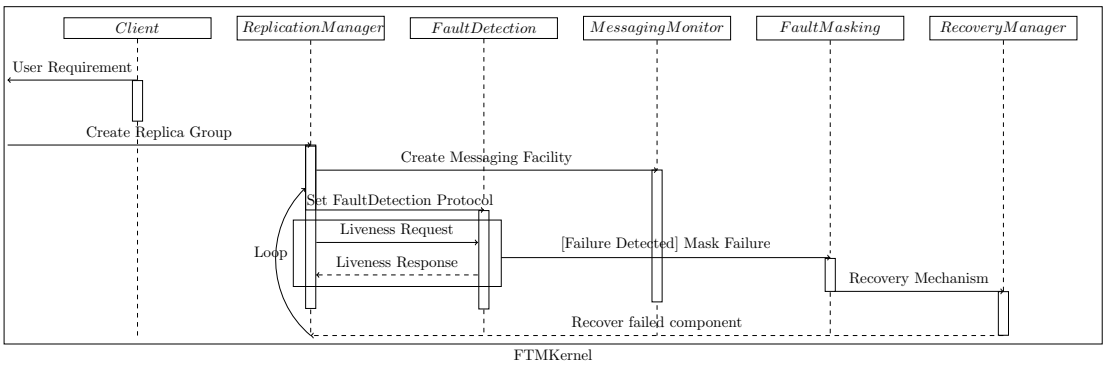


Figure 4.6: An example of a workflow, represented as a sequence diagram illustrating the interaction among all the components of the DM for a single user request

Here, the dep_unit performs a live migration of the VM instance (e.g., [29, 119]) such that the entire OS at host h_3 is moved to another location (host h_5) while maintaining the established session so that the customers of the banking service do not experience any impact of the failure at host h_3. Therefore, user's high availability requirements can be fulfilled using the fault masking mechanisms.

4.5.6 Recovery Manager

The goal of this component is to achieve system-level resilience by minimizing the down-time of the system during failures. To this aim, this component supports dep_units that realize recovery mechanisms so that an error-prone node can be resumed back to a normal operational mode. In other words, this component provides support that is complementary to that of the failure detection/prediction manager and fault masking manager, especially in the condition when an error is detected in the system. We note that DM maximizes the lifetime of the Cloud infrastructure by continuously checking for occurrence of faults using the failure detection/prediction manager and, when exceptions happen, by recovering from failures using the recovery manager.

Example 4.5.4. *As described in the Example 4.5.3, using fault masking manager the high availability goals of the user can be met even when a failure happens at host h_3. However, the service offered by the infrastructure provider may be affected since the system consists of only four working nodes. In this context, it is critical for the infrastructure provider to apply robust recovery mechanisms in order to increase its system's lifetime. The support offered by the recovery manager resumes host h_3 (that is marked with F or P in the resource graph) to working state (W in resource graph).*

4.5.7 Messaging Monitor

Messaging monitor extends through all the components of our framework (as shown in Figure 4.4) and offers the communication infrastructure in two different forms: message exchange within a replica group, and inter-component communication within the framework. Since dep_units, and other components in DM are realized as web services, the communication between any two components (and the replicas) must be reliable even in the presence of component, system or network failure. To this aim, messaging monitor integrates WS-RM standard [138, 12] with other application protocols and establishes an appropriate messaging infrastructure that supports the composition engine in designing a robust dep_sol. We note that this component is critical in providing maximum interoperability, and serves as a key QoS factor.

Example 4.5.5. *Based on Examples 4.5.1 to 4.5.4, here, a summary of the implicit workflow that happens across all the components of DM as a response to a single user request is presented. As shown in Figure 4.6, the service invocation process starts when DMKernel gathers the user's requirements from the client interface using the <`receive`> BPEL activity. Based on user's requirements, DMKernel first selects appropriate web service modules (dep_units) from the service directory, and the composition engine defines (dep_sol) an execution logic among selected web service modules. DMKernel delivers the dep_sol by first triggering the replication manager to create a replica group for the user's application. In terms of BPEL language constructs, activity <`invoke`> is specified. Once the replica group is created, DMKernel triggers the messaging monitor and the failure detection/prediction manager to create a messaging infrastructure and to invoke fault prediction protocol respectively, using the <`flow`> activity, so that both dep_units run continuously in parallel. The dep_unit associated with the fault masking manager is triggered immediately (<`if`> activity) after a failure is predicted, and finally the recovery manager is invoked (using <`invoke`> activity) to recover the failed node. We note that throughout the workflow, the evaluation unit monitors the service instance to ascertain that DM satisfies the user's requirements and maintains the QoS.*

4.6 Chapter Summary

In this chapter, we presented a comprehensive view to transparently deliver dependability to applications deployed in virtual machine instances. In particular, we presented an innovative approach for realizing generic dependability mechanisms as independent modules, validating dependability properties of each mechanism, and matching user's requirements with available dependability modules to obtain a comprehensive solution with desired properties. The proposed approach when combined with our delivery scheme enables a service provider to offer long-standing dependability support to users' applications. Furthermore,

we discussed the design of a framework that allows the service provider to integrate its system with the existing Cloud infrastructure and provides the basis to generically realize our approach in delivering dependability as a service. We note that the components of the proposed framework can be extended in a straight-forward manner to improve the overall resilience of the Cloud infrastructure.

The approach discussed in this chapter relieves users (e.g., SaaS providers) from implementing low-level dependability mechanisms, and improves the reliability and availability of their applications (e.g., eCommerce service) when deployed using IaaS services. This approach satisfies specific dependability requirements of individual applications and offers significantly higher level of dependability when compared to simple certification schemes (second level of dependability offered in this thesis).

In the next chapter, we discuss a set of techniques that allow the dependability service provider to realize components of the Dependability Manager and deliver its service.

5

Supporting the notion of Dependability as a Service

In this chapter, we present a set of techniques that together realize the notion of offering dependability as a service to users' applications. We divide the overall problem into four sub-problems and present an approach to solve each sub-problem. First, we present an approach to analyze the failure behavior of the envisioned Cloud and dependability properties that can be obtained using a given dependability mechanism, and use this analysis to translate high level users' requirements to low level mechanisms. Second, we present an approach to translate specific configuration requirements of low level dependability mechanisms to resource level constraints. Third, we present an approach to allocate virtual machine instances while satisfying dependability constraints, to finally deliver the chosen service. Finally, we present a resource management scheme that balances users' performance and dependability requirements by adapting the current resource allocation of her applications whenever her requirements not satisfied due to various system changes.

Given the goal of this thesis is to improve the dependability of Cloud computing, particularly by reducing the risks of using IaaS and SaaS services, the solutions for the aforementioned sub-problems improve the reliability and availability of users' applications deployed using IaaS services, and consequently, provide higher level of dependability when compared to certification scheme (second level of dependability offered in this thesis).

5.1 Introduction

In Chapter 4, we advocated the approach of offering dependability as a service wherein users can acquire desired dependability properties for their applications from a third party. This approach allows users to implement highly available and reliable applications without having to implement low level dependability mechanisms. The design of the conceptual framework consisting of all the components necessary to realize the new approach is discussed in the previous chapter as the Dependability Manager (DM). This chapter divides the overall problem of offering dependability as a service into the following four sub-

problems, and presents an approach to solve each sub-problem. These four activities, when integrated together, realize the functionality of the DM and its components.

1. **Mapping high-level user requirements to low-level dependability mechanisms and their specific configuration.** The DM implements a range of dependability mechanisms as independent modules dep_units, and based on user's requirements, selects appropriate modules to compose a dependability solution dep_sol. This requires the service provider to measure the effectiveness of each dependability mechanism, in different configurations, considering the failure characteristics of the Cloud infrastructure. One approach to measure the effective of a given mechanism is to estimate the level of reliability and availability that can be obtained with its use. This approach must take into account the failure behavior of the infrastructure components, the correlation between individual failures, and the impact of each failure on the user's applications. Finally, using this analysis, the service provider can define a search algorithm to identify the mechanisms that satisfy user's requirements.

2. **Specifying the configuration of dependability mechanisms in terms of resource-level constraints.** The service provider must enforce the deployment conditions inherent to the configuration of the selected dependability solution dep_sol in order to correctly deliver the service. For example, if the selected dep_sol improves availability by replicating application tasks, then, the service provide must ensure that each replica is allocated on a different physical host in the infrastructure so as to avoid a single point of failure. This requires the service provider to specify and integrate the restrictions imposed by a given dep_sol within the resource allocation algorithm of the IaaS service provider. These restrictions may concern the dependability and performance of users' applications, and are critical to the overall service.

3. **Satisfying resource-level constraints while allocating VM instances.** Typically, the IaaS service provider uses a heuristics-based algorithm to allocate the virtual machine instances requested by the user. The objective of such heuristics is to maintain high QoS while increasing the monetary profits of the service. For example, allocation algorithms are often designed to reduce the energy consumption costs of the infrastructure, thus improving economic returns.

   The dependability service provider is the intermediary between the users and the Cloud service provider and, as a consequence, it must implement several system-level algorithms that allows it to *i)* offer its service while providing sufficient abstraction from low-level details to the users, and *ii)* support the Cloud service provider in achieving its business goals. This is in contrast to the dependability certification scheme of Chapter 3, where the certification authority is required to validate the effectiveness of the mechanisms implemented by the user (considering the client's service as an encapsulated package).

Hence, the service provider requires an algorithm that allocates virtual machine instances on the Cloud infrastructure while satisfying all the restrictions inherent to the selected dependability solution dep_sol (activity 2) and supporting IaaS service provider's objectives. The result of successfully executing this algorithm is that the dep_sol can actually deliver its functionality to the user's application (completing the design stage of the DM).

4. **Adaptive resource management to ensure users' requirements at runtime.** Cloud computing environment is highly dynamic and requires runtime monitoring of the delivered service. When system changes affect the desired dependability output (e.g., reduction in the availability of user's application due to a server crash), the affect of such changes needs to masked, requiring the service provider to implement an algorithm that adapts the current allocation of the application. This algorithm ensures that the service provider delivers solution satisfying user's requirements also during runtime.

5.1.1 Chapter Outline

The remainder of this chapter is organized as follows. Section 5.2 presents an approach to quantify the effectiveness of dependability mechanisms using fault trees and Markov models. It also presents a search algorithm that selects dependability mechanisms that most appropriately match user's requirements. Section 5.3 investigates and formulates different constraints that both dependability service provider and IaaS service provider may wish to specify. These constraints impose restrictions on the allocations to be made to the hosts and express conditions on the placement, and relative placement, of virtual machines. Section 5.4 presents a virtual machine provisioning algorithm that satisfies all allocation constraints, allowing the service provider to effectively deliver its service to the users. It also describes an adaptive resource management algorithm that ensures user's dependability and performance requirements during runtime. Section 5.5 provides some simulation results. Section 5.6 provides chapter summary and some concluding remarks.

5.2 Mapping Users' Requirements to Dependability Mechanisms

The first step to realize the notion of dependability as a service is to design a mechanism that allows users to specify their requirements with ease, and a scheme that matches user's high-level requirements with low-level dependability techniques. In other words, as the first step, there is a need to build the functionality of the Client Interface, Service Directory, and Composition Engine components of the DM (see Section 4.5). To achieve this, the service provider requires the ability to *i)* identify the failure characteristics of the Cloud infrastructure, *ii)* quantify the reliability and availability obtained by each

implemented mechanism (dep_units and dep_sols), and *iii)* evaluate different configurations of various dependability mechanisms in order to suitably deliver its service to the user. The remainder of this section first discusses an approach that realizes the three aforementioned abilities, and then defines a dependability policy selection scheme.

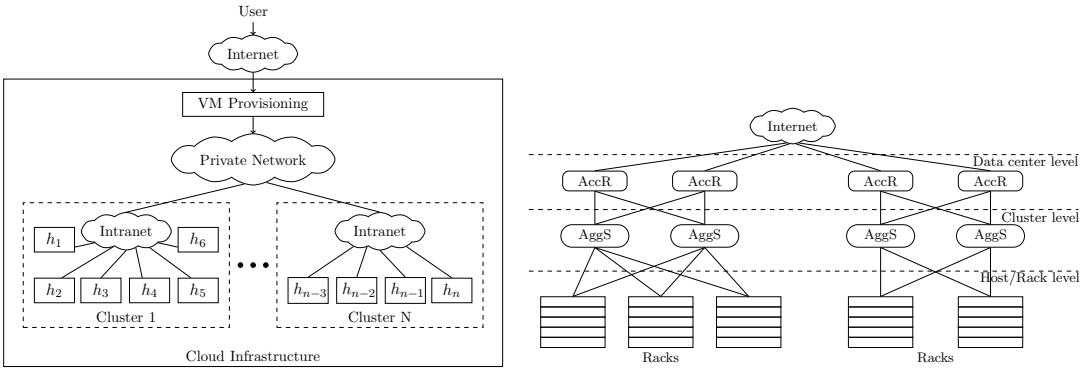
5.2.1 Analysis of Failure Characteristics of System Components

A Cloud computing user must engage with the service provider to obtain dependability support for her applications. The goal of the service provider is to create a dependability solution based on user's requirements and deliver the solution by taking into account the failure characteristics of the Cloud infrastructure. This section provides an overview of a typical Cloud infrastructure and derives an approach to characterize failures in the system.

Overview of the Cloud infrastructure

Cloud computing infrastructure can be viewed as a large pool of interconnected physical hosts $\mathcal{H}$ that is partitioned into a set $\mathcal{C}$ of clusters. A cluster $C \in \mathcal{C}$ can be formed by grouping together all the hosts that have identical resource characteristics or administrative parameters (e.g., hosts that belong to the same network latency class or geographical location). Each host or server contains multiple processors, storage disks, memory modules and network interfaces. Hence, the resource characteristics of each physical host $h \in \mathcal{H}$ can be represented using a d dimensional vector $\overrightarrow{h} = (h[1], h[2], \ldots, h[d])$, where each dimension represents the amount of host's residual capacity (i.e., resources not yet allocated) corresponding to a distinct resource type (e.g., CPU, memory, storage, network bandwidth). For simplicity, the resource capacity of hosts can be denoted using normalized values, say, between 0 and 1. For example, the host h characterized as $\overrightarrow{h}$=(CPU, Mem)=$(0.6, 0.5)$ implies that 60% of CPU, 50% of memory on h is available for use.

A hypervisor is deployed on each host to virtualize its resources, and required amounts of computing resources are delivered to the user in the form of virtual machine instances. All the hosts are connected using several network switches and routers. In particular, as described in [51], we consider that hosts are first connected via a 1Gbps link to a Top of Rack switch (ToR), which is in turn connected to two (primary and backup) aggregation switches (AggS). The subsystem formed by the group of servers under an aggregate switch can be viewed as a cluster. An AggS connects tens of switches (ToR) to redundant access routers (AccR). This implies that each AccR handles traffic from thousands of servers and route it to core routers that connect different data centers to the Internet. The left portion of Figure 5.1 illustrates an example of a Cloud infrastructure consisting of N clusters of interconnected hosts, and each cluster is connected through a network that is private to the service provider. The right portion of the figure illustrates a part of the data center network architecture.



(a) Example Cloud infrastructure (b) Network architecture

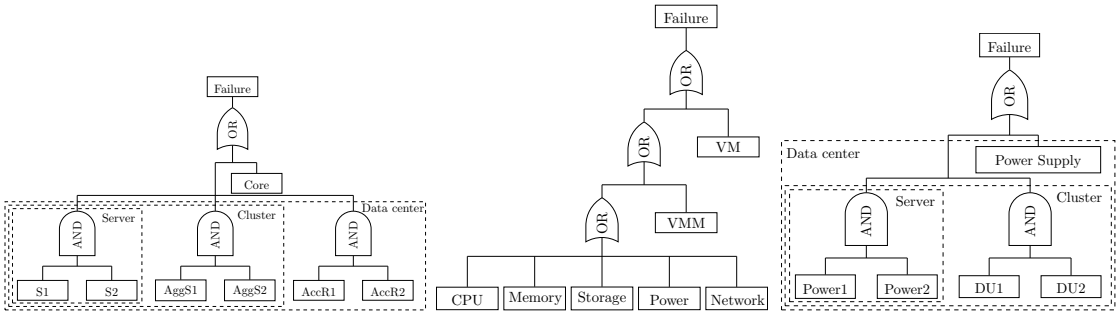
Figure 5.1: Cloud infrastructure showing different deployment levels

Failure behavior of system components

Failure behavior of the system must be modeled in the service provider's perspective, that is, the infrastructure component failures which result in a user application failure must be implicitly represented. The failure behavior is modeled here using the notion of fault trees [131, 39] since the dependence between individual failures in the Cloud infrastructure and the boundaries on the impact of each failure can be taken into account. The failures on three main types of resources (network, server and power) are discussed in the following.

- *Network*: Figure 5.2(a) represents the fault tree for an application considering the network failures in the infrastructure, based on the network architecture described in Section 5.2.1. The system fails when the top-event value in the fault tree is `true`. In this context, a failure implies that the application is not connected to the rest of the network or gives errors during data transmission. We note that the fault tree clearly defines the boundaries on the impact of network failures (using server, cluster and data center level blocks), and allows the service provider to increase the dependability of user's applications (e.g., by placing individual replicas of an application in different failure zones). The concept of failure zones is described in detail in Section 5.2.3 in terms of deployment levels. A network failure happens if there is an error in all redundant switches ToR, AggS, AccR or core routers, or the network links connecting the physical host and other network components.

- *Server*: An application deployed in a virtual machine instance that is hosted on a server may fail if there is a failure in the physical host or the management software. In other words, a failure/error either in the *i)* processor, memory modules, storage disks, power supply or network interfaces, or *ii)* the virtual machine manager (VMM), or *iii)* the virtual machine (VM) instance itself, may lead the application to a failure.



(a) Network failure (b) Server failure (c) Power failure

Figure 5.2: Fault tree models for server, network and power failures

Figure 5.2(b) illustrates this behavior as a fault tree where the top-event represents the failure in user's application (i.e., when the top-event's value is `true`). Service provider can determine the reliability and availability of a server and its components using Markov models (see Section 5.2.2).

- *Power*: We assume that a data center receives the power via an uninterrupted power network, and a redundant distribution unit (DU) is deployed for each cluster within the data center. A DU provides power to all the servers within a cluster. A failure in the DU is independent of other DUs and the central power supply. Figure 5.2(c) depicts the fault tree of power failures in a Cloud infrastructure.

This method can be extended to incorporate other failures (e.g., cloud manager errors) in a straightforward manner, and are not discussed here. Failure characteristics and fault trees are used in Section 5.2.3 to select appropriate deployment configuration for the dependability mechanism applied on a given users' application.

5.2.2 Analysis of Dependability Metrics

This section discusses representative dependability mechanisms that can transparently handle component failures in a Cloud infrastructure.

Dependability Mechanisms

The task of offering dependability as a service requires the service provider to realize dependability mechanisms that can transparently function on user's applications. In this context, a dep_unit defines an independent module that applies a coherent dependability mechanism to a recurrent set of system failures at the granularity of a VM instance (see Section 4.4). The notion of dep_unit is based on the observation that the impact of hardware failures on user's applications can be handled by applying dependability mechanisms

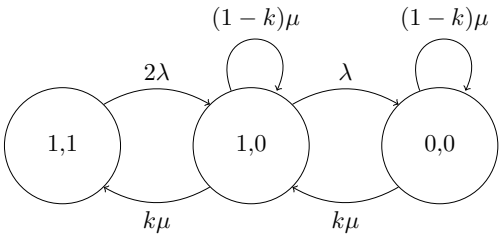


Figure 5.3: An example of a Markov model for semi-active replication

directly at the virtualization layer than the application itself. A brief discussion on three main configurations of dep_sols that use dep_units realizing replication schemes is provided below. These dep_sols represent the majority of dependability implementations that are currently being used.

Semi-active replication. The input is either provided to all the replicas or state information of the primary replica is frequently transmitted to the backup replicas. The primary as well as the backup replicas executes all the instructions, but only the output generated by the primary replica is made available to the user. The output messages of backup replicas are logged by the hypervisor. In case the primary replica fails, one of the backup replicas can readily resume the service execution. For each replica failure, the DM must create an equivalent replica (VM instance) on another host and update its state. We note that in a cloud computing environment, resources are often over-provisioned, and hence it is possible to create backup resources with a very high probability. An example of a technique that falls in this category is the VMware's dependability [134] that is designed for mission-critical workloads. We note that the availability obtained by using this technique is very high, but it comes at high resource consumption costs.

As discussed in the previous section, Markov models can be used to determine the reliability and availability of the application that uses this replication scheme because failure behavior of physical hosts (servers) can be taken into account. Figure 5.3 depicts the Markov model of a representative dep_sol that is based on semi-active replication scheme with two replicas (λ is the failure rate and μ is the recovery rate). Each state is represented as (x, y) where x=1 implies that the primary replica is working and x=0 implies that it failed. Similarly, y represents the state of the backup replica. The system starts and remains in state (1,1) during normal execution, i.e., when both replicas are available. When a VM instance (either primary or backup replica) fails, the system moves to state (0,1) or (1,0) where other replica takes over the execution process. We note that a single state is sufficient to represent this condition in the Markov model since, in the service provider's perspective, both the replicas are equivalent. In state (0,1) or (1,0), DM initiates the recovery mechanism defined in the dep_sol, and the system moves to state

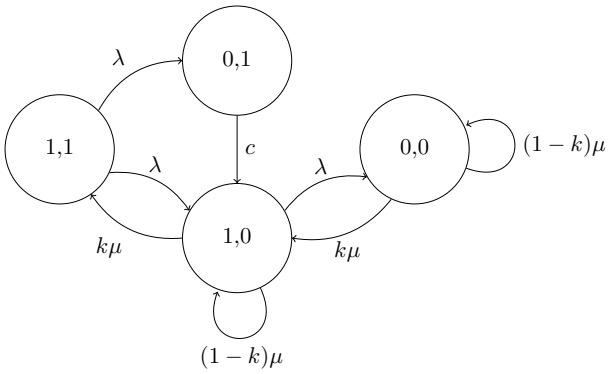


Figure 5.4: An example of a Markov model for semi-passive replication

(1,1) if the recovery is successful; if the server experiences a failure, the system transits to state (0,0) where the user's application becomes unavailable.

Semi-passive replication. The state information is obtained by frequently checkpointing the primary replica and buffering the input parameters between each checkpoint, and replication is performed by transferring the state information to the backup replicas. The backup replicas do not execute the instructions but saves the latest state obtained from the primary replica. In case the primary replica fails, a backup replica is initiated and updated to the current state with some loss in the present execution cycle and reasonable downtime. Remus [29] is a typical example of a system that is used by the Xen hypervisor and realizes a configuration of semi-passive replication. We note that the availability obtained from this technique is less than that by semi-active replication, but the resource consumption costs are reduced since the backup replicas do not execute instructions.

Figure 5.4 represents the Markov model of an application for which the semi-passive replication mechanism with two replicas is applied by the service provider. In this model, when a failure in the primary replica happens, the system moves from state (1,1) to state (0,1) and begins the update process. The backup replica assumes the execution process (becomes the new primary replica) and the system implicitly moves to state (1,0). In this state, DM invokes a new replica and provides it with the latest checkpoint. If the new backup replica is successfully commissioned, the system again moves to state (1,1), otherwise it remains in state (1,0). A failure in the primary replica in state (1,0) results in a complete system failure (i.e., both replicas become unavailable), and the system transits to state (0,0).

Passive replication. The state information of a VM instance is regularly stored on a backup. In case of a failure, DM recommissions another VM instance and restores the last saved state. We note that a backup can be configured to share the state of several VM instances or it can be dedicated to a particular application, and the VM

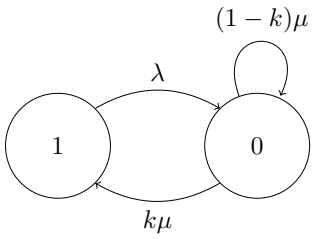


Figure 5.5: An example of a Markov model for passive replication

recommissioning process can be performed based on a priority value assigned to each VM instance. VMware's High Availability solution [133] is a typical example of this replication technique. This approach consumes least amount of resources but provides reduced availability than the former methods. Figure 5.5 illustrates the Markov model of an application for which a dedicated (passive) backup is applied.

A dep_sol can perform replication of a user's application, detection of failures, and recovery from a failed state without requiring any changes to the application's source code. This implies that it is feasible for the service provider to transparently enforce dependability on specified applications. For the sake of simplicity, only the availability property of the system is discussed in the above examples, but similar Markov models can be used to study the reliability property as well.

We note that, based on the system design, dependability of a given application also depends on the number of replicas of each application task as well as the location of each replica in the infrastructure. Furthermore, availability and performance may often be competing attributes for the application. For example, availability improves by increasing the number of replicas but that may diminish the performance due to additional processing and communication required in maintaining consistency. In fact, Brewer's theorem states that consistency, availability, and partition tolerance are the three commonly desired properties by a distributed system, but it is impossible to achieve all three [50]. This implies that, in addition to the replication strategy, the service provider must identify a suitable deployment level and estimate the number of application replicas by taking into account both the dependability and performance parameters.

5.2.3 Deployment Levels in Cloud Infrastructures

Dependability, resource costs, and performance of an application may vary based on the location of its replicas. This section discusses three different deployment scenarios and identify how fault tree of the service instance can be integrated based on the chosen scenario. A deployment scenario corresponds to the location (or configuration) of the physical host on which individual replicas (VM instances) of an application under a

single implementation of a dependability mechanism are created. We assume failures in individual resource type to be independent of each other.

Multiple machines within a cluster. Two replicas of an application can be placed on hosts that are connected by a ToR switch i.e., in a LAN. This deployment provides benefits in terms of low latency and high bandwidth but offers least failure independence. Replicas cannot communicate and execute the dependability protocol upon a single switch failure, or a failure in the power distribution unit results in an outage of the entire application. In fact, if both replicas are placed on the same host, a single component failure will affect both replicas. In this deployment scenario, the cluster level fault tree blocks for each type of resource failure (see Figure 5.2) must be connected with a logical AND operator (e.g., DU1, DU2 of the cluster $\wedge$ S connecting the hosts $\wedge$ individual host components). We note that the overall availability and reliability obtained from each dependability mechanism with respect to host failures must be determined using a Markov model.

Multiple clusters within a data center. Two replicas of an application can be placed on hosts that belong to different clusters in the same data center i.e., connected via a ToR switch and AggS. This deployment still provides moderate benefits in terms of latency and bandwidth, and offers higher failure independence. The replicas are not bound to an outage with a single power distribution or switch failure. Therefore, to represent the overall availability of an application, in this scenario, the cluster level blocks from the fault trees may be connected with a logical OR operator in conjunction with power and network with an AND operator.

Multiple data centers. Two replicas of an application can be placed on hosts that belong to different data centers i.e., connected via a ToR switch, AggS and AccR. This deployment has a drawback with respect to high latency and low bandwidth, but offers a very high level of failure independence. A single power failure has least effect on the availability of the application. In this scenario, the data center level blocks from the fault trees may be connected with a logical OR operator in conjunction with the network in the AND logic.

In general, the subsystem chosen within the infrastructure to deploy users' applications can be denoted as the deployment level DL. A partial ordered hierarchy $(DL, \preceq_{DL})$ can be defined, where DL denotes the deployment level and $\preceq_{DL}$ defines the relationship between different deployment levels. For example, $C_1 \preceq_{DL} DC_1$ indicates that data center DC_1 is a larger subsystem or deployment level when compared to cluster C_1. A transitive closure $\preceq^*_{DL}$ that indicates the "contains-in" relationship also exists on $\preceq_{DL}$. For example, $h_1 \preceq^*_{DL} C_1 \preceq^*_{DL} DC_1$ indicates that host h_1 is part of cluster C_1 that in turn exists in data center DC_1. Intuitively, availability increases with increasing deployment level; that is, availability of an individual host is smaller than the availability of a cluster,

Table 5.1: Availability of users' applications using dep_sols with different replication schemes and deployment scenarios

	Same Cluster	Same Data center, diff. clusters	Diff. Data centers
Semi-Active	0.9871	0.9913	0.9985
Semi-Passive	0.9826	0.9840	0.9912
Passive	0.9542	0.9723	0.9766

which is still smaller than the availability of a data center. On the other hand, network latency increases with increasing deployment level; that is, hosts in the same rack have a lower network latency than hosts across different clusters. Hence, if $L(DL)$ denotes the maximum latency between two hosts in the deployment level DL, then DM can decide suitable DL based on users' desired performance in terms of expected response time.

5.2.4 Analysis of dep_sol Behavior under Different Configurations

Since input parameters and availability values of hardware and system software are normally vendor-confidential, this data is derived from the tables published in [118, 39, 127] for the purpose of this study. Based on this data and using the evaluation scheme discussed in this section, as an example, analysis of dependability and performance parameters is provided in the following. A detailed description of the simulation environment setup and the evaluation methodology is presented in Section 5.5.

Analysis of Applications' Availability

Replication scheme vs. deployment level. The first simulation studies the overall availability of each representative replication scheme of dep_sols with respect to different deployment levels. Table 5.1 illustrates the availability results. We can see that availability of the application is highest when replicas are placed in two different data centers. The value is slightly lower for the deployment level 2 (replicas in two different clusters) and still lower for scenario where replicas are placed inside the same LAN. Similarly, the overall availability obtained by semi-active replication is slightly higher than semi-passive replication, and lowest for simple passive replication scheme. The values in this table can be used by the service provider to select appropriate dep_sols and its deployment level.

Number of replicas vs. deployment level. Consider that DL^{max} is the highest deployment level within which DM executes a given application; that is, all the applications tasks are deployed and run on the hosts within the deployment level DL^{max}. Then DM can use the following study to assign appropriate DL^{max} value for each users' application.

Let us first extend the approach described in the previous sections to characterize

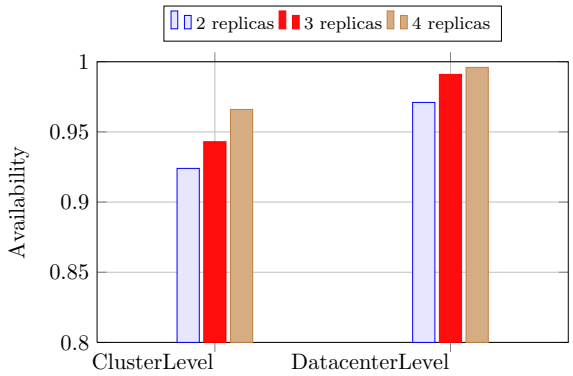


Figure 5.6: Availability at different deployment levels with varying number of replicas

the failure behavior of deployment levels (in contrast to individual hosts). As typically considered in the literature [70], we can model the failures at deployment levels using a Poisson process with rate λ_{DL}=$1/MTBF_{DL}$, where $MTBF_{DL}$ is the mean time between failures for deployment level DL. Since users' applications are executed on the Cloud infrastructure, their failure arrival process must also modeled as a Poisson process with rate $\sum_{DL} \lambda_{DL}$. This implies that a failure event affects the deployment level DL with probability

$$\frac{\lambda_{DL}}{\sum_{DL} \lambda_{DL}}$$

and causes the application to fail if DL^{max} lower than DL. Hence, the MTBF for a given application M in a given configuration is

$$MTBF_M = \Big(\sum_{\forall DL,\ \exists T_i \in M,\ DL^{max}(R(T_i)) \preceq^*_{DL} DL} MTBF_{DL}^{-1} \Big)^{-1}$$

Based on the above modeling, an estimation of the availability of an application having 2, 3 and 4 replicas, deployed in different clusters within a data center and different data centers in the Cloud is performed. The scenario where task replicas are deployed within a cluster since their behavior only depends on the availability and performance attributes of the physical hosts, and therefore, are not discussed here. Figure 5.6 illustrates how availability of the application changes for different configurations. We observe that the availability increases as the number of replicas of the application increase, and availability when the replicas are placed in different data centers is greater than the availability when replicas are placed in different clusters in the same data center.

Analysis of Applications Response Time

Number of replicas vs. deployment level. Three parameters that influence the performance of an application are considered: *i)* number of replicas for each application

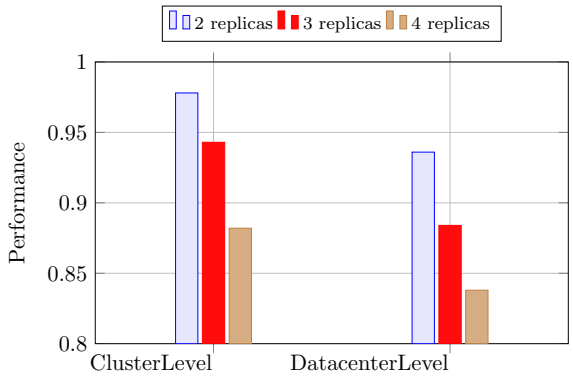


Figure 5.7: Performance at different deployment levels with varying number of replicas

task, *ii)* amount of resources allocated to each task, and *iii)* network latency between replicas. To quantify the performance of a given configuration, similarly to [70], [104], the layered queuing network formalism [47] is used as the application model. This queuing network model allows DM to predict the response time and resource requirements of the application for a chosen configuration and workload. In this study, application tasks are represented using first-come first-served (FCFS) queues and resource requirements (size of the VM instance) using the processor sharing queues. In this context, DM can measure the parameters of the model such as the response time, whenever a request arrives, by calculating the delay between the incoming request and the outgoing response.

Figure 5.7 illustrates how the response time of an application changes for different configurations. The performance values reported here largely depends on the network latency in the infrastructure since it is assumed that sufficient CPU capacity is allocated to each application task. For clarity, the results are presented using normalized values between 0 and 1. We can observe that the performance decreases as the number of replicas of the application increase, and performance of an application when its replicas are placed in different clusters within a data center is better than the performance when its replicas are placed in different data centers.

Therefore, in practice, we can see that there is a strict dependency between the availability and response time for an application, and the configuration choice for the application must take into account both performance and availability metrics. Finally, we note that this approach of quantifying different aspects of dependability solutions allow the DM to select dep_sols satisfying users' requirements.

5.2.5 Dependability Policy Selection Scheme

The design stage of the DM starts when a user requests the service provider to offer dependability support to her application (see Section 4.4). In this stage, the service

provider must analyze the user's requirements, match them with available dep_units, and form a complete dependability solution using appropriate dep_units. We assume that the service provider realizes a range of dependability mechanisms as dep_units, combines them to obtain different dep_sols, and determines the reliability and availability values that can be obtained using each dep_sol for different configurations (e.g., no. of replicas) and deployment levels (e.g., Table 5.1). This data is then bound to each dep_sol as its metadata and stored in the Service Directory. In this section, we introduce a method to select appropriate dependability mechanisms (dep_sols) and deployment levels based on user's requirements.

We briefly discuss the notion of dependability properties for the sake of completeness. The dependability property p of a dep_sol is denoted using a triple $p{=}(s, \hat{p}, A)$ where s is the dep_sol, $\hat{p}$ represents the high level abstract properties such as reliability and availability, and A denotes the set of structural, functional and operational attributes that refers to the granularity at which s can handle failures, benefits and limitations of using s, inherent resource consumption costs and quality of service metrics. Each attribute $attr{\in}A$ can take a value $v(attr)$ from a domain $\mathcal{D}_{attr}$ and a partial ordered relationship $\preceq_{attr}$ exists on the domain. For instance, dependability property of a dep_sol s_1 can be denoted as $p{=}(s_1$, {availability=99.995%, reliability=98%}, {mechanism=semi-passive-replication, fault_model=server_crashes, network_faults, power_failures, fault_detection_time=5ms, recovery_time=8ms, n.replica=3}). A hierarchy of dependability properties $\preceq_p$ can also be defined; if P is the set of all properties, and given two properties $p_i{,}p_j{\in}P$, $p_i{\preceq_p}p_j$ if $p_i{\cdot}\hat{p}{=}p_j{\cdot}\hat{p}$ and $\forall attr{\in}A$, $v_i(attr){\preceq}v_j(attr)$. The attribute values depend on the configuration of the dependability mechanism and values for abstract properties are determined using Markov models and fault trees. According to our discussion in Section 4.4, a user can specify her requirements in terms of desired abstract properties $\hat{p_c}$ and constraints on attribute values A_c.

Let S be the set of dep_sols available in the Service Discovery. For a given user request, first shortlist the dep_sols that satisfy user's abstract property requirements. Let $S'{\subseteq}S$ be the shortlisted set of dep_sols for which $\hat{p_c}{\preceq_p}\hat{p_i}$, $\forall i{\in}S$. Any $s{\in}S'$ can be used to deliver the service if the user does not provide constraints in terms of total resource usage costs or performance of the dependability protocol since high level reliability and availability requirements can be satisfied by any $s{\in}S'$. However, since a user's input may contain specific attribute values, for each dep_sol in S', compare attribute values for each $attr{\in}A$ to obtain a set S'' of candidate dep_sols. In particular, compare the values of each attribute $v_i(attr)$ with the value specified by the user $v_c(attr)$, and include those dep_sols in S'' for which $v_c(attr){\preceq_a}v_i(attr)$. For example, fault_detection_time or recovery_time must be less than or equal to the specified value, whereas number of replicas must be greater than or equal to the specified value. By performing this step, the DM selects only those dep_sols that satisfy both user's high level requirements and additional conditions on the attributes. We note that there may be some inconsistencies in the matching and comparison processes that can be handled based on the priorities specified by the user. Finally, compare each

Table 5.2: An example of dependability properties of dep_sols and user's requirements

PROPERTY OF AVAILABLE DEP_SOLS

dep_sol	p	$\hat{p}$	A
s_1	p_1	Availability=99.9% Reliability=99%	mechanism=semi-active-replication n.replicas=3 fault_detection_time=2ms recovery_time=2ms deployment_level=2
s_2	p_2	Availability=95 %	mechanism=passive-replication recovery_time=30sec dimension=shared
s_3	p_3	Availability=99% Reliability=98%	mechanism=semi-active-replication n.replicas=2 fault_detection_time=4ms recovery_time=8ms deployment_level=1

USER'S REQUIREMENTS

	p	$\hat{p}$	A
s_c	p_c	Availability$\geq$99% Reliability$\geq$98%	n.replicas$<$3 fault_detection_time$\leq$5ms recovery_time$\leq$10ms

dep_sol within S'' and order them with respect to user's requirements. The first dep_sol in the ordered set S'' can be finally used to provide dependability service to user's application since it most appropriately satisfies user's requirements.

Example 5.2.1. *Assume that the service provider realizes three dep_sols with properties described in Table 5.2. The Service Discovery engine of the DMKernel first generates the set* $S'=(s_1, s_3)$ *since*

$$s_c(reliability) \leq s_1(reliability) \wedge s_c(availability) \leq s_1(availability),$$

$$s_c(reliability) \leq s_3(reliability) \wedge s_c(availability) \leq s_3(availability)$$

It then compares the elements in S' *to generate an ordering among the shortlisted solutions. In this case, it discards* s_1 *from* S' *since* $s_1(n.replicas) \not\leq 3$*; hence,* $S''=\{s_3\}$*. Finally, since* $|S''|=1$*, the dep_sol* s_3 *is used by the service provider to deliver the dependability service to the user. That is, two replicas of the users' application are created and placed within the same cluster; the semi-active replication scheme is used to maintain the state of each replica.*

Typically, to ensure high levels of dependability for the users' applications, the service provider must realize the delivery scheme consisting of both the design and runtime stage.

- In the design stage, once an appropriate dep_sol is selected according to the user's request, the service provide must enhance the resource allocation algorithm of the IaaS service provider to actually deliver the service. In particular, the allocation algorithm must be modified to integrate specific requirements (e.g., deployment level of each replica) of chosen dependability mechanisms and their configuration. The resulting resource management algorithm should satisfy allocation requirements of the application that are defined by the DM, as well as, meet IaaS service provider's business goals. We discuss such resource allocation algorithm in Section 5.4.2

- Since the context of a dependability solution may change at runtime due to the dynamic nature of the Cloud computing environment, the attribute values of each dependability solution offered to the user must be continuously monitored in the runtime stage. For example, the real-time attributes of the host on which a replica is located must be monitored in the runtime stage to ensure that user's dependability requirements are satisfied throughout the application life-cycle. The current allocation of the application must be changed whenever the availability or performance of the application is affected due to system changes. We discuss such adaptive resource management algorithm in Section 5.4.3.

We note that the above two aspects realize the Resource Manager, Replication Manager, Fault Detection/Prediction Manager, and Fault Masking Manger of the DM.

5.3 Integrating Dependability Policy Conditions within the IaaS Paradigm

In general, when a user requests the IaaS service provider to allocate her a set of computing resources, the VM provisioning algorithm follows a heuristics-based approach to allocate VM instances on physical hosts and provides them to the user. Most implementations presently build their provisioning algorithms either by focusing on realizing the service with agility (hence not scaling well to the granularity of individual resource types on physical hosts) or to meet the IaaS service provider's business objectives (e.g., utilize fewer number of physical hosts to save energy consumption costs). However, the IaaS service provider may need to impose a set of additional conditions on the allocation algorithms to maintain the security and performance of its system, and the dependability service provider (DM) may need to impose conditions to improve the dependability of users' applications. To this aim, in this section, we first categorize and formalize the requirements that the IaaS service provider and the dependability service provider may want to specify with respect to the security, reliability, availability and performance of the system. We then address

the satisfaction of such requirements in the overall problem of resource allocation in IaaS Clouds.

Let us start by modeling users' applications. In order to keep the system independent of specific formalisms for representing users' applications, we can assume an application M to be as a set of interacting tasks $M = \{\tau_1, \ldots, \tau_m\}$. The DM may replicate critical tasks of the application to improve its dependability, and obtain a set of task replicas $R_k = \{\tau_1^1, \ldots, \tau_m^{|r_k|}\}$. This implies that the set of tasks to be deployed in the Cloud is $T = \{t_i\} = \bigcup_{\tau_k \in M} R_k$. Similarly to hosts, a task $t \in T$ can be characterized by a vector $\overrightarrow{t} = (t[1], \ldots, t[d])$, where each dimension represents the task's requirements for specific computing resources (e.g., CPU, Mem), corresponding to the dimensions of physical hosts. We can assume that tasks are to be executed in a virtual environment and, for each task, a virtual machine that is capable of executing that task can be instantiated. In other words, we can assume existence of a virtual machine image I that can be characterized by $\overrightarrow{t}$. This assumption reduces our problem to a VM provisioning problem, where each VM instance $v \in \mathcal{V}$, represented by $\overrightarrow{v} = (v[1], v[2], \ldots, v[d])$, needs to be mapped on a physical host $h \in \mathcal{H}$. Similarly to hosts, resource requirements of VM instances can also be normalized to values between 0 and 1. For example, a "small" instance in Amazon EC2 service may be translated to $\overrightarrow{v}$=(CPU, Mem)=(0.4, 0.25) [44].

5.3.1 Resource Allocation Objective

In the perspective of the DM, the task of resource provisioning involves allocation of VM instances of specified dimensions on physical hosts, and their delivery to the user. VM provisioning can be characterized by a *mapping* function $p: \mathcal{V} \rightarrow \mathcal{H}$ that takes the set $\mathcal{V}$ of VM instances as input and maps each $v \in \mathcal{V}$ on the physical hosts $h \in \mathcal{H}$ as output. The notation $p(v)=h$ represents that the VM instance v is provisioned on the physical host h. Figure 5.8 illustrates an example of mapping generated by $p: \{v_1, \ldots, v_6\} \rightarrow \{h_1, h_2, h_3\}$ where $p(v_1)=p(v_4)=p(v_6)=h_1$, $p(v_5)=h_2$ and $p(v_2)=p(v_3)=h_3$. In the figure, a one-to-one correspondence is assumed between the VM images and application tasks, and VM instances are allocated on the Cloud infrastructure. The VMs and hosts are represented using rectangles to denote two resource dimensions (CPU and Memory) by its sides. A physical host can accommodate more than one VM instance, but an individual VM can be allocated only on a single host. DM must guide the mapping function to meet one (or both) of the following IaaS service provider's objectives:

- To reduce the energy consumption and operational costs, VM instances are consolidated on physical hosts to maximize the number of *free* hosts. For example, the mapping function $p: \{v_1, \ldots, v_6\} \rightarrow \{h_1, h_2, h_3\}$ is guided to achieve $p(v_1)=p(v_4)=p(v_6)=h_1$ and $p(v_2)=p(v_3)=p(v_5)=h_3$ so that the host h_2 remains unused. This allows the IaaS service provider to conserve the energy of running host h_2, and increase its service-response capability.

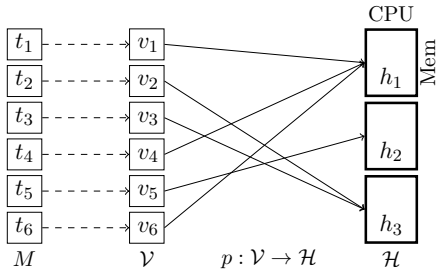


Figure 5.8: An example of mapping generated by $p{:}\mathcal{V}{\rightarrow}\mathcal{H}$ function

- To reduce the load variance of physical hosts across all the clusters in the Cloud in order to improve the performance and resilience of the system. For example, if we assume that $h_1{\in}C_1$, $h_2{\in}C_2$ and $h_3{\in}C_3$, the mapping function $p{:}\{v_1,\ldots,v_6\}{\rightarrow}\{h_1,h_2,h_3\}$ must to be guided to achieve $p(v_1){=}p(v_4){=}h_1$, $p(v_5){=}p(v_6){=}h_2$ and $p(v_2){=}p(v_3){=}h_3$ so that the VM instances (resource usage) are uniformly distributed across the clusters.

Section 5.4.2 defines a mapping function that considers the two objectives during VM provisioning and integrates it as part of the DM so that it can satisfy and deliver the dependability policy requirements chosen for the users' application.

5.3.2 Resource Allocation Constraints

The IaaS service provider's and DM's requirements are modeled as *placement constraints* and the mapping function is guided to satisfy all the constraints. We distinguish three kinds of placement constraints:

- *Global constraint* that applies to all the hosts and VM instances in the system at any given instant of time;
- *Infrastructure-oriented constraints* that are specified by the IaaS service provider to maintain the security and quality of its service;
- *Application-oriented constraints* that are specified by DM, based on the chosen dependability mechanism, to increase the availability and reliability of users' applications.

For simplicity we consider constraints specified with respect to specific VM or host identifier, but they can also be specified with reference to their properties (e.g., all the hosts that belong to a cluster or deployment level). Table 5.3 provides a summary of the constraints in all three perspectives.

Global Constraint

The classical *resource capacity* constraint states that the amount of resources consumed by all the VM instances that are mapped on a single host cannot exceed the total capacity of the host in any dimension. Formally, for all the VM instances $v \in \mathcal{V}$ and hosts $h \in \mathcal{H}$ in the system, the mapping function $p{:}\mathcal{V} \rightarrow \mathcal{H}$ must satisfy

$$\forall h \in \mathcal{H}, d \in \mathcal{D}, \sum_{v \in \mathcal{V} | p(v)=h} v[d] \leq h[d] \tag{5.1}$$

This placement requirement is typically supported by all the solutions existing in the literature. However, several solutions do not consider that the amount of resources consumed by a VM when placed in isolation on a host and with other co-hosted VMs may not be the same. When multiple VMs are placed on a host, the hypervisor or host operating system may consume additional resources (e.g., CPU cycles or I/O bandwidth during resource scheduling), and VM instances may interfere with each other and consume higher amounts of shared resources (e.g., the L2 cache during context switching). Formally, if $v_j, v_k \in \mathcal{V} | p(v_j)=p(v_k)=h$, and utilize $v_j[d]$ and $v_k[d]$ amount of resources in the d^{th}-dimension when allocated individually, then $v_j[d]$ and $v_k[d]$ together may utilize a bit more than $(v_j[d]+v_k[d])$ resources from the host h (unless v_j and v_k uses the same VM image). To avoid performance degradation and inconsistent system state due to the above factors, we allow the IaaS service provider to define an upper bound on the resource capacity of each host that can be used for VM provisioning. The remaining capacity is then used by the service provider for VM management. The resource capacity constraint demands that VM instances not be allocated on a host if its capacity in any dimension reaches the upper bound or *threshold* value specified by the service provider, that is,

$$\forall h \in \mathcal{H}, d \in \mathcal{D}, \sum_{v \in \mathcal{V} | p(v)=h} v[d] \leq (h[d] * threshold[d]) \tag{5.2}$$

We note that the $threshold[d]$ value can be specified in terms of percentage or normalized value between $(0, 1)$.

Example 5.3.1. *Suppose that the IaaS service provider specifies the upper bound on CPU and Memory usage of host h_1 as 80% and 70% respectively for VM provisioning. Assuming that the resource capacity of host h_1 in each dimension is normalized to 1, the resource capacity constraint specifies that the mapping function must satisfy:*

$$v_1[cpu]+v_4[cpu]+v_6[cpu] \leq 0.8 \quad and$$

$$v_1[mem]+v_4[mem]+v_6[mem] \leq 0.7$$

Table 5.3: An example of constraints on the mapping function

Perspective	Applied by	Constraints	Description
Global	IaaS Service Provider	resource capacity	Resources consumed by VM instances must be less than the upper bound (*threshold*) of host's capacity
Infrastructure oriented	IaaS Service Provider	forbid	Forbid a set of VM instances from being allocated on a specified host
		count	The number of VM instances deployed on a host must be less than a given value
Application oriented	DM (User)	restrict	Map a VM instance only on a specified set of physical hosts
		distribute	Allocate a specified pair of VM instances on different hosts
		latency	The network latency between the specified pair of VM instances must be less than a given value

Infrastructure-oriented Constraints

The IaaS service provider may need to impose restrictions on the mapping function, involving a set of physical hosts in its infrastructure, to improve the security, operational performance and reliability of its service. As typical needs, we include two representative infrastructure-oriented constraints in our framework that can be adapted to satisfy such requirements: forbid and count.

Forbid. To improve security, a IaaS service provider may need to specify that a set of hosts in its infrastructure must execute only system-level services (e.g., the access control engine or reference monitor) and the mapping function must not allow DM to allocate VM instances running users' applications on those hosts. To satisfy such requirements, the forbid constraint prevents a VM instance v from being allocated on a physical host h. Formally, when the IaaS service provider defines a set $Forbid=\{(v_i, h_j)|v_i \in \mathcal{V}$ and $h_j \in \mathcal{H}\}$ specifying the VM instances $v_i \in \mathcal{V}$ that must be forbidden from being allocated on hosts $h_j \in \mathcal{H}$, the VM provisioning algorithm guides the mapping function $p:\mathcal{V} \rightarrow \mathcal{H}$ to satisfy the following condition:

$$\forall v \in \mathcal{V}, h \in \mathcal{H}, \quad (v, h) \in Forbid \implies p(v) \neq h \tag{5.3}$$

Count. As the number of co-hosted VM instances on a physical host increases, its performance degrades. For example, the performance of a storage disk decreases if the number of I/O intensive applications in the VM instances increases; similarly, the network traffic from a host, VM management costs, and cpu utilization costs gradually increases. To avoid such conditions, the count constraint allows a IaaS service provider to limit the number of VM instances that can be allocated on a given host. Formally, when the IaaS

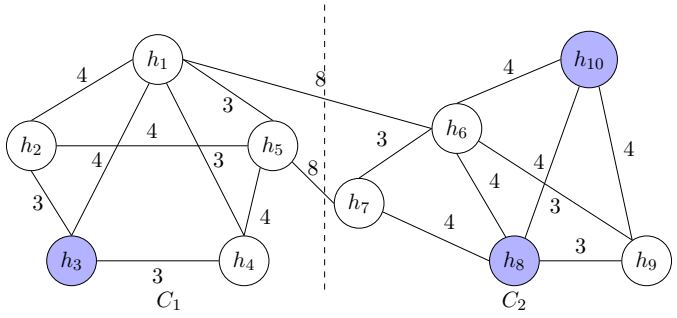


Figure 5.9: An example of a Cloud infrastructure with network latency values for each network connection and service provider's constraints (the shaded nodes forbid allocation of user's VM instances)

service provider defines $count_h$, the maximum number of VM instances allowed on host h, the mapping function $p{:}\mathcal{V}\rightarrow\mathcal{H}$ ensures that the following condition is satisfied:

$$\forall v \in \mathcal{V}, h \in \mathcal{H}, \quad |\{v \in \mathcal{V} | p(v) = h\}| \leq count_h \tag{5.4}$$

Example 5.3.2. *Figure 5.9 illustrates an example of a Cloud infrastructure that consists of two clusters* $\mathcal{C}$=$\{C_1,C_2\}$ *and each cluster contains five physical hosts* C_1=$\{h_1,\ldots,h_5\}$ *and* C_2=$\{h_6,\ldots,h_{10}\}$. *Assume that the IaaS service provider* i) *runs security services on hosts* $\{h_3,h_{10}\}$, ii) *has taken host* h_8 *under maintenance due to a failure,* iii) *requires all the hosts in cluster* C_1 *to allocate not more than 3 VM instances each and utilize up to 80% of their aggregate CPU and 90% of Memory capacity, and* iv) *requires hosts in cluster* C_2 *to accommodate a maximum of 2 VM instances each. If we assume that the IaaS service provider forbids all the VM instances from being allocated on physical hosts that are either used to run security services or that have undergone failures, the IaaS service provider can specify its requirements using the following constraints:*

- $count_{h_1},\ldots,count_{h_5} \leq 3$
- $count_{h_6},\ldots,count_{h_{10}} \leq 2$
- $threshold_{h_1}[CPU]=\ldots=threshold_{h_5}[CPU]=0.8$
- $threshold_{h_1}[Mem]=\ldots=threshold_{h_5}[Mem]=0.9$
- $\forall v \in \mathcal{V}$: $Forbid=\{(v,h_8),(v,h_3),(v,h_{10})\}$

Application-oriented Constraints

The DM may need to impose a set of restrictions on the placement of users' application's VM instances based on the chosen dependability policy. For example, consider that a generic dependability mechanism such as a replication technique is chosen by DM to increase the reliability and availability of users' application. The DM may then need to impose a set of additional conditions on the system parameters and the relative placement of VM instances to successfully implement the dependability mechanism while satisfying the performance goals. As typical needs, we include three representative application-oriented constraints in our framework that can be used to realize such conditions: restrict, distribute and latency.

Restrict. Based on the system's failure behavior, users' requirements and chosen dependability policy, DM may want to allocate each application task replica in specific a deployment level. To satisfy such requirements, the restrict constraint limits a VM instance $v \in \mathcal{V}$ on being allocated only on a specified group of physical hosts $H \subset \mathcal{H}$. When DM defines a set $Restr = \{(v_i, H_j) | v_i \in \mathcal{V} \text{ and } H_j \subset \mathcal{H}\}$, the mapping function $p: \mathcal{V} \rightarrow \mathcal{H}$ ensures the following condition:

$$\forall v_i \in \mathcal{V}, H_j \in 2^{H_j}, (v_i, H_j) \in Restr \implies p(v_i) \in H_j \tag{5.5}$$

We note that the set H_j defined above can refer to a deployment level DL, maximum deployment level DL^{max} or a cluster C, and ensure satisfaction of dependability requirements. The restrict constraint is also applicable in other scenarios such as *i)* based on the security and privacy policies [13], and mandatory government enforced obligations (e.g., EU Data Protection 95/46/EC Directive), a user may require that her VM instances be always located within a given community area (e.g., within EU countries), and *ii)* to improve the application's performance, a user may require the mapping function to place her VM instances on hosts whose geographical location is closest to her customers.

Distribute. Any replication-based dependability mechanism inherently requires that each replica be placed on different physical hosts to avoid single points of failure. For instance, if the DM replicates an application on two VM instances, and if both the virtual machines are allocated on the same host, then a failure in the host results in a complete outage of the user's application. To avoid reaching such a state, the distribute constraint that allows a user to specify that two VM instances v_i and v_j be never located on the same host at the same time. When the DM defines the set $Distr = \{(v_i, v_j) | v_i, v_j \in \mathcal{V}\}$ of pairs of VM instances that cannot be deployed on the same host, the mapping function $p: \mathcal{V} \rightarrow \mathcal{H}$ satisfies the following condition during VM provisioning:

$$\forall v_i, v_j \in \mathcal{V}, h \in \mathcal{H}, (v_i, v_j) \in Distr \implies p(v_i) \neq p(v_j) \tag{5.6}$$

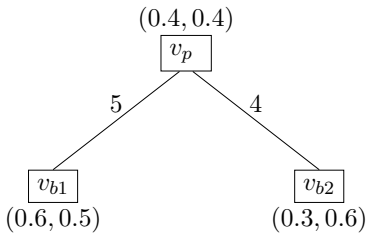


Figure 5.10: An example of users' resource requirements and network latency constraints

Latency. The DM may want to specify allowed latency between task replicas so that the response time of the users' application is within the desired value. The latency constraint enforces the mapping function to allocate two VM instances v_i,$v_j \in \mathcal{V}$ such that the network latency between them is less than a specified value T_{max}. When DM defines a set $MaxLatency=\{(v_i, v_j, T_{max}|\ v_i,v_j \in \mathcal{V})\}$ that specifies the acceptable network latency T_{max} between two VM instances v_i and v_j, the mapping function $p:\mathcal{V} \rightarrow \mathcal{H}$ ensures the following condition:

$$\begin{aligned} \forall v_i, v_j \in \mathcal{V}: \quad & (v_i, v_j, T_{max}) \in MaxLatency \\ & \implies latency(p(v_i), p(v_j)) \leq T_{max} \end{aligned} \tag{5.7}$$

assuming that the network latency between two virtual machines is equal to the network latency between the physical hosts on which they are deployed. For instance, in a checkpoint-based reliability mechanism, the state of the backup VM instance must be frequently updated with that of the primary instance to maintain the system in a consistent state. This task involves high amounts of message exchanges, and hence an upper bound in the network delay is essential; otherwise, the wait-time of the primary instance during which the state transfer to the backup takes place may increase significantly and the overall availability of the application may be reduced.

Example 5.3.3. *In Example 5.2.1, the service provider uses the matching and comparison processes and selects the dep_sol with property* p_3=(s_3, {*availability=99%, reliability=98%*}, {*mechanism=semi-active-replication, n.replicas=2, fault_detection_time=4ms, recovery_time=8ms, deployment_level=1*}). *According to the chosen property, consider that DM replicates the primary VM instance* v_p *of the user's application on two backup hosts* v_{b1}, v_{b2} *(in* p_3, *n.replicas=2), and applies the semi-active replication scheme. To specify the allocation requirements inherent to the property* p_3, *the DM may wish to impose a restriction on allocating* v_p, v_{b1} *and* v_{b2} *in the same cluster* C_1 *(in* p_3, *deployment_level=1), particularly with* v_p *begin allocated on one of the hosts* $h_2 \ldots h_5$ *(based on the infrastructure defined in Example 5.3.2). Furthermore, to avoid a single point of failure, it may wish to ensure that* v_p, v_{b1} *and* v_{b2} *be deployed on different physical hosts. Finally, to balance the application's performance and availability, the DM may require that the network latency*

limit T_{max} between v_p and v_{b1} is at most 5ms and between v_p and v_{b2} is at most 4ms. Figure 5.10 illustrates an example of a DM's requirements for VM instances and associated allocation constraints. These requirements can be specified as follows:

- $Restr$=$\{(v_p, \{h_2, \ldots, h_5\}), (v_{b1}, \{h_1, \ldots, h_5\}), (v_{b2}, \{h_1, \ldots, h_5\})\}$
- $MaxLatency$=$\{(v_p, v_{b1}, 5), (v_p, v_{b2}, 4)\}$
- $Distr$=$\{(v_p, v_{b1}), (v_p, v_{b2}), (v_{b1}, v_{b2})\}$

5.4 Delivering Dependability Support

We describe an approach to VM provisioning in which each mapping of the function p:$\mathcal{V}\rightarrow\mathcal{H}$ is regulated to satisfy the allocation constraints specified by both the DM and the IaaS service provider. The objective of the mapping function is to reduce the load variance and energy consumption costs of the Cloud infrastructure. We note that the objective of this algorithm focuses on the business goals of the IaaS service provider, thus providing benefits to users (dependability support) as well as the Cloud service providers. An approach to resource allocation with a user-centric objective is discussed in Chapter 6.

Given the NP-hardness of the VM provisioning problem, a greedy, heuristics-based approximation algorithm is used to solve it. At a high-level, each time the DM is required to deploy a users' application, the provisioning algorithm analyzes the Cloud infrastructure to identify the clusters and physical hosts that can be used for resource allocation, and for each shortlisted physical host, it identifies the set of VM instances that can be allocated on that host.

The provisioning algorithm reduces the load variance between different clusters by allocating VM instances in the cluster that has highest amount of available resources. This heuristic is based on the observation that the load variance between clusters can be implicitly reduced by selecting the less-used cluster and utilizing its resources (by allocating new VM instances on its hosts). When selecting a cluster, resource availability of a cluster corresponding to a specific resource type may be greater than other clusters but smaller for other resource types (e.g., CPU and Mem availability of clusters C_1, C_2 may be (0.6, 0.7) and (0.4, 0.8) respectively). In such cases of inconsistency, the provisioning algorithm determines the total amount of resources required by the users' application in each dimension, and selects the cluster whose availability is largest for the dimension that is most required by the application. To reduce the energy consumption costs, VM provisioning is performed in a host-centric manner; that is, once a cluster with maximum resource availability is selected, each host within the cluster is analyzed (e.g., by following identifier's order) to allocate as many VM instances on that host as possible. This heuristic improves the resource capacity usage of individual hosts and map all the VM instances on fewer number of physical hosts. The task of allocating multiple VM instances on a given physical host is generally referred as VM consolidation, and is discussed below.

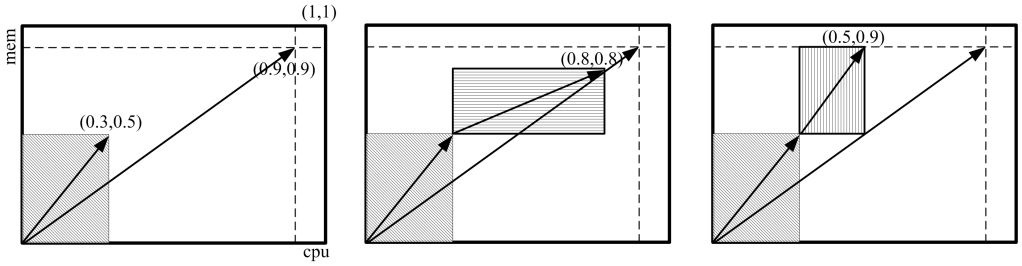


Figure 5.11: VM allocation on a host using the vector dot-product method

5.4.1 Virtual Machine Consolidation

The task of allocating virtual machines on physical hosts is implemented as a variant of the well known bin-packing problem. In this context, the bins correspond to the physical hosts and the items correspond to the VM instances. The VM provisioning algorithm of the DM performs consolidation (packing) using the vector dot-product method, similarly to [106, 117]. VM instances and physical hosts are represented as vectors (see Section 5.2.1), and therefore, the vector dot-product value measures how a VM instance imposes itself on a given host based on the angle between optimum allocation (i.e., reaching point $(1, 1)$), vector magnitude, and present state of the host. In particular, given a host h_j, the vector dot-product method performs the following operations.

1. A weight $w[d]=(\sum_{v\in\mathcal{V}|p(v)=h_j} v[d])/h_j[d]$ is associated with each resource dimension of the host; $w[d]$ is the ratio between the total resource consumption and capacity of the host. Semantically, the weight assigned to the host reflects the scarcity of resources on that host as a scalar quantity.

2. The free capacity of the host $h'_j[d]$ is then calculated as the difference between the host's capacity and total resource consumption; $h'_j[d]$ reflects the ability of the host to accommodate further resources.

3. For each VM instance $v_i \in V$, the vector dot-product value is calculated as $hv=\sum_d w[d]h'[d]v[d]$. Semantically, the VM instance with highest hv value is most suitable to be allocated on host h_j.

Example 5.4.1. *Figure 5.11 illustrates an example of the vector dot-product method for host h_j and VM instances v_1 and v_2. The host and VM instances are shown using rectangles to represent each side with a specific resource type (CPU and memory). The VM instance v_0, represented with slanted pattern, is already allocated on host h_j, and allocation of VM instances v_1 and v_2 on current status of h_j is represented using horizontal and vertical patterns respectively. Resource requirements of VM instances are $\overrightarrow{v_1}=(0.5, 0.3)$ and $\overrightarrow{v_2}=(0.2, 0.4)$, and their dot-product values are* 0.166 *and* 0.129 *respectively. By extracting*

the VM with larger dot-product value (v_1, which is CPU-intensive) the resource utilization of the host complements the already containing VMs (v_0, which is Memory-intensive).

5.4.2 Virtual Machine Provisioning

Figure 5.12 illustrates the algorithm that is executed each time the DM must allocate a set of VM instances $V=\{v_1, \ldots, v_n\}$. The resource requirements for each $v_i \in V$ are specified in d dimensions and the application's dependability and performance requirements are specified in the form of constraints. The algorithm takes the set V, set of all hosts $h \in \mathcal{H}$, parameters that group physical hosts into clusters $\mathcal{C}$ and snapshot of VM instances $\mathcal{V}$ already allocated in the Cloud as input. It also takes the sets that specify the dependability, security, and performance requirements of users' applications and IaaS service providers (*Restr*, *Forbid*, *MaxLatency*, *Distr*, *Count*) for all VM instances and hosts, and provisions the requested resources while satisfying all the constraints, as output.

To select the least-used cluster for allocating $v_i \in V$, the priority queue CL is built based on the resource availability in each cluster (line 5), and then, the cluster C with maximum resource availability is extracted from CL (line 8). Similarly, the vector dot-product value of each $v_i \in V$ is calculated and stored in the VMreq priority queue (line 22). Entries from VMreq are extracted in the decreasing order of the dot-product values (line 25) and analyzed for performing the final allocation. We note that in the absence of additional constraints, the above two mechanisms of: *i)* selecting the least-used cluster and *ii)* allocating VM instances on its hosts using the dot-product method are sufficient to perform VM provisioning and meeting service provider's business goals. In our context, the provisioning algorithm introduces the following controls to ensure that the specified allocation constraints are satisfied.

Based on the observation that forbid and restrict constraints define conditions on the association between VMs and physical hosts, this algorithm applies controls corresponding to these constraints mainly while building the VMreq priority queue (i.e., when analyzing the suitability of allocating $v_i \in V$ on host h). Note that only the VM instances that are extracted from VMreq are considered for allocation on any given host. Hence, it creates a temporary set V' that contains all the VMs that must be allocated, discard all the VMs v from set V' (line 18):

- If an entry $(v, h) \in$Forbid exists (line 15), that is, discard all $v \in V'$ that are specified by the IaaS service provider in the *Forbid* set for host h.
- If a set of hosts H is specified in the *Restr* set for a VM $v \in V$ but the present host h does not belong to the set H (line 16), that is, discard v if $\exists (v, H) \in Restr \wedge h \notin H$.

and provide the set V' as input to build the priority queue VMreq. This control allows the provisioning algorithm to enforce the forbid and restrict constraints by ensuring that none of the VM instances that are in conflict with these constraints are allocated on host h.

```
1:INPUT     H, C, V, V, Restr, Forbid, Count, Distr, MaxLatency
2:OUTPUT    p:V→H
3:MAIN
4:/*Initialize the priority queue CL based on resource availability in each cluster*/
5:CL:=Build_Priority_Queue(C) /*Analyze the clusters*/
6:WHILE V≠φ do /*There are still VMs to be allocated*/
7:              /*Select the cluster C that has maximum resource availability*/
8:              C:= Extract_Max(CL)
9:               /*Consider each host in cluster C that does not violate the count constraint*/
10:             for each h∈C ∧ h'[d+1]<count_h do /*h'[d+1] is the usage counter*/
11:   V':=V /*Initialize the set V' of VMs which can be allocated on host h*/
12:   /*Remove each VM that violates forbid/restrict constraints wrt host h*/
13:   for each v∈V do
14:     if
15:       (v,h)∈Forbid ∨    /*If VM v is forbidden from host h or*/
16:       {(v,H)∈Restr∧h∉H} /*If h is not in the restricted set of hosts*/
17:     then
18:       V':=V'\{v}
19:   end for
20:    /*Initialize the set VMreq of VMs v∈V' with their respective*/
21:    /*dot-product values computed wrt host h*/
22:   VMreq:=Build_Priority_Queue(h,V')
23:   while VMreq≠φ do /*There are still VMs in VMreq*/
24:      /*Select the VM with largest resource needs*/
25:     v:=Extract_Max(VMreq)
26:     /*If the capacity and count constraints are satisfied, and*/
27:     /*If not in conflict with distribute constraint*/
28:     if
29:       v⃗≤h'⃗ ∧  /*If VM v can be allocated in residual capacity of h and*/
30:       ∄(v,v_j)∈Distr|p(v_j)=h  /*If distribute constraint is satisfied*/
31:     then
32:       /*If v is not related to v_j∈V by latency constraints*/
33:       if ∄(v, v_j, T_max)∈MaxLatency then
34:         /*Allocate v on h and update residual resource capacity value*/
35:         p(v):=h
36:         h'⃗:= h'⃗−v⃗
37:         V:=V\{v}
38:       else
39:         /*Initialize the set of VMs that must be allocated in the same all-*/
40:         /*ocation cycle since they are linked by latency constraints to v*/
41:         Reserve_list:={(v, h)}
42:         /*Find an allocation for ∀v_j related to v by latency constraints*/
43:         if Forward_Allocate(v, h) then
44:         /*There is an allocation ∀v_j related to v by latency constraints*/
45:         WHILE Reserve_list≠φ do
46:         /*Allocate v_j on host h_j*/
47:         p(v_j):=h_j
48:         h_j'⃗:= h_j'⃗−v_j⃗
49:         Reserve_list:=Reserve_list\{(v_j,h_j)}
50:         V:=V\{v_j}
51:         end while
52:   end while
53:             end for
54:end while
```

Figure 5.12: VM provisioning with Capacity, Forbid, Restrict, Count, Distribute and Latency constraints

```
1:FORWARD_ALLOCATE(v, h)
2:/*Identify all v_j directly/indirectly related to v by latency constraints*/
3:V_l:=Build_Priority_Queue(v_j|(v, v_j)∈MaxLatency∧v_j∉Reserve_list)
4:WHILE V_l≠φ do /*There are VMs to be allocated*/
5:  /*Select the VM with the most stringent latency constraint*/
6:  v_l:= Extract_Min(V_l)
7:  /*Create the set K of hosts that qualify for v_l wrt the latency value */
8:  /*forbid and restrict constraints*/
9:  K:=Build_Priority_Queue(
10:   h_i|latency(h, h_i)≤T_max(v, v_l) ∧  /*Hosts that satisfy latency constraint*/
11:   (v, h_i)∉Forbid ∧    /*Hosts that do not violate forbid constraint*/
12:   h_i∈H|(v, H)∈Restr)  /*Hosts that satisfy the restrict constraint*/
13:  WHILE K≠φ do /*There are hosts to be considered*/
14:   /*Select a host k*/
15:   k:= Extract(K)
16:   /*Verify the capacity, count and distribute constraints*/
17:   if
18:    v⃗≤k⃗ ∧ /*If the capacity and count constraints are satisfied and*/
19:    ∄(v, v_j)∈Distr|p(v_j)=k /*If not in conflict with distribute constraint*/
20:   then
21:    /*Add (v_l, k) to the set of VMs related by the latency constraint*/
22:    /*to v which can be allocated*/
23:    Reserve_list:=Reserve_list ∪{(v_l, k)}
24:    /*Look for an allocation for VMs related by latency constraint to v_l*/
25:    if Forward_Allocate(v_l, k) then
26:     K:=φ /*Allocation found - no other hosts need to be considered*/
27:    else
28:     /*Allocation not found - discard the tentative allocation for v_l*/
29:     Reserve_list:=Reserve_list\{(v_l, k)}
30:   /*Else the host k does not satisfy the capacity, count and distribute*/
31:   /*constraints and thus, another host in K should be considered for v_l*/
32:  /*Verify if an allocation has been found for v_l*/
33:  if ∄(v_l, k)∈Reserve_list then
34:   /*No allocation found ∀v_l directly/indirectly related to v by*/
35:   /*latency constraints*/
36:   return 0
37:  end while
38:end while
39:/*All v_i directly/indirectly related to v with latency constraints can be*/
40:/*allocated satisfying all constraints*/
41:return 1
```

Figure 5.13: The Forward_Allocate function of the VM provisioning algorithm

The capacity and count constraints are confined to the resource usage of individual physical hosts. To ensure these constraints, the d dimensional vector representation of hosts and VM instances is extended. In particular, as a control to the count constraint, a new add dimension d+1 on each physical host is added and initialized its value to the number of VM instances that can be allocated on that host based on the value specified by the IaaS service provider i.e., $h[d+1]{:=}count_h$. Similarly, the $[d+1]^{th}$ dimension of each VM is initialized to 1, and before allocating VM $v{\in}V$ on host h, the count control is enforced along with the capacity constraint (line 29, 36, 48). To enforce the capacity constraint, the residual resource capacity of each physical host $\overrightarrow{h'}$ that is initialized using the threshold values specified by the service provider is maintained. Before allocating a VM instance $v{\in}V$ on host h, the algorithm verifies if the resource requirements of the VM instance $\overrightarrow{v}$ is less than the residual capacity of that host $\overrightarrow{h'}$, that is, if $\overrightarrow{v}{\leq}\overrightarrow{h'}$ (covering all d+1 dimensions). If VM v is allocated on host h, the residual capacity of the host is updated as $\overrightarrow{h'}{:=}\overrightarrow{h'}-\overrightarrow{v'}$ (line 36, 48).

To enforce the distribute constraint, the provisioning algorithm introduces a simple control that verifies if host h already contains a VM instance v_j for which the user has specified a condition $(v, v_j){\in}Distr$ before allocating a VM v on that host (line 30). If $p(v_j){=}h$ is found true, the algorithm skips that VM to the next host, hence satisfying the distribute constraint.

Lastly, to enforce the latency constraints, we introduce the notion of *forward allocate* and *reserve list*. When a VM v that satisfies all other constraints with respect to a host h is found, and if the VM v is related to other VMs v_j by the latency constraints, then v cannot be allocated until an allocation for all v_j is found. To this aim, the provisioning algorithm tentatively allocates the VM by saving the pair (v, h) in the Reserve_list (line 41) and calls the function Forward_Allocate to find an allocation for other VMs (line 43). The Forward_Allocate function first determines the set of all VMs that are related to v by the latency constraints and saves them in the priority queue V_l (line 3 in Forward_Allocate function). Each VM $v_l{\in}V_l$ is then extracted in the increasing order of T_{max} (line 6), and the set of hosts K that can be reached from h within the specified network threshold time, not conflicting the restrict and forbid constraints, is selected (lines 9-12). Each host $k{\in}K$ is considered and verified for capacity, count and distribute constraints (using similar controls as described above). When a host k is found for v_l that satisfies all the constraints and is not in turn associated with other VMs by latency constraints, the pair (v_l, k) is saved in the Reserve_list (line 23). The Forward_Allocate function is recursively called until an allocation for all VMs is determined (line 25). If an allocation is not obtained, the entires from the Reserve_list are removed (line 29), and the function resumes from another host. The Forward_Allocate function returns 1 when an allocation for all VMs is found (line 41), otherwise it returns 0 (line 33-36) to the main function.

Example 5.4.2. *Consider the Cloud infrastructure and IaaS service provider's constraints described in Example 5.3.2, and DM's request for VM instances with constraints described*

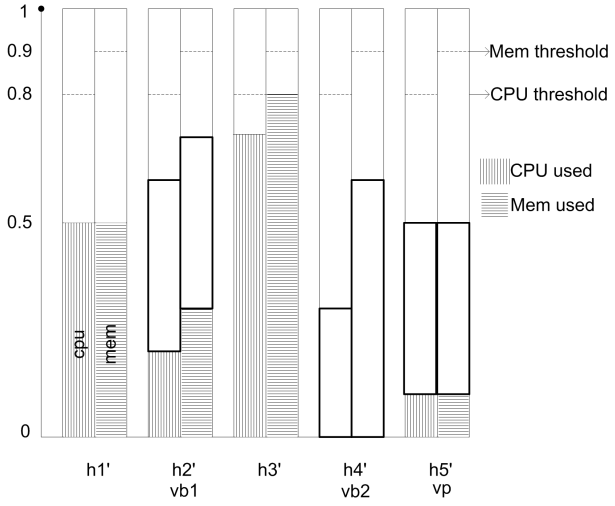


Figure 5.14: An example illustrating allocation of VMs on hosts using DM's provisioning algorithm

in Example 5.3.3. As an example, assume that some VM instances are already allocated on the hosts in cluster C_1 and occupy resources as represented in Figure 5.14.

Table 5.4 illustrates the working of the VM provisioning algorithm. On being invoked by the DM, a priority queue CL is first created by analyzing the resource availability in each cluster based on the assumed state of the infrastructure (see Figure 5.14), cluster $C{:=}C_1$ is extracted, and host h_1 is selected. The residual resource capacity in terms of CPU, memory and $count_{h_1}$ for host h_1 is $\overrightarrow{h_1'}{=}(0.3, 0.4, 2)$. The algorithm removes VM instances that are in conflict with h_1 based on restrict and forbid constraints, and therefore, VM v_p is not included in the VMreq priority queue. VMs v_{b1}, v_{b2} are considered one after another but since they do not satisfy the capacity constraint, the algorithm moves to the next host h_2. Residual resource capacity of host h_2 is determined and each VM $v{\in}V$ is analyzed based on the restrict and forbid constraints. Since none of the VM instances are in conflict with either constraints, they are included in the priority queue VMreq and vector dot-product values are calculated. VM v_{b1} is selected to be allocated on host h_2 and the capacity and distribute constraints are verified. Both the constraints are satisfied, but since v_{b1} is related to v_p by latency constraint, the pair (v_{b1}, h_2) is added to the Reserve_list and Forward_Allocate function is called. The Forward_Allocate function recognizes v_p to be the VM instance for which an allocation must be found, and generates the set of hosts $K{=}(h_2, h_3, h_4)$ that can be reached from h_2 in less than 5 latency units. Note that h_1 is not included due to the restrict constraint. Each host in K is analyzed to allocate v_p – host h_2 is in conflict with the distribute constraint since (v_{b1}, h_2) exists in the Reserve_list (i.e., tentatively allocated), and host h_3 forbids all the VMs to be allocated on it. Host h_5 satisfies

Table 5.4: An example of the working of the VM provisioning algorithm

h	$\vec{h'}$	Restrict	Forbid	Algorithm execution	Reserve_list	Allocation
1	$(0.3, 0.4, 2)$	v_p	–	VMreq=$(v_{b1}, 0.22)(v_{b2}, 0.18)$ v_{b1}: no capacity (line 29) v_{b2}: no capacity (line 29)		
2	$(0.6, 0.6, 2)$	–	–	VMreq=$(v_p, 0.13)(v_{b1}, 0.18)(v_{b2}, 0.16)$ v_{b1}: capacity ok (line 29) distribute ok (line 30) $(v_p, v_{b1}, 5) \in MaxLatency$ found (line 33) Forward_Allocate(v_{b1}, h_2) v_l=v_p K=h_2, h_3, h_5 h_2: distribute constraint in conflict (line 19) h_3: forbid constraint in conflict (line 11) h_5: capacity ok (line 18) distribute ok (line 19) $(v_p, v_{b2}, 4) \in MaxLatency$ (line 25, 3) Forward_Allocate(v_p, h_5) v_l=v_{b2} K=h_1, h_2, h_4 h_1: no capacity (line 18) h_2: no capacity (line 18) distribute constraint in conflict (line 19) h_4: capacity ok (line 18) distribute ok (line 19) latency ok (line 25)	(v_{b1}, h_2) (v_{b1}, h_2), (v_p, h_5) (v_{b1}, h_2), (v_p, h_5), (v_{b2}, h_4)	$p(v_{b1})$:=h_2 $p(v_p)$:=h_5 $p(v_{b2})$:=h_4

the capacity and distribute constraints but VM v_p is related to latency constraint with v_{b2} (v_{b1} is not considered since it already exists in the Reserve_list). Similarly, (v_p, h_5) is added to the Reserve_list and Forward_Allocate identifies (v_{b2}, h_4) to be suitable for allocation. Since all the VMs are now tentatively allocated, satisfying all the constraints (including latency), VM instances are finally allocated (VM instance v_p is allocated on host h_5, v_{b1} on host h_2 and v_{b2} on host h_4).

The VM provisioning algorithm allows the DM to realize the chosen dependability policy (e.g., dep_sol s_3 with property p_3 in our examples) and deliver its service to the users' application. We note that the provisioning algorithm actually deploys users' VM instances in the Cloud while satisfying all the constraints specified by the DM. Therefore, successful execution of this algorithm ends the design phase of the two-stage approach adopted by the service provider. At this point, the runtime phase begins and monitoring of relevant attribute values is performed. For example, a set of rules such as r_1:n.replicas$\geq$2, r_2:recovery time$\leq$8ms can be defined and corresponding attribute values be verified. The current allocation of the users' application (e.g., $p(v_{b1})$:=h_2, $p(v_p)$:=h_5, $p(v_{b2})$:=h_4) must be adapted whenever the policy rules are violated due to infrastructure changes, affecting the availability or performance of the application. The adaptive resource management algorithm is discussed in the next section.

5.4.3 Adaptive Resource Management

Cloud environment is highly dynamic in terms of task activation, bandwidth availability, component failures and recovery. As a consequence, static deployment strategies that perform only initial allocation (such as the $p{:}\mathcal{V}{\rightarrow}\mathcal{H}$ function) may not provide satisfactory results at runtime, and an adaptive approach to resource management is necessary.

One method to respond to system changes is to recompute the allocation from *scratch* using the $p{:}\mathcal{V}{\rightarrow}\mathcal{H}$ function. However, this method is rather naive and may not scale well during runtime. To this aim, we describe a heuristics-based approach that minimizes the performance and availability degradation of users' applications due to various system changes. This heuristic is realized as the online controller and introduced in the envisioned Cloud environment. The online controller uses the system's monitoring information (e.g., application workload, server's failure behavior, processor and bandwidth usage), and re-deploys the applications as a response to the events that may violate the application's availability or performance goals (i.e., whenever $f(s, \mathcal{R}){\rightarrow}false$). In particular, it generates a new allocation for the users' applications by deploying new application task replicas in case of host failures and by migrating individual tasks on (other working hosts) orthogonally across different deployment levels in the system. If the output generated by the online controller is unfeasible for the user's application, the DM must restart the service delivery process by executing the matching and comparison processes, specifying allocation constraints, and running the VM provisioning algorithm.

The online controller provides dependability support during runtime, and corresponds to the Fault Masking Manager of the DM. The activities required to change the current allocation status and re-deploy users' applications are realized using the virtualization technology constructs. By treating the task replicas as individual tasks, the online controller generates the new configuration in terms of the following actions:

- Launch(v, h): Due to system failures, the controller may identify that new replicas of a given task must be created. To realize this function, it instantiates a VM v, hosting the task replica $t{\in}T$, on the physical host $h{\in}\mathcal{H}$ using the Launch(t, h) action.

- Migrate(v, h_i, h_j): As a response to performance or availability degradation, the online controller may have to change the current location of a subset of task replicas. For example, to respond to network congestion in cluster C_1, the online controller may want to move task t_1 (initially hosted in C_1) to another cluster C_2. This function can be realized using the Migrate(t, h_i, h_j) action by specifying that VM instance deployed on host $h_i{\in}\mathcal{H}$, containing a task replica $t{\in}T$, must be moved to host $h_j{\in}\mathcal{H}$.

- Delete(v, h): Due to performance overhead, the online controller may need to reduce the replication level of a task. This action can be specified using the Delete(t, h) construct that removes the VM instance, hosting task replica $t{\in}T$, from host $h{\in}\mathcal{H}$.

1: **RECONFIGURE**
2: **INPUT** $p{:}\mathcal{V}\rightarrow\mathcal{H}$, $T_i{\in}M$, $\mathcal{H}$, $Restr$, $Forbid$, $Count$, $Distr$, $MaxLatency$
3: **OUTPUT** Set containing actions $Action$
4: $Action{:=}\emptyset$
5: /* If real availability is lower than the desired availability*/
6: **if** $\text{Avail}^r < \text{Avail}^d$ **then**
7: /*Identify the application tasks with replica failures*/
8: **for each** $T_i \in a$ with $\text{nR}(T_i) < |T_i|$ **do**
9: /*Create task replicas in the original deployment level DL*/
10: /*without violating the performance goals*/
11: **while** $(\text{Avail}^e \geq \text{Avail}^d \ \vee \text{nR}(T_i) \ \geq |T_i|) \wedge (\text{Perf}^e \geq \text{Perf}^d)$ **do**
12: /*Include the launch action in $Action$*/
13: $\forall t_{i,j}{\in}T_i$, $map(t_{i,j}){\in}DL$,
14: $Action := Action \cup \{\textbf{Launch}(t_{i,j}, map(t_{i,j}))\}$
15: **end while** /*Expected availability or replication level is met*/
16: **end for**
17: /*If expected availability is still lower than the desired one*/
18: **while** $\text{Avail}^e < \text{Avail}^d$ **do**
19: /*Move task replicas to the higher deployment levels DL*/
20: **if** $\forall T_i{\in}a$, $t_{i,j}{\in}T_i$, DL, $map(t_{i,j}){\in}DL$ s.t. $\text{Perf}^e \geq \text{Perf}^d$ **then**
21: /*Change in configuration by migrating task is possible*/
22: $Action = Action \cup \{\textbf{Migrate}(t_{i,j}, p(v_{i,j}), map(t_{i,j}))\}$
23: **else if**
24: /*Increase number of replicas to improve availability*/
25: /*Traverse from highest deployment level to lowest*/
26: $\forall T_i \in a, DL$, $\text{Perf}^e \geq \text{Perf}^d$,
27: $Action = Action \cup \{\textbf{Launch}(t_{i,j},\ map(t_{i,j}))\}$
28: **end while**
29: /* If real performance is lower than the desired performance*/
30: **if** $\text{Perf}^r < \text{Perf}^d$ **then**
31: /*Identify the application tasks with affected response time*/
32: **for each** $T_i \in a$ with $L(T_i) > L_{max}$ **do**
33: /*Delete task replicas in the original deployment level DL*/
34: /*without violating availability goals*/
35: **while** $(\text{Perf}^e \geq \text{Perf}^d \vee L(T_i) \leq L_{max}) \wedge (\text{Avail}^e \geq \text{Avail}^d)$ **do**
36: /*Include the delete action in $Action$*/
37: $\forall t_{i,j}{\in}T_i$, $map(t_{i,j}){\in}DL$,
38: $Action := Action \cup \{\textbf{Delete}(t_{i,j},\ map(t_{i,j}))\}$
39: **end while** /*Expected performance or latency obtained*/
40: **end for**
41: /*If expected performance is still lower than the desired one*/
42: **while** $\text{Perf}^e < \text{Perf}^d$ **do**
43: /*Move task replicas to the lower deployment level DL*/
44: **if** $\forall T_i{\in}a$, $t_{i,j}{\in}T_i$, DL, $map(t_{i,j}) \in DL$ s.t. $\text{Avail}^e \geq \text{Avail}^d$ **then**
45: /*Change in configuration by migrating task is possible*/
46: $Action = Action \cup \{\textbf{Migrate}(t_{i,j}, p(v_{i,j}), map(t_{i,j}))\}$
47: **else**
48: /*Decrease the number of replicas to improve performance*/
49: /*Traverse from lowest deployment level to highest*/
50: $\forall T_i \in a, DL$, $\text{Avail}^e \geq \text{Avail}^d$,
51: $Action = Action \cup \{\textbf{Delete}(t_{i,j},\ map(t_{i,j}))\}$
52: **end while**
53: **return** $Action$ /*and call p to schedule the actions*/

Figure 5.15: Pseudo-code algorithm for generating a new configuration plan

The online controller uses a mapping function $map{:}\mathcal{V}{\rightarrow}\mathcal{H}$ that behaves similarly to p but performs only a tentative search. That is, the mappings generated by *map* do not reflect on the infrastructure and must be explicitly *committed* using $p{:}\mathcal{V}{\rightarrow}\mathcal{H}$. For the sake of clarity, we now introduce the following notations: $\mathrm{nR}(t_i)$ denotes the number of replicas of application task t_i; Avail^r, Avail^d and Avail^e denote the real, desired and expected availability of the user's application. Avail^r is computed using the real attribute value $attr{\in}A$ at runtime, Avail^d is the availability value specified by the user in p_c, and Avail^e is the value that DM estimates for a given configuration using Markov models and fault trees. Similarly, Perf^r, Perf^d and Perf^e denote the application's performance metrics.

Figure 5.15 depicts the pseudo-code of the algorithm that computes the set of actions that, when *committed*, generates a new allocation for a given users' application. It takes the current allocation, system status, application tasks and the sets specifying allocation constraints as input, and generates the sequence of actions that brings the system to a new allocation state. This algorithm is invoked when a failure or performance degradation event happens. The algorithm consists of two main conditions, one concerning availability (dependability) violation due to system failures (lines 5–28) and other concerning performance degradation (lines 29–52). If the real availability of an application is less than the desired one, the online controller first identifies the task replica failures and tentatively launches new replicas at the same deployment level using the *map* function. We note that the launch action is performed only until the current replication level is same as the original level and performance goals are not violated (lines 8–16). When addition of replicas does not satisfy the requirements, the algorithm tries to move task replicas to a higher deployment level using the Migrate action (note that the availability increases with increasing deployment levels). This action allows the online controller to generate the new allocation solution without increasing the resource consumption costs. If the performance condition conflicts by moving tasks to higher deployment levels, additional replicas must be created to improve the availability. To create new replicas, the online controller starts from higher deployment levels and moves gradually to lower levels, creating the replicas at the level where availability and performance goals are fulfilled. These actions are realized using the migrate and launch actions (lines 18–28). When users' availability requirements are satisfied, the algorithm realizes the actions in *Action* using the $p{:}\mathcal{V}{\rightarrow}\mathcal{H}$ function. In contrast, when real performance is less than desired performance, instead of launching new replicas, VM instances are deleted, and instead of moving higher in the hierarchy, migration takes place to lower deployment levels. These actions are based on the observation that decrease in the deployment or replication level, improves the application performance. We note that the online controller is invoked only when an application experiences failures or performance degradation, and therefore, it is suitable for long-running tasks; short-running tasks are practically managed by the DM during initial deployment.

5.5 Simulation Results

This section reports the simulation results of the experiments conducted to evaluate the VM consolidation, VM provisioning, and adaptive resource management schemes.

5.5.1 Virtual Machine Consolidation

This section evaluates the vector dot-product approach used by the DM to perform VM consolidation (see Section 5.4.1). First, the background related to consolidation schemes is described, and then the results of our approach are compared to the state-of-art schemes.

Background. First Fit Decreasing (FFD) is one of the most widely used heuristic to solve the one dimensional bin packing problem. In FFD, items are sorted by size in decreasing order and then placed sequentially in the first bin that has sufficient capacity. The algorithm in Figure 5.12 also uses a variant of the first first decreasing algorithm to perform VM provisioning. In general, there is no standard way of generalizing FFD for the multi-dimensional case. As a consequence, several VM consolidation heuristics that are currently being used in research prototypes and real VM management tools adopt different approaches. For example, Sandpiper [136], a research system that enables live migration of VMs around overloaded hosts, uses a heuristic that takes the product of CPU, memory, and network loads. We denote this approach as *FFDProd* and compute the heuristic value as follows.

$$hv = \Pi_{x \leq d, v \in \mathcal{V} | p(v)=h} v[x]h[x] \tag{5.8}$$

Another approach is to define a vector such as:

$$w[d] = \sum_{x \leq d, v \in \mathcal{V} | p(v)=h} a[x]v[x] \tag{5.9}$$

where the vector $a(1, \ldots, d)$ is a scaling vector designed to *i)* normalize demands across each dimension, and *ii)* weight the demands according to their importance or likelihood of being a bottleneck for consolidation. This approach is denoted as *FFDSum* in the experiments below.

An application placement controller by IBM performs application consolidation handling resource demands for CPU and memory [120]. This work combines the two dimensions into a scalar by taking the ratio of the CPU demand to the memory demand. Although this heuristic does not exactly suit the following *FFDAvgSum* heuristic, it shares the characteristic of ignoring complementary distributions of resource demands in various dimensions. The *FFDAvgSum* heuristic assigns the average demand $avgdem = 1/d \sum_{\forall x \leq d} v[x]$ across all the dimensions for a given VM instance. Similarly, in *FFDExpSum*, the weights are considered exponential in $avgdem$, where $avgdem = \exp(k.avgdem)$ for a suitable constant k (k=0.01 in the experiments below).

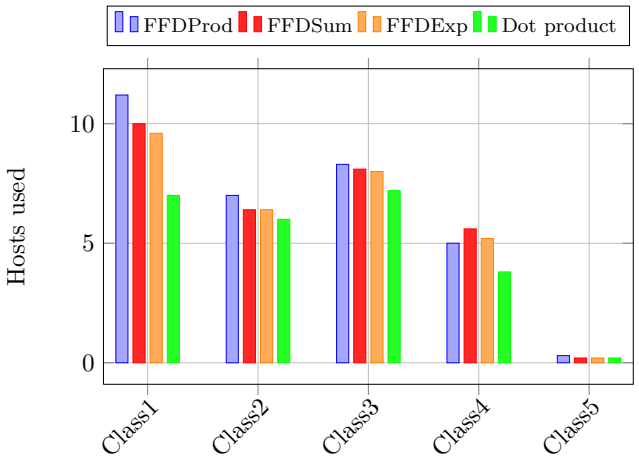


Figure 5.16: Number of hosts used by each consolidation algorithm, for each input class, with 100 VM instances of 2 dimensions

The DM uses the vector dot-product heuristic to perform VM consolidation, similarly to [106, 117]; but differs in the way the scarcity (weight) of the host is taken into account.

Input parameters. Since realistic workloads vary widely across organizations in terms of heterogeneity and resource requirements, this experiment is run on synthetic instances generated randomly from different distributions. The goal of the experiment is to compare the quality of solution from various approaches. The input data is generated in the manner that allows testing across different correlations and dimensions. In particular, five classes of VM sizes are defined, and drawn randomly and independently for each dimension. The range of each class is set to [0.1, 0.4], [0.1, 1], [0.2, 0.8], [0.05, 0.2], [0.25, 0.1], allowing different context for VM requests. Each algorithm is coded in C++ and executed on a machine having Intel i7-2860QM 2.50GHz processor, with 16GB of memory, running Windows 7 operating system.

Results. Figure 5.16 shows the number of bins used by each algorithm, for each class of input. This experiment considers the request for 100 virtual machines for each input class. We can see that *FFDSum* and *FFDExp* provide similar results, while *FFDProd* provides least performance guarantees, for all input classes. This underlines that the weighting parameter (absent in *FFDProd*) is a significant factor for consolidation. In contrast to the algorithms, the vector dot-product method best utilizes the resources in the infrastructure. The performance is particularly high for input classes 1, 4 and 5.

Figure 5.17 compares the performance of *FFDProd*, *FFDExp* and *Dot product* methods for 2, 3 and 10 dimensions respectively. This experiment considers allocation of 2500 VM instances given input class 5. *FFDSum* is not considered since its performance is similar to

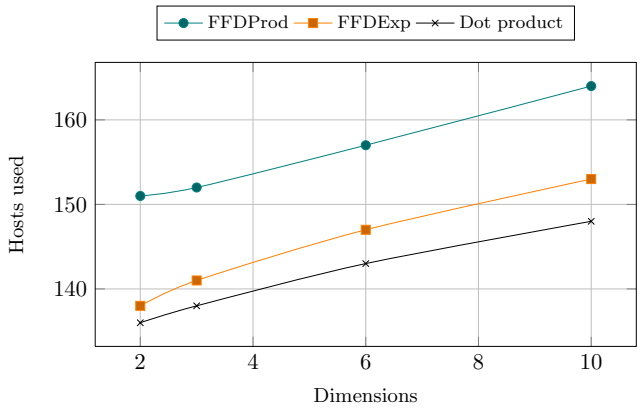


Figure 5.17: Number of hosts used by FFDProd, FFDExp and Dot product algorithms, with varying number of dimensions

FFDExp. We can see that the number of hosts utilized by the dot product scheme is less that the other two algorithms for all dimensions. The performance results are particularly improve significantly as the number of dimensions increase.

5.5.2 Virtual Machine Provisioning

This section reports the results of the experiments performed to evaluate the scalability of the virtual machine provisioning algorithm illustrated in Figure 5.12. An infrastructure consisting of 200 hosts is considered and each host initialized with a random resource availability value. The infrastructure is divided into 4 clusters, each having 50 hosts; the division is performed in the order of the host's identifier (e.g., C_1=$\{h_1, \ldots, h_{50}\}$). During each experiment, 15 hosts are randomly selected and included in the set $Forbid$. Similarly, if network latency between the hosts within a cluster is x, then latency between various clusters in varied from values between $1.1x$ to $1.5x$.

Figure 5.18 illustrates the amount of time required by the algorithm to provision resources in three configurations *i)* no constraints specified (A), *ii)* all other constraints but the latency constraint specified (B), and *iii)* all constraints specified (C). The user's request for 5, 10, 15 and 20 virtual machine instances is provided as input to the algorithm. For each request, the algorithm is run 10 times and the results reported here are the average of 10 executions. Given a user request, pairs of tasks are randomly included in the $Distr$, $MaxLatency$ sets, and two clusters are randomly included in the $Restr$ set corresponding to each virtual machine. We can see that the execution time of the algorithm increases as the size of the users' application increases. Algorithms A and B have similar distributions, and the overhead in execution time due to additional constraints (except latency) is about 4.5%. On the other hand, when the latency constraints is introduced, the algorithm (C)

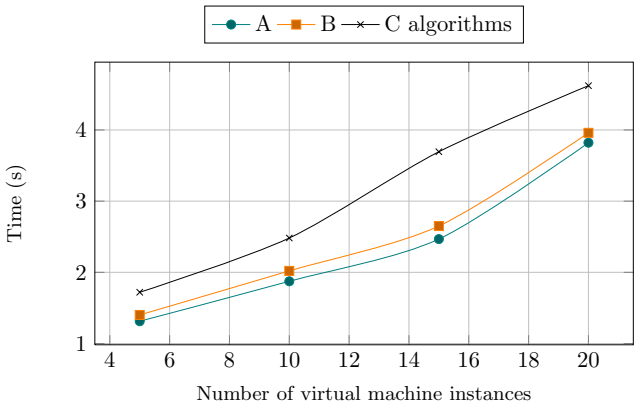


Figure 5.18: Time to compute allocation for different sizes of users' applications

takes about 14% additional time to allocate 20 virtual machines (wrt algorithm A). The distribution of algorithm C is different, from that of algorithms A and B, due to repetitive execution of the Forward Allocate function.

5.5.3 Adaptive Resource Management

This section reports the simulation results of the experiments conducted to evaluate the online controller. In particular, the validity of the controller in terms of *i)* the time required to compute a new configuration using the algorithm in Figure 5.15, *ii)* increase in overall availability of an application, and *iii)* improvement with respect to the performance in varying system contexts.

Setup. The hardware failure rates are provided by many companies in the form of tables [121]. However, the task of attributing the cause of failures and estimating the mean time to failure for software components (e.g., hypervisor) is difficult. To this aim, the input parameter values are derived here from [74] (e.g., 2.654e+003 and 3.508e-001 as mean time to failure and recovery respectively for virtualized hosts) and the ORMM Markov analysis tool [61] is used to obtain the output measures. To make the results applicable for systems with different MTBF and MTTR values, normalization of all times to MTBF is performed, and the MTTR is varied over a range from 0.01 to 2.0. This indicates variation in repair times from 10% to 200% of actual MTBF, hence providing different availability values. Similarly, the parameters for performance are obtained using the layered queueing network solver [47]. We note that Markov analysis tool and queueing network solver are used offline, and output parameter values are used to configure the online controller simulator written in C++. The simulation is executed on a machine having Intel i7-2860QM 2.50GHz processor, with 16GB of memory, running Windows 7 operating system.

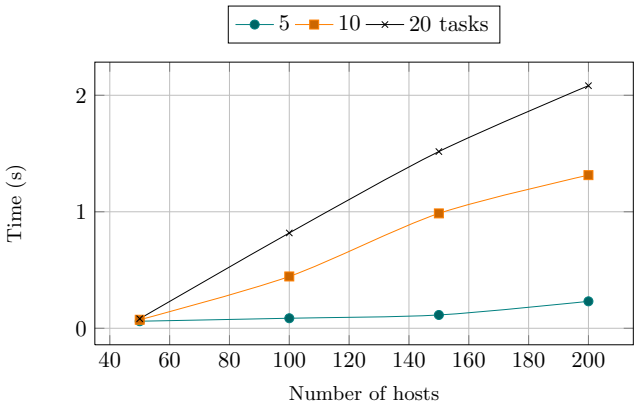


Figure 5.19: Time to compute the new configuration, varying number of hosts and tasks

The Cloud infrastructure is configured by randomly initializing the hosts with different amounts of residual resources. This forms the basis for the online controller to manage VM instances of a given application on the current resource status of the Cloud. The utilization of hosts are updated after launch, migrate and delete actions, providing results on incremental resource management. Varying network latency values between deployment levels, the MTBF and relative MTTR rates are also initialized similarly. For example, network latency between VM instances vary depending on the deployment configuration (if replicas share a host, rack, cluster, or a data center). The network latency within a host is considered 0; if latency between two hosts in a rack is x, latency is set to $1x$, $1.5x$ and $2.5x$ for different racks, clusters, and data centers respectively. The applications with different number of tasks (see below) are selected and replicated task sets randomly chosen. The simulation results presented here are the mean values of ten executions of each configuration.

Processing time evaluation. This experiment studies the amount of time it takes for the online controller to compute the new configuration for applications with 5, 10 and 20 tasks, on an infrastructure containing 50 to 200 hosts. Figure 5.19 illustrates how the processing time varies for different contexts. For smaller size instances of applications and infrastructure, the solution can be computed in the order of a few milli-seconds. When the application contains 20 tasks, and infrastructure has 200 hosts, the processing time is about 2 seconds. In particular, we observe higher processing time when cluster level failures affect multiple task replicas. Although computing new configuration has acceptable scalability, note that the amount of time to actually reconfigure the system may be larger due to several system parameters (e.g., the time to migrate a VM may be in the order of minutes, particularly when the VM size is large, and the target host is connected via Internet).

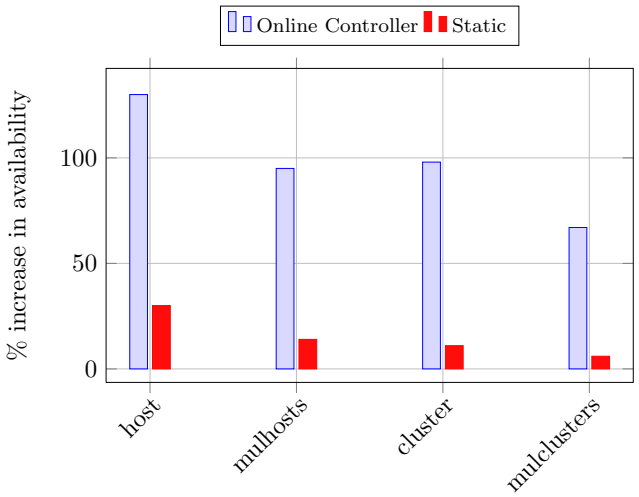


Figure 5.20: Percentage increase in availability due to reconfiguration

Availability and performance evaluation. For simplicity, the virtual machines are allocated as per the p function, using the first-fit bin-packing strategy. This allocation is considered *static*, and simulations are then compared against the static scheme. To evaluate the increase in availability, failures following the MTBF values are introduced at different deployment levels (hosts and clusters) in the infrastructure. Similarly, to evaluate the performance, a deployment level (hosts and clusters) is randomly selected and an increase in the network latency connecting those resources is assumed. For each failure and change in network latency, the online controller is invoked to compute the new configuration.

The percentage increase in the availability of an application with 10 tasks, comparing static approach and the online controller approach, for different failure levels, is calculated. A cluster failure implies assuming all the hosts in that cluster have failed. Figure 5.20 shows the difference between the availability levels. In case of single host failures (that have least MTBF values), the online controller is estimated to improve an application's availability by 120 percent (when compared to static allocation). The increase in availability is about 95 percent in case of multiple hosts and single cluster failures, whereas, in case of multiple cluster failures (with higher MTBFs), application's availability is estimated to improve by 70 percent when compared to static deployment methods.

Similarly, the change in the response time of an application is calculated by increasing the network latency at different levels in the infrastructure. Figure 5.21 illustrates that online controller approach can significantly reduce the performance degradation among applications when changes in the network latency affects multiple clusters within a data center. On the contrary, the percentage improvement in performance due to disruptions

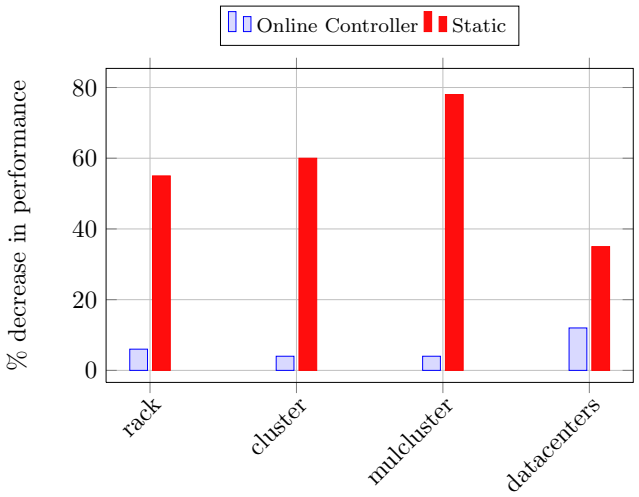


Figure 5.21: Percentage change in term of response time/performance degradation due to reconfiguration

at data center level is marginal. Since, the online controller regenerates or migrates the task replicas in the event of failures or performance degradation as opposed to the static scheme, the results cannot be consistently compared. Nevertheless, the simulation results clearly show that the adaptive resource management algorithm of the online controller can provide high levels of graceful service degradation to the users.

5.6 Chapter Summary

In this chapter, we presented a set of techniques that allow the service provider to realize DM, and consequently, offer dependability as a service to users' applications. First, we discussed an approach for selecting low level dependability mechanisms based on high level users' requirements, and then presented a set of constraints that allow specification of dependability and performance conditions inherent to the specific configuration of the chosen mechanism. The proposed solution can be extended to include other constraints (e.g., related to vulnerability of hosts and applications, as shown in Chapter 6) in a straight-forward manner. We presented a virtual machine provisioning scheme whose objective is to improve the profits for the IaaS service provider while satisfying various dependability constraints. Furthermore, we presented an approach to adapt the current allocation of a given application to balance its dependability and performance requirements at runtime. Finally, we provided some simulation results that indicate the scalability and performance of our algorithms, highlighting the practical applicability of the proposed approach.

Using the techniques proposed in this chapter, dependability of users' applications deployed using IaaS services is improved manifold, particularly when compared to the certification scheme presented in Chapter 3. Users are not only relieved from implementing low level dependability mechanisms but also are capable of obtaining and changing specific dependability properties of their applications based on business needs. This solution denotes the second of the three levels of dependability offered in this thesis.

In the next chapter, we consider complex applications, and present a set of algorithms that further improves dependability and robustness of users' applications.

6

Secure Application Deployment and Execution

In the previous chapter, we discussed techniques to realize the notion of dependability as a service. The resource management algorithms improved dependability by means of placement constraints and supported infrastructure providers in achieving their business goals. In contrast, in this chapter, we consider complex applications, and present advanced resource management schemes that deploys applications with improved optimality when compared to algorithms in Chapter 5. Our algorithms improve dependability of a given application by minimizing its exposure to existing vulnerabilities, while being subject to same dependability policies and resource allocation constraints as in Chapter 5. In particular, the following aspects are considered:

- An approach to model the application deployment problem as a task allocation problem with a security-oriented objective, subject to various dependability constraints, and a solution based on the A* algorithm for searching the solution space.

- The design of an interruptible, elastic, task allocation algorithm whose optimality improves as the execution time provided to it increases. This algorithm is beneficial in the context of mission-critical applications where the processing time available for resource management is often limited and varying.

- A cost-effective approach to harden the set of computational resources that have been selected for executing a given application in order to provide applications with further protection, availability and fault tolerance.

For a given user request, if we replace the algorithms in Chapter 5 with secure application deployment algorithms of this chapter, the dependability, robustness, and security of the algorithm improves, thus denoting the third level of dependability solution offered in this thesis.

6.1 Introduction

Cloud infrastructures are typically prone to a number of failures and vulnerable to a wide range of cyber-attacks. Such failures and vulnerabilities evidently have an impact on the users' applications ranging from simple response time degradation to complete unavailability and loss of critical data. Consequently, a number of solutions have been proposed in the literature to address users' security concerns. Such solutions improve security either by designing the system with network hardening tools such as intrusion detection systems and firewalls, or by developing applications using security measures such as data obfuscation and memory management. However, they fail to take into account the complex interdependencies between the network infrastructure, application tasks, and residual vulnerabilities in the system. This implies that, using existing solutions, it is not possible to remediate all existing vulnerabilities and applications have to be executed on the infrastructure containing multiple and interdependent vulnerabilities.

The approach presented in this chapter addresses users' security concerns by identifying an execution plan that minimizes the risk that residual vulnerabilities may impact their applications. In particular, it discusses an application-centric framework that combines resource management and network hardening techniques. First, the current vulnerability distribution of the Cloud is considered and users' applications are deployed in the manner that minimizes application's exposure to the residual vulnerabilities in the infrastructure. Then, network hardening techniques are applied to protect the deployed applications from possible cyber-attacks. We note that, for this approach to be effective, existing techniques for identifying network vulnerabilities, generating attack models, and assessing the risk that residual vulnerabilities may pose to each element of the computing infrastructure must be integrated within the overall framework.

To deploy users' applications in most secure manner possible, the task allocation problem is formulated with a security-oriented objective, and an algorithm based on A* search scheme is provided to solve it. In addition, elastic resource management algorithms that combine the A* search scheme with the Anytime processing approach are provided [53, 81]. These algorithms are elastic in the sense that they *i)* provide an initial sub-optimal result rather quickly and continue the execution process to generate solutions with improved optimality; that is, optimality of the algorithm improves as the processing time increases, *ii)* converge to an optimal solution eventually, and *iii)* can be interruptible i.e., the best solution computed so far is returned whenever the algorithm is interrupted. The users' applications are deployed according to the solution returned by the allocation algorithm, relevant resources are further hardened using the notion of attack graphs, and monitoring of resources begin. When the monitoring system detects an anomaly, redeployment algorithm that adapts the application's current allocation is invoked so as to minimize the impact of system changes. The redeployment algorithm also dynamically adjusts the optimality level of the allocation solution based on the magnitude of system changes.

6.1.1 Chapter Outline

The remainder of this chapter is organized as follows. Section 6.2 describes the system model, and formulates the secure application deployment problem as a task allocation problem. Section 6.3 discusses an approach to solve the task allocation problem using the A* algorithm. Section 6.4 presents a scheme that transforms A* to an elastic task allocation algorithm and discusses the design of elastic redeployment algorithm. Section 6.5 presents a network hardening scheme that further protects the resources used the application. Section 6.6 outlines some concluding remarks.

6.2 System Model

Consider the scenario in which a user wishes to execute her mission-critical applications (missions) using the Cloud-based IaaS model. To compute the secure mission execution plan, she interacts with the resource manager that analyzes the vulnerability distribution of the infrastructure and deploys her missions such that their exposure to vulnerabilities is minimum. This section presents the infrastructure and mission model (slightly modified with respect to Chapter 5), and formalizes the secure-mission deployment problem.

Cloud infrastructure. Despite careful security engineering, a number of vulnerabilities remain in the Cloud infrastructure, and allow malicious adversaries to launch different types of cyber-attacks. For example, an attacker may exploit vulnerabilities in services such as *ftp*, *rsh*, and *sshd* to gain desired access privileges on a given host. Such exploits can in turn be used to compromise users' missions deployed in the system. Vulnerabilities and attack paths in the network can be analyzed using vulnerability scanners, and approaches based on attack graphs, dependency graphs, and attack surfaces. Analysis tools can also be extended with probabilistic schemes and ranking methods to quantify the vulnerability level of individual hosts. For simplicity, in this chapter, assume that a vulnerability value Vul_h is pre-computed for each host $h \in \mathcal{H}$ in the infrastructure by adopting one of the existing techniques. A physical host $h \in \mathcal{H}$ can then be characterized by a vector $\overrightarrow{h} = (h[1], \ldots, h[d+1])$, where the first d dimensions represent the residual capacity of each resource on the host, and the last dimension denotes its vulnerability score Vul_h. Similarly to the system model in Chapter 5, elements of $\overrightarrow{h}$ are normalized to values between 0 and 1. For example, $\overrightarrow{h} = (\text{CPU, Mem, Vul}_h) = (1, 1, 1)$ implies that CPU and memory are fully available and the host is extremely vulnerable.

Missions. The user's mission-critical application or mission M can be seen as a composition of a set tasks $M = \{\tau_1, \ldots, \tau_m\}$. This model-independent definition allows representation of different software architectures for the missions (e.g., web services, business processes, scientific applications) as well as different formalisms (e.g., Petri

Nets, work flows). For example, a mission can be a three-tier web application realizing an e-Commerce service or a scientific tool with tasks performing graph theoretical calculations on geographical maps. Intuitively, a mission is successful if *i)* all the tasks start from a correct initial state, perform their operations, and generate the correct output in a specified amount of time, and *ii)* the protocol that composes the information from individual tasks can justifiably be trusted. Each task in the mission can be associated with a tolerance value *tol* when it is implemented using some security mechanisms (e.g., memory management guards to protect from buffer overflow attacks). Intuitively, the *tol* value provides an estimate of the maximum level of vulnerability that the task can be exposed to without compromising its successful completion. The user (or FTM) may replicate critical tasks of the mission to improve its fault tolerance and resilience, and obtain a set of task replicas $R_k{=}\{\tau_1^1, \ldots, \tau_m^{|r_k|}\}$ for each task, and the overall mission becomes a composition of a set of replicated task sets $T = \{t_i\} = \bigcup_{\tau_k \in M} R_k$. Each task replica can be treated as independent task for the purpose of mission deployment. Similarly to hosts, a task $t{\in}T$ can be characterized by a vector $\overrightarrow{t}{=}(t[1], \ldots, t[d+1])$, where the first d dimensions represent the task's requirements for specific computing resources and the $d+1^{th}$ dimension is the task's risk tolerance value $tol(t)$. The elements of $\overrightarrow{t}$ are also normalized to values between 0 and 1.

Problem formulation. The first step to securely execute a given mission is to deploy the mission tasks in the Cloud such that their exposure to vulnerabilities is minimized. Since requests for mission deployment may arrive at any time, the deployment strategy must consider the current resource allocation and vulnerability status of the Cloud, and the allocation for the new mission must be computed based on the availability of currently unused resources. After each mission deployment, the resource allocation and vulnerability status of the system must be updated accordingly.

Mission deployment can be modeled as a task allocation problem that is defined by the function $p{:}\mathcal{V}{\rightarrow}\mathcal{H}$ which maps each task $t{\in}T$ to a physical host $h{\in}\mathcal{H}$, assuming one-to-one correspondence between tasks and virtual machine images satisfying $\overrightarrow{t}$. The binary variable p_{ij} denotes the truth value of $p(v_i) = h_j$; that is,

$$(\forall t_i \in T, v_i \in \mathcal{V}, h_j \in \mathcal{H}) \quad p_{ij} = \begin{cases} 1 & \text{if } p(v_i) = h_j \\ 0 & \text{otherwise} \end{cases}$$

The objective is to minimize mission's exposure to system's vulnerabilities. We note that each time a task deployed in virtual machine v_i is allocated on a host h_j, new vulnerabilities are potentially introduced on the host. As a consequence, the vulnerability score of h_j may increase by an amount $\Delta\text{Vul}_{v_i h_j}$. Furthermore, though multiple hosts may have similar configurations and, consequently, similar vulnerability scores, their vulnerability scores may vary significantly at runtime, as tasks are dynamically allocated and deallocated. Let $\text{Vul}^*_{h_j}$ denote the vulnerability score of host h_j after mission deployment. The objective is

to find, among all possible allocations $p \in \mathcal{P}$, the allocation that minimizes the largest h_j amongst all the hosts involved in the mission, that is

$$\min_{p \in \mathcal{P}} \max_{h_j \in \mathcal{H} | \exists t_i \in T, v_i \in \mathcal{V}, p(v_i) = h_j} \text{Vul}^*_{h_j} \tag{6.1}$$

Each allocation must satisfy the following constraints to ensure the dependability of the mission (performance attributes are omitted here for simplicity):

- *Consistent allocation:* The following properties must hold.

$$(\forall t_i \in T, v_i \in \mathcal{V}) \quad \sum_{h_j \in \mathcal{H}} p_{ij} = 1 \tag{6.2}$$

$$(\forall h_j \in \mathcal{H}, 1 \leq x \leq d) \quad \sum_{v_i \in \mathcal{V}} p_{ij} \cdot v[x] \leq h[x] \tag{6.3}$$

  Equation 6.2 implies that each task must be allocated on a single physical host (each host can accommodate multiple tasks). Equation 6.3 implies that the amount of resources consumed on a single host cannot exceed its capacity in any dimension.

- *Distribution:* All replicas of a task must be allocated on different hosts to avoid single points of failure.

$$(\forall \tau_k \in M)(\forall \tau'_k, \tau''_k \in R_k) \quad p(\tau'_k) \neq p(\tau''_k) \tag{6.4}$$

- *Vulnerability tolerance:* A task t can be mapped only to hosts h whose vulnerability score $\text{Vul}_h{=}h_j[d{+}1]$ is less than the vulnerability tolerance $tol(t){=}t_i[d{+}1]$ of that task.

$$(\forall h_j \in \mathcal{H}, t_i \in T, v_i \in \mathcal{V}) \quad t_i[d+1] \geq p_{ij} \cdot h_j[d+1] \tag{6.5}$$

In general, the task allocation problem can be solved using any search algorithm. However, in the context of missions (mission-critical applications), optimality and scalability of the search algorithm becomes of critical importance. An approach to design such algorithm is discussed in the following sections.

6.3 Mission Deployment using A*

A* is a widely used best-fit search approach that provides optimal results in acceptable execution time. This section presents an approach to solve the mission deployment problem using A*. In particular, Section 6.3.1 briefly describes how to construct the data structure and cost function. Section 6.3.2 presents the A* search scheme to solve the secure task allocation problem, and Section 6.3.3 provides some simulation results.

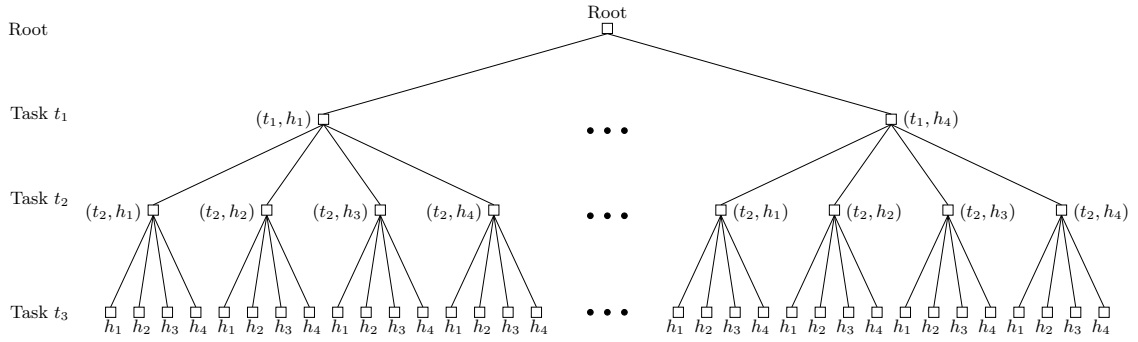


Figure 6.1: State-space tree for a network with four hosts and mission with three tasks

6.3.1 Data Structure and Cost Function

To enable A* exploration, the overall state-space is represented as a tree, where each state $s(v_i, h_j) \in \mathcal{V} \times \mathcal{H}$ represents a possible choice of allocating task t_i, deployed in virtual machine v_i, on host h_j. Figure 6.3 illustrates the complete state-space tree for allocating a mission with three tasks on to an infrastructure consisting of four hosts. The *root state* is the initial state where no tasks have been allocated yet and the *goal state* is the leaf state in which all the tasks have been allocated. Given a state s, an *operation* of the A* algorithm generates the set of feasible child states for s. This is done by choosing the next task to be allocated and identifying all compatible hosts, as described in the next section. A *solution path* is the path from the root state to the goal state. The objective is to find the solution path that minimizes the vulnerability score of the mission.

States generation. The set T of tasks to be allocated is initially sorted in increasing order of the risk tolerance value *tol*. The i^{th} task in the sorted list corresponds to the i^{th} level of the state-space tree. The number of levels in the tree is equal to the number of tasks to be allocated plus the root state (level 0).

Each state is generated dynamically when A* explores the overall state-space to find the solution path. The $getSuccessors$ function takes the current state $s(v, h)$ and generates the child states of state s. First, the task t^* following t in the sorted list is selected, and the set of hosts satisfying the dependability constraints w.r.t. t^* are shortlisted (in case of root state, the first task in the sorted list is considered). A virtual machine v^* is chosen for the selected task t^* and, a state $s(v^*, h_j)$ corresponding to each shortlisted host h_j is generated and included as the child state (successor) of s.

The dependability constraints are ensured as follows. First, the vulnerability tolerance constraint is enforced by selecting hosts $h_j \in \mathcal{H}$ that satisfy $t^*[d+1] \geq h_j[d+1]$. Next, the capacity constraint enforced by select the hosts that satisfy resource requirements

for task v^*; that is, the hosts h_j for which $v^*[x] \leq h_j[x]$ for all $x \in [1, d]$. Finally, the distribute constraint is enforced w.r.t. all the hosts by verifying that there does not exist a state $s(v', h_j)$ in the current solution path such that v^* and v' (corresponding to t^* and t') belong to the same replicated task set R_k.

Cost function. The order in which the states must be expanded is determined using the following cost function:

$$f_{vul}(s) = g_{vul}(s) + h_{vul}(s) \tag{6.6}$$

$g_{vul}(s)$ denotes the aggregate vulnerability score associated with the allocation path from the root state to state s. It is assumed to be ∞ if state s is not expanded yet. $h_{vul}(s)$ is a heuristic estimate of the minimum additional vulnerability associated with completing the allocation of the mission's tasks. Therefore, $f_{vul}(s)$ is the vulnerability score estimate of the complete allocation from root state to the goal state through s.

The $g_{vul}(s)$ can be computed as follows.

$$g_{vul}(s) = g_{vul}(parent(s)) + \text{Vul}_{h_j} + \Delta\text{Vul}_{v_i,h_j} \tag{6.7}$$

where $g_{vul}(parent(s))$ denotes the aggregate vulnerability score associated with the allocation path leading to the parent state of s and $\text{Vul}_{h_j}+\Delta\text{Vul}_{v_i,h_j}$ denotes the updated vulnerability score of host h_j after the allocation of task t_i. If s is the root state, then $g_{vul}(s)$ is initialized to 0.

In the simplest case, a *uniform cost search* can be assumed and the lower bound estimate can be defined as $h_{vul}(s)$=0. In this case, the algorithm may expand and visit a higher number states before reaching an optimal goal state. On the other hand, a heuristic that estimates $h_{vul}(s^*)$ can be adopted to improve the performance of the search process. One approach of estimating $h_{vul}(s^*)$ when the traversal algorithm is at state s^* of the state-space tree is as follows.

1. In the current state s^*, use an *operation* of the A* algorithm to obtain the set S of feasible child states.
2. Since the cost function is the aggregate vulnerability score associated with complete task allocation, compute $\text{Vul}_{h_j}+\Delta\text{Vul}_{v_i,h_j}$ for each state in S (lines 6–9).
3. To ensure that $h_{vul}(s)$ is a lower bound, select the state with the smallest value of $\text{Vul}_{h_j}+\Delta\text{Vul}_{v_i,h_j}$ and temporarily mark it as the current state (lines 10–13).
4. Repeat steps 1, 2 and 3 until a goal state is reached, and return h_{vul} as the aggregate vulnerability score along the path leading to this goal state (line 15).

Note that h_{vul}, computed using an admissible heuristic, improves the performance of the state-space search without influencing the final results of the search. We demonstrate the effectiveness of the above heuristic through Example 6.3.1 as well as through the experimental evaluation in Section 6.3.3.

```
1: A* STATE SPACE EXPLORATION SCHEME(v, h)
2: g_vul(root) ← 0
3: OPEN ← OPEN ∪ {(root, f_vul(root))}
4: while OPEN ≠ ∅
5:   s ← arg min_{x∈OPEN} f_vul(x)
6:   OPEN ← OPEN \ {s}
7:   if s = goal state then
8:     CLOSE ← CLOSE ∪ {(s, f_vul(s))}
9:     constructSolution(s)
10:  else
11:    S ← getSuccessors(s)
12:    for all s* ∈ S
13:      new_g_vul ← g_vul(s) + (Vul_{h_j} + ΔVul_{v_i,h_j})
14:      if (s* ∈ OPEN ∪ CLOSE) then
15:        if g_vul(s*) ≤ new_g_vul then
16:          continue
17:        end if
18:      end if
19:      g_vul(s*) ← new_g_vul
20:      f_vul(s*) ← g_vul(s*) + h_vul(s*)
21:      OPEN ← OPEN ∪ {(s*, f_vul(s*))}
22:    end for
23:      CLOSE ← CLOSE ∪ {(s, f_vul(s))}
24:  end if
25: end while
```

Figure 6.2: A* state space exploration algorithm

6.3.2 State-space Exploration Scheme

A* uses the OPEN and CLOSE lists to manage the state-space exploration process. In particular, it uses OPEN to store the set of states that it has visited but not yet expanded (i.e., the states it plans to expand) and CLOSE to store the set of states that are already expanded. Each entry in OPEN and CLOSE consists of a state s and its $f_{vul}(s)$ value. OPEN is initialized with $(root, f_{vul}(root))$ and CLOSE is made empty. The algorithm proceeds by expanding the state with minimum $f_{vul}(s)$ value from OPEN so as to focus the search towards the optimal solution path. When a state s is expanded, it is removed from OPEN, and the successors of state s are generated using the *getSuccessors* function. The $f_{vul}(s^*)$ value of each successor s^* is computed and state $(s^*, f_{vul}(s^*))$ is added to OPEN; at this point, state s is considered "expanded" and added to CLOSE. The algorithm continues to expand states iteratively in this manner until the goal state is reached or OPEN is empty. When the goal state is reached, the *constructSolution*(s) function is used to extract the solution path by tracing the states in the reverse order from the goal to the root state. We note that the state-space tree is dynamically generated based on the states that are visited and expanded. The A* scheme is outlined in Algorithm 6.2 for easy comparison with our elastic search scheme (Algorithm 6.8).

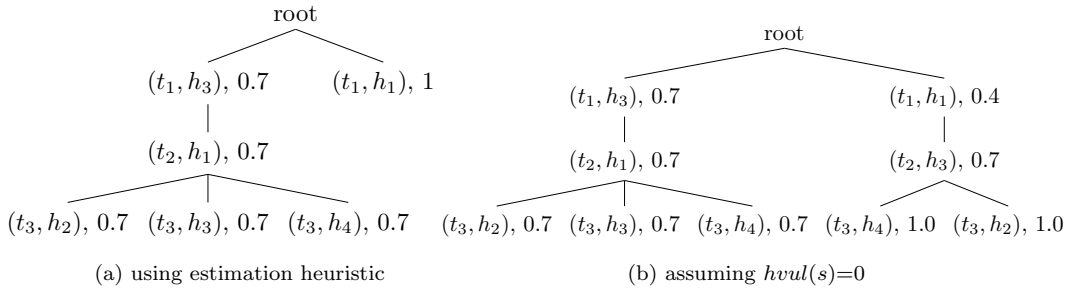
Example 6.3.1. *Consider an infrastructure with four hosts* $\mathcal{H}=\{h_1, \ldots, h_4\}$ *and a mission*

Table 6.1: Example scenario for mission deployment

Infrastructure		Mission	
Host	**Residual CPU capacity, Vulnerability level**	**Task**	**CPU Requirement, Vulnerability tolerance**
$h_j \in \mathcal{H}$	$\overrightarrow{h}$(CPU, Vul)	$t_i \in T$	$\overrightarrow{t}$(CPU, tol)
h_1	0.5, 0.2	t_1	0.4, 0.2
h_2	0.3, 0.2	t_2	0.4, 0.2
h_3	0.7, 0.1	t_3	0.3, 0.4
h_4	0.5, 0.3		

Table 6.2: Vulnerability differential values

$\Delta \text{Vul}_{t_i,h_j}$	h_1	h_2	h_3	h_4
t_1	0.2	0.1	0.1	0.3
t_2	0	0.1	0.2	0.1
t_3	0.1	0.1	0.2	0



(a) using estimation heuristic (b) assuming $hvul(s)$=0

Figure 6.3: State-space tree expanded using A* traversal algorithm

with two tasks $M=\{\tau_1, \tau_2\}$*, where* $\mathcal{R}_1=\{\tau_1^1, \tau_1^2\}$ *and* $\mathcal{R}_2=\{\tau_2^1\}$*. The tasks to be allocated* $T=\{t_1, t_2, t_3\}$ *are mapped to virtual machines* v_1*,* v_2 *and* v_3*. Mission deployment is driven by* p*:*$\{v_1, v_2, v_3\} \rightarrow \{h_1, h_2, h_3, h_4\}$*, and distribute constraint holds for virtual machines* v_1 *and* v_2*. For simplicity, consider only a single resource dimension for hosts and tasks (say CPU). Table 6.1 outlines available CPU capacity and vulnerability level of each host, and CPU requirements and vulnerability tolerance threshold of each task. Table 6.2 provides details on the increase in vulnerability values for each allocation.*

Figure 6.3(a) illustrates the state-space tree generated during mission deployment by the algorithm discussed in the previous section. The algorithm starts from the root state *by generating the states for the first level in the tree. The* operation *considers task* t_1*, discards the hosts* h_2 *and* h_4 *since they violate the capacity and vulnerability threshold constraints respectively, and generates states* $s(v_1, h_3)$ *and* $s(v_1, h_1)$*. The* $f_{vul}(s)$ *values for the two states are calculated as 0.7 and 1.0 respectively and pushed in* `OPEN`*.*

Since state $s(v_1, h_3)$ *has minimum* $f_{vul}(s)$ *value, it is extracted from* `OPEN` *and marked as the current state. Its successors are then generated and* $f_{vul}(s)$ *values calculated; in this case only state* $s(v_2, h_1)$ *with* $f_{vul}(s) = 0.7$ *is returned and pushed in* `OPEN`*. In particular,*

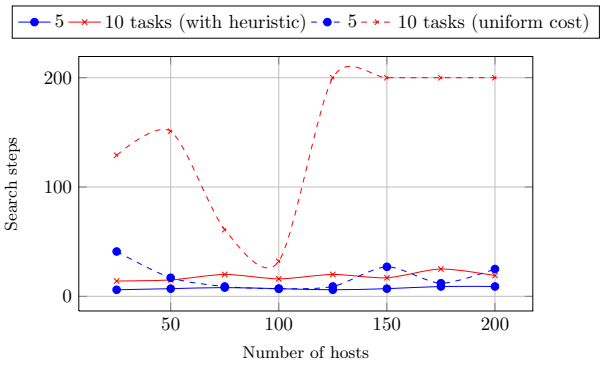


Figure 6.4: Number of search steps with varying number of hosts

after calculating $g_{vul}(s) = 0.4$*, the procedure described in Section 6.3.1 is used to estimate the vulnerability value* $h_{vul}(s)$ *along this path. In this case, feasible states corresponding to virtual machine* v_3 *(task* t_3*) are considered, and the state with minimum* $g_{vul}(s)$ *value (0.3) is returned since states corresponding to task* t_3 *are leaf nodes.*

At this point, state $s(v_2, h_1)$ *is the entry with the lowest* $f_{vul}(s)$ *value in* `OPEN`*. This state is marked as the current state and its successors* $s(v_3, h_4)$*,* $s(v_3, h_3)$ *and* $s(v_3, h_2)$ *are generated. The* $f_{vul}(s)$ *value of all these states are calculated and pushed in* `OPEN`*. The state* $s(v_2, h_1)$ *is now pushed in* `CLOSE`*. The states corresponding to the task* t_3 *are similarly expanded and visited. The state-space search has now reached the goal state and found the complete task allocation. The algorithm pushes* $s(v_3, h_4)$ *in* `CLOSE`*, and returns* $p(v_1)$=h_3*,* $p(v_2)$=h_1 *and* $p(v_3)$=h_4 *as the complete allocation solution.*

In case the heuristic to estimate the vulnerability values at each state is not considered i.e., assuming uniform cost $h_{vul}(s)$=0*, 9 states are expanded to perform task allocation as shown in Figure 6.3(b), while the algorithm with the heuristic expands only 6 states.*

6.3.3 Experimental Evaluation

This section reports the experiments conducted to validate the A* state-space search algorithm. Here, the objective is to evaluate the performance of in terms of processing time, number of steps required to identify a complete solution, and approximation ratio for different network configurations and mission scenarios. In order to consider different network configurations, the vulnerability score and the available capacity of each host $h_j \in \mathcal{H}$ is randomly initialized. The vulnerability tolerance and the resource requirements have been similarly initialized for each virtual machine $v_i \in V$. All the results reported here are averaged over multiple executions.

First, as expected, the number of search steps required for complete task allocation using the heuristic discussed in the previous section is smaller than in the case of uniform cost search, particularly for larger missions. Figure 6.4 shows how search complexity

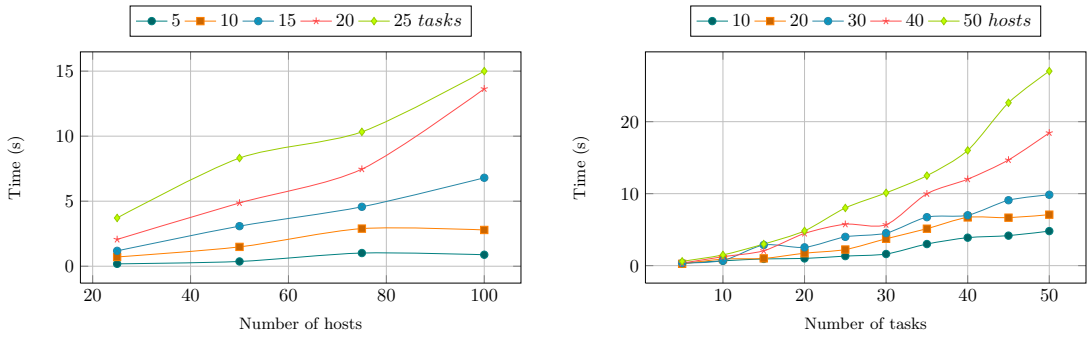


Figure 6.5: Processing time to compute allocation: (left) with varying number of hosts for different sizes of missions and (right) with varying number of tasks for different sizes of the infrastructure

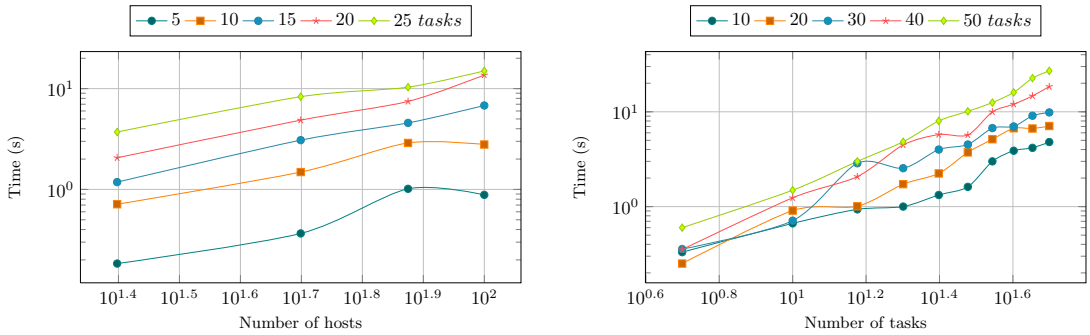
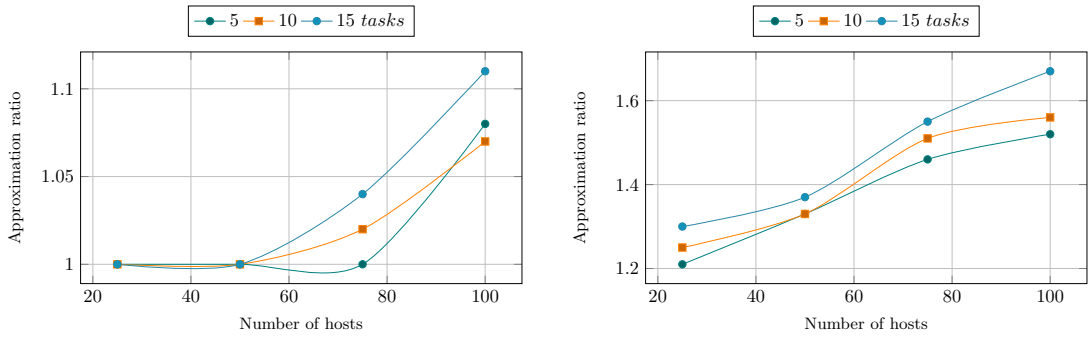


Figure 6.6: Processing time in logarithmic scale: (left) with varying number of hosts for different sizes of missions and (right) with varying number of tasks for different sizes of the infrastructure

increases when the number of hosts increases, for $|T| = 5$ and $|T| = 10$. Note that the number of search steps for uniform cost search is comparable to the heuristics-driven search when the number of tasks is 5. However, search complexity increases exponentially as the size of the mission increases.

The scalability of the algorithm is studied, in terms of processing time, for different network and mission sizes. Figure 6.5 (left) shows how the processing time increases as the number of hosts increases. It is clear that processing time remains in the order of a few milliseconds for small missions ($|T| = 5$ or $|T| = 10$), irrespective of the number of hosts. However, as the size of the mission increases, processing time increases steeply (up to 15 seconds for mission with 25 tasks on a network with 100 hosts). Figure 6.5 (right) shows how processing time increases as the number of tasks increases. Processing times for missions with 10–30 tasks are roughly similar independently of the size of the network,



(a) Mission's vulnerability exposure

(b) Global optimal in the network

Figure 6.7: Optimality of allocation, i.e., the approximation ratio with varying number of hosts for different sizes of missions

and increases for missions with 40–50 tasks (particularly, on networks with over 30 hosts). Figure 6.6 shows the relationship between processing time, mission size and network size on a logarithmic scale. These charts show that the processing time of the A* algorithm with a heuristic increases linearly with the size of both missions and networks. Relatively small missions can be allocated on large-scale networks within a few seconds.

Finally, the optimality of the solution is evaluated in two perspectives: *i)* the mission's exposure to network vulnerabilities and *ii)* the quality of deployment in the whole network. Figure 6.7(a) shows how the approximation ratio (i.e., the ratio between the aggregated vulnerability score of the mission deployed using the A* algorithm and that computed using an exhaustive search) changes for different configurations of the network and different mission sizes. It is clear that the approach discussed in this section deploys missions with acceptable security, in a time efficient manner. The approximation ratio remains well within 1.2 in all instances. Similarly, Figure 6.7(b) shows how the approximation ratio varies with respect to security across the whole network. In this respect, the variance is low for the missions of all sizes when the network consists of less than 50 hosts.

6.4 Mission Deployment using an Elastic Algorithm

Resource management algorithms in Cloud computing are studied in the literature for a wide range of objectives. However, existing solutions are based on heuristics that provide "all-or-nothing" results; that is, the algorithm must run for at least a given period of time to provide a feasible solution. Unless intelligently designed, such algorithms may take very long period of time to generate highly sub-optimal results. It is often observed in many real-world scenarios, particularly involving mission-critical applications, that the processing time available for resource management algorithms is very limited and often varying, raising the need to build algorithms that take a different approach.

This section presents a scheme that transforms A* algorithm to an elastic algorithm. The transformation scheme is based on the notion of weighted heuristics and built on the principles of Anytime algorithms [53]. This scheme can generally transform any best-fit search algorithm to an elastic algorithm; however, here it is designed specifically for secure mission deployment using A*. Section 6.4.1 outlines the behavior of weighted heuristics. Section 6.4.2 presents the transformation scheme, and Section 6.4.3 provides some simulation results.

6.4.1 A* with Weighted Heuristics

The behavior of A* largely depends on the heuristic $h_{vul}(s)$. The first goal state expanded during state-space exploration is guaranteed to be optimal if the heuristic $h_{vul}(s)$ is *admissible* [96, 113]. That is, $h_{vul}(s)$ is always the lower bound of the true vulnerability score of reaching the goal state from s. On the other hand, a heuristic is *consistent* if $h_{vul}(s) < c_{vul}(s, s^*) + h_{vul}(s^*)$ for any successor s^* of s if $s \neq$*goal state* and $h_{vul}(s)=0$ if $s=$*goal state*. Here, $c_{vul}(s, s^*)=\text{Vul}_{h_j}+\Delta\text{Vul}_{v_i,h_j}$ is the vulnerability score due to allocation of task in virtual machine v_i on host h_j in state s^* (i.e., the cost of edge from s to s^*). We note that consistency implies admissibility, and non-admissibility implies inconsistency. We also note that a consistent $h_{vul}(s)$ guarantees that $g_{vul}(s)$ for a state s is optimal when chosen for expansion and a state s is never expanded more than once.

Starting from [102], many researchers have explored the effect of weighting $g(s)$ and $h(s)$ in the A* state-space exploration process to find a bounded optimal solution with less computational effort. In this approach, the cost function is defined as $f'(s) = g(s)+\epsilon\times h(s)$, where ϵ is a user-defined parameter, and state-space exploration is performed using $f'(s)$. We use the notation $f'(s)$ to distinguish between the weighted cost function from the conventional one $f(s)$.

The elastic algorithm in Section 6.4.2 assumes existence of an admissible heuristic $h_{vul}(s)$, and uses it to create a weighted heuristic $h'_{vul}(s) = \epsilon \times h_{vul}(s)$, where $\epsilon \geq 1$ is initialized by the user and dynamically changed by the algorithm at runtime. The weighted heuristic accelerates the search process by making the states closer to the goal more attractive, providing a depth-first search aspect, and implicitly adjusts the trade-off between search effort and solution quality. On one hand, setting $\epsilon = 1$ results in standard A* and the solution is guaranteed to be optimal. On the other hand, inflating the heuristic may violate the admissibility property and, as a consequence, it is possible for a state to have a higher-than-optimal $g_{vul}(s)$ value when expanded (i.e., the $g_{vul}(s)$ value for some states may decrease during the A* search).

A state is considered to be *inconsistent* if its $g_{vul}(s)$ value decreases during the iterative search. An inconsistent state is moved from `CLOSE` to `OPEN`, and eventually re-expanded, so that the correct (reduced) $g_{vul}(s)$ value is again associated with it. Inconsistency in a state also introduces inconsistency among its successors. For example, if the vulnerability score of the allocation performed at level 2 of the tree decreases ($g_{vul}(s_2)$), overall vulnerability

score at level 3 also decreases (since $g_{vul}(s_3) = g_{vul}(s_2) + c(s_2, s_3)$). Hence, the improved $g_{vul}(s)$ value of a state must be propagated to its successors by re-expanding the state. This series of state re-expansions must be carried out either until the successors no longer depend on state s or the inconsistency is corrected for all the successors. This iterative process results in the same state being expanded multiple times (as opposed to conventional A* where a state is never expanded more than once). Finally, we note that the `OPEN` list consists of all the inconsistent states, representing the states with which state expansion must proceed in order to correct the propagation of inconsistency.

6.4.2 Elastic Task Allocation Algorithm

This section presents the scheme that transforms the weighted A* into an elastic algorithm. The elastic mission deployment algorithm should possess the following desired properties.

- The optimality of the solution generated by the algorithm must improve as the execution time made available to it increases.
- The algorithm must return a solution whenever it is interrupted, and optimality bound of the returned solution must be measurable.
- Performance w.r.t. A* must improve so that convergence to the optimal solution can be achieved quickly.

Figure 6.8 illustrates the elastic algorithm that satisfies the three aforementioned desired properties. The existence of an admissible heuristic $h_{vul}(s)$ is assumed and the following definition of the weighted cost function is provided:

$$f'_{vul}(s) = g_{vul}(s) + \epsilon \times h_{vul}(s) \tag{6.8}$$

The algorithm repeatedly executes A*, starting from (a large) $\epsilon = \epsilon_0$ value greater than 1, decreasing ϵ by a small fixed amount for each iteration, until $\epsilon = 1$ (see *main* function). In the first iteration, when $\epsilon = \epsilon_0$ is set to a value greater than 1, the *ImproveAllocation* function computes a solution path by expanding only a few states when compared to the conventional A*. This is possible since the first goal state can be reached by expanding only the states whose vulnerability cost is ϵ_0 times the optimal solution (discussed below). This implies that the first solution, though sub-optimal, can be computed within a time fractional to running A*. The algorithm continues to the subsequent execution of A* by decreasing the ϵ value by a small quantity in order to build a new solution with improved optimality. In this case, the *ImproveAllocation* function reuses the previous search results (expanded states and heuristic values) and expands some additional states to compute the new solution path whose optimality is within the factor of new ϵ. In general, as the value of ϵ decreases, additional states are expanded, and the optimality of the solutions improve according to ϵ. This process continues either until the algorithm is interrupted

```
1: ELASTIC TASK ALLOCATION(v, h)
2: ImproveAllocation()
3: while OPEN ≠ ∅ and not interrupted do
4:   s ← arg min_{x∈OPEN} f'_vul(x)
5:   OPEN ← OPEN \ {s}
6:   if incumbent=nil or f'_vul(s)<f'_vul(incumbent)
7:     CLOSE ← CLOSE ∪ {s}
8:     S ← getSuccessors(s)
9:     for all s*∈S such that g_vul(s) + c_vul(s, s*) + h_vul(s*) < f'_vul(incumbent)
10:      if s* = goal state then
11:        incumbent ← s*
12:        g_vul(s*) ← g_vul(s) + c(s, s*)
13:        f'_vul(incumbent) ← g_vul(s*)
14:        return constructSolution(incumbent)
15:      else
16:        if s* ∈ OPEN ∪ CLOSE and g_vul(s*) > g_vul(s) + c(s, s*) then
17:          g_vul(s*) ← g_vul(s) + c(s, s*)
18:          f'_vul(s*) ← g_vul(s*) + ε × h_vul(s*)
19:          if s* ∉ CLOSE then
20:            OPEN ← OPEN ∪ {s*} with new f'_vul(s*)
21:          else
22:            INCONS←INCONS∪{s*} with new f'_vul(s*)
23:            CLOSE ← CLOSE \ {s*}
24:          end if
25:        else
26:          g_vul(s*) ← g_vul(s) + c(s, s*)
27:          f'_vul(s*) ← g_vul(s*) + ε × h_vul(s*)
28:          OPEN ← OPEN ∪ {s*}
29:        end if
30:      end if
31:    end for
32:  else
33:    INCONS ← INCONS ∪ {s}
34:  end if
35: end while
36:
37: main()
38: g_vul(root) ← 0, ε ← ε_0
39: OPEN ← OPEN ∪ {(root, f_vul(root))}
40: CLOSE ← INCONS ← ∅
41: ImproveAllocation() /*publish the current solution*/
42: while ε > 1 do
43:  Decrease ε
44:  Move states from INCONS to OPEN
45:  Update f_vul(s) values for all s ∈ OPEN
46:  CLOSE ← ∅
47:  ImproveAllocation() /*publish the current solution*/
48: end while
```

Figure 6.8: Elastic task allocation algorithm

or it converges to an optimal solution. We note that the algorithm is guaranteed to return an optimal solution when $\epsilon = 1$ since it becomes consistent. We also note that the *ImproveAllocation* function primarily behaves as an incremental A* and uses `OPEN` and `CLOSE` lists to store the visited and expanded states respectively.

The sub-optimality bound of the solution computed during each iteration must be measurable. This feature is critical for mission deployment since it allows the user to assess the vulnerability risk of the mission for a given deployment and accordingly decide the amount of time for which the algorithm must be executed. We remind that, if $h_{vul}(s)$ is consistent, the optimality of the solution is within the factor ϵ of the optimal. That is, the overall vulnerability score of the mission is no larger than ϵ times the vulnerability score of the optimal solution. As discussed in Section 6.4.1, when $\epsilon > 1$, some states may become inconsistent and may be expanded multiple times in a given iteration. In this context, it possible to bound the sub-optimality of a solution by the factor of ϵ by restricting each state to re-expand no more than once in an iteration [81]. The elastic algorithm achieves this by modifying A*, and storing all the inconsistent states that have been previously expanded ($s \in$ `CLOSE`) into the `INCONS` list (lines 20-22). By doing so, the algorithm avoids visiting previously expanded inconsistent states (since space exploration is performed based on the states in `OPEN`) and the union of `OPEN` and `INCONS` lists now gives the set of all inconsistent states. Before each iteration, all the states from `INCONS` are again moved to `OPEN`, ensuring convergence to the optimal (line 43).

The above technique of blocking state-space expansion from any state stored in the `INCONS` list also allows to expand more distinct states during each iteration. This step significantly improves the overall performance of the elastic algorithm as the number of close-to-optimal solutions for the mission deployment problem is typically very large. In addition to limiting state re-expansions, the elastic algorithm also uses bounds to prune the search space and improve the performance. This step is based on the following observation. Using conventional A*, the cost function $f_{vul}(s)$ with an admissible heuristic gives a lower bound on the vulnerability score of an allocation solution. If x denote the goal state corresponding to the best allocation found so far, $f_{vul}(x)$ represents an upper bound on the vulnerability score of an optimal solution. Hence, any state with $f_{vul}(s)$ value (lower bound) greater than or equal to the current $f_{vul}(x)$ (upper bound) need not be expanded since it cannot lead to an improved solution [53]. To implement this step, the algorithm do not insert the newly generated state whose $f_{vul}(s)$ value is greater than or equal to the current upper bound in the `OPEN` list, and block its expansion, by adding it to `INCONS` (lines 8 and 32). Even after pruning, the best solution available is optimal when there are no unexpanded states in `OPEN` (i.e., it is empty) with $f_{vul}(s)$ value less than $f_{vul}(x)$.

6.4.3 Experimental Evaluation

The simulation results of the experiments conducted to evaluate the elastic algorithm are reported in this section. In particular, the analysis of the algorithm in terms of *i)* the

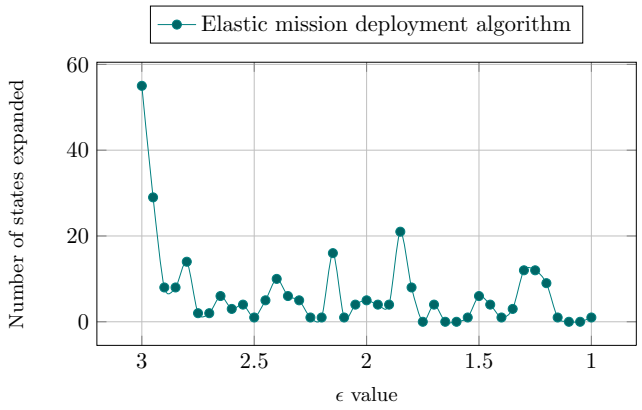


Figure 6.9: Number of states expanded during each iteration

number of search steps required to find an allocation solution, and *ii)* the cost of using an elastic algorithm in terms of execution time is provided.

Setup. The Cloud infrastructure is configured by randomly initializing hosts with different amounts of residual resources and vulnerability scores. Similarly, missions with different number of tasks are defined, replicated task sets randomly chosen, and resource requirement and vulnerability tolerance values are initialized. The ϵ value is initialized to $\epsilon_0 = 3$, and decreased in the steps of 0.05 until $\epsilon = 1.0$. The heuristic value $h_{vul}(s)$ is calculated using exhaustive search to ensure admissibility and consistency. The simulation results presented here are the mean values of 10 executions of each configuration. The program realizing the elastic algorithm shown in Figure 6.8 is written in C++ and executed on a machine having Intel i7-2860QM 2.50GHz processor, with 16GB of memory, running Windows 7 operating system.

Number of search steps. Figure 6.9 shows the number states that the elastic algorithm expands during each iteration to return an allocation solution. The algorithm computes the first solution by expanding only 55 states (when $\epsilon = 3$). In the next iteration, it expands inconsistent states and iteratively builds the state space tree, to return a solution whose optimality is at most 2.95 times the optimal solution, by expanding 29 states altogether. This process continues until $\epsilon = 1$, where the algorithm returns the optimal solution. The total number of state expansions is 277.

The weighted A* is executed iteratively in order to compare the difference in the number of state expansions. When $\epsilon = 3$, A* returns a solution by expanding 103 states. As we can see, the number of search steps increase as the ϵ value decreases (note that A* is run from scratch during each iteration) and returns an optimal solution ($\epsilon = 1$) by expanding 248 states. This implies that the expense of transforming A* to our elastic

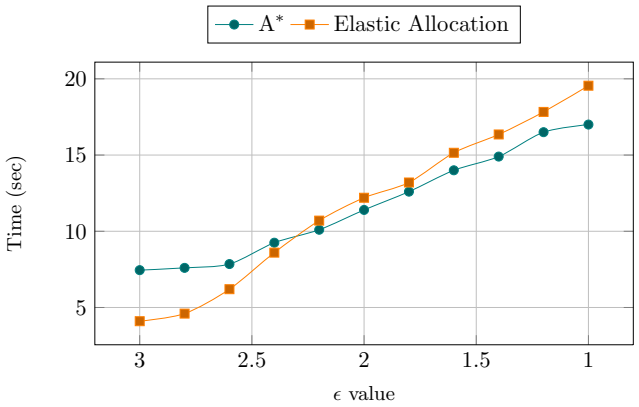


Figure 6.10: Total execution time varying ϵ value

algorithm is $277/248 \equiv 1.11$ (12% overhead).

Processing time evaluation. Figure 6.10 compares the total execution time between iterative A* and the elastic algorithm. These results do not include the time for calculating the heuristic $h_{vul}(s)$ since it is determined using an exhaustive search. The elastic algorithm computes the first solution (4.1 sec) much quicker than A* weighted with $\epsilon = 3$ (7.45 sec). The total amount of time taken by the elastic algorithm to return the optimal solution is nearly 20 seconds while A* (with $\epsilon = 1$) takes about 17 seconds to return an optimal solution running from scratch.

6.4.4 Adaptive Mission Deployment using an Elastic Algorithm

Cloud environment is highly dynamic with respect to host configuration, vulnerability distribution, and resource availability. For example, the vulnerability score of a host may improve at runtime if the user includes new firewall rules to prevent suspicious network connections. Similarly, a host failure may diminish dependability, and discovery of new attacks may increase the vulnerability score of network elements. This implies that the vulnerability score $c(s, s^*)$ of one or more mission tasks may change at runtime and allocation performed using the algorithm illustrated in Figure 6.8 (or Figure 6.2) may not remain ϵ sub-optimal. Consequently, an approach that adapts the current allocation based on the new vulnerability scores (system configuration) is necessary to ensure the secure execution plan for a mission.

This section first describes how to modify A* to adaptively compute a solution path, and then discusses how to introduce anytime behavior on it.

Assume that the cost changes are identified by the monitoring system, and an algorithm that computes a new solution path whenever the changes affect the security of the mission

is needed. Similarly to [75, 80], the algorithm must be adaptive in the way that it provides new allocation only for a subset of tasks (immediately affected by system changes) that, when realized, ensures ϵ sub-optimality of the mission deployment. An approach to build such algorithm is to recalculate those $g_{vul}(s)$ values that have changed (or not calculated before), and similarly to algorithms in Figures 6.2 and 6.8, use the heuristic $h_{vul}(s)$ to focus the search by expanding only those states that provide an optimal solution path. In addition to $g_{vul}(s)$ and $h_{vul}(s)$, the algorithm should store a $look\_ahead\_h_{vul}(s)$ value that estimates the vulnerability score one-step ahead. This implies that $look\_ahead\_h_{vul}(s)$ is better informed than traditional heuristic. $look\_ahead\_h_{vul}(s) = 0$ if s is the *goal state*, otherwise, it is $\min_{s^* \in Succ(s)}(c_{vul}(s, s^*) + h_{vul}(s^*))$. A state s is consistent if $h_{vul}(s) = look\_ahead\_h_{vul}(s)$; otherwise, it is either over-consistent $h_{vul}(s) > look\_ahead\_h_{vul}(s)$ or under-consistent $h_{vul}(s) < look\_ahead\_h_{vul}(s)$.

The cost function (originally $f_{vul}(s)$) of each state is represented using the vector $k_{vul}(s) = [k^1_{vul}(s), k^2_{vul}(s)]$, where

$$k^1_{vul}(s) = \min(h_{vul}(s), look\_ahead\_h_{vul}(s)) + g_{vul}(root, s)$$

$$k^2_{vul}(s) = \min(h_{vul}(s), look\_ahead\_h_{vul}(s))$$

and a ranking among the states is defined based on $k_{vul}(s)$ values. In particular, given two states s, s' with cost functions $k_{vul}(s)$ and $k_{vul}(s')$, $k_{vul}(s) \preceq k_{vul}(s')$ if $k^1_{vul}(s) < k^1_{vul}(s')$ or $\{$when $k^1_{vul}(s) = k^1_{vul}(s')$, $k^2_{vul}(s) < k^2_{vul}(s')\}$. The extended cost function is equivalent to $k_{vul}(s) = [f_{vul}(s), h_{vul}(s)]$ in terms of conventional A*. Intuitively, the component $k^1_{vul}(s)$ ensures that only new over-consistent states that can potentially decrease the cost of the goal state are expanded, and only new under-consistent states that can invalidate the cost of current solution path are processed. The component $k^2_{vul}(s)$ restricts the search towards the states that are potentially relevant in adapting the current solution path. We note that the notion of under-consistent states was not considered by the algorithm in Figure 6.8. In such cases, states must be inserted into `OPEN` with the minimum of the old and updated cost, computed without weighted heuristic ($\epsilon = 1$), to ensure the increase in vulnerability score is correctly propagated to successors.

Similarly to the algorithm in Figure 6.8, the adaptive elastic algorithm performs a series of searches with decreasing ϵ, and computes an ϵ sub-optimal solution during a given iteration. When vulnerability score of an allocation changes, the algorithm updates the cost function values $g_{vul}(s)$, $h_{vul}(s)$, $look\_ahead\_h_{vul}(s)$ and inserts relevant states in `OPEN`. Inconsistent states that are already expanded are stored in `INCONS`, and at the start of each iteration, states are moved from `INCONS` to `OPEN`, and `CLOSE` is made empty. The state-space exploration is then performed in the reverse-order (starting from goal state and moves towards the root) based on increasing state ranking, until no state in `OPEN` has vulnerability cost less than that of the current goal state. A large number of states need to be expanded to compute ϵ sub-optimal solution when the system changes are significant. In such cases, the algorithm increases the ϵ value and computes a less optimal solution

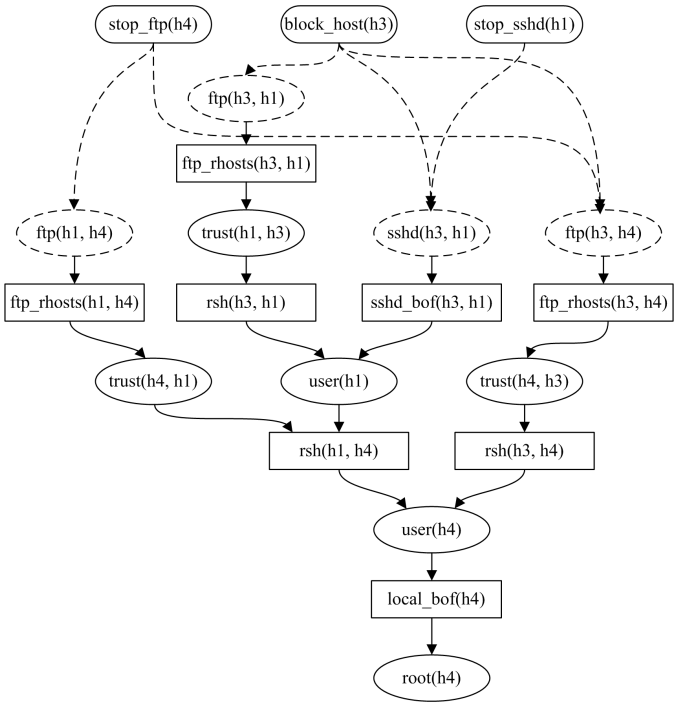


Figure 6.11: Example of an attack graph including possible hardening actions, initial conditions, intermediate conditions, and exploits

quickly by expanding fewer number of states. The VMs in which affected mission tasks are deployed are migrated on to new physical hosts according to the new solution. This ensures new deployment configuration of the mission.

6.5 Mission Protection

The first step to securely execute a given mission is to deploy it in the infrastructure by minimizing its exposure to vulnerabilities. This can be achieved using the algorithms discussed in Sections 6.3 and 6.4. The second step is to harden the hosts and network links used by the mission in order to further improve the mission's resilience against possible cyber-attacks. This section formulates the hardening problem and the cost model, and discusses an approach to solve the problem using attack graphs. The solution discussed here is presented in [2].

Problem formulation. A network hardening strategy is a set of atomic actions that can be taken to guard various resources in the network. For instance, an action may consist in stopping the *ftp* service on a given host. Let us start by introducing the notion

of *attack graphs* that represent prior knowledge about vulnerabilities, their dependencies, and network connectivity. Given a set E of exploits, a set of security conditions C (e.g., existence of a vulnerability on a host or connectivity between two hosts), a require relation $R_r \subseteq C \times E$, and an imply relation $R_r \subseteq E \times C$, an attack graph is a directed graph $G = (E \cup C, R_r \cup R_i)$, where $E \cup C$ is the vertex set and $R_r \cup R_i$ is the edge set [2]. The term initial conditions refers to the subset of conditions $C_i = \{c \in C | \nexists e \in E \text{ s.t. } (e, c) \in R_i\}$, whereas other conditions, which are usually consequences of exploits, are referred to as intermediate conditions.

Example 6.5.1. *In Example 6.3.1, mission tasks are allocated on hosts h_3, h_1 and h_4. Assume that our objective is to prevent the attacker from gaining root privileges on host h_4 i.e., we want to avoid reaching condition $root(h_4)$ so as to protect task t_3.*

Figure 6.11 illustrates an example attack graph in which exploits are represented using rectangles and conditions using ovals. The dashed ovals are the initial conditions and other ovals represent intermediate conditions. The attack graph is simplified in several ways. For example, a single condition $ftp(h_s, h_d)$ is used to denote transport-layer ftp connectivity between two hosts h_s and h_d, physical-layer connectivity, and existence of the ftp daemon on host h_d. The attack graph depicts a simple scenario, with hosts h_3, h_1 and h_4, and four types of vulnerabilities: ftp_rhosts, rsh, $sshd\_bof$, and $local\_bof$. An example of an attack path is: the attacker can establish a trust relationship with host h_4 (using condition $trust(h_4, h_3)$) from its host h_3 via exploit $ftp\_rhosts(h_3, h_4)$ on host h_4. It can then gain user privileges on host h_4 (using condition $user(h_4)$) with an rsh login, and achieve the goal condition $root(h_4)$ using a local buffer overflow attack.

An allowable *hardening action* is any subset of initial conditions such that all the conditions can be jointly disabled in a single step, and no other initial conditions needs to be disabled. The rounded rectangles in the attack graph in Figure 6.11 are examples of allowable hardening actions:

- $stop\_ftp(h_4) = \{ftp(h_3, h_4)\}$
- $block\_host(h_3) = \{ftp(h_3, h_1), sshd(h_3, h_1), ftp(h_3, h_4)\}$
- $stop\_sshd(h_1) = \{sshd(h_3, h_1)\}$

Given an attack graph, a set A of allowable actions and a set of target conditions $C_t = \{c_1, \ldots, c_n\}$, a *hardening strategy* S is a set of hardening actions such that conditions in C_t cannot be reached after all the actions are applied.

We note that removing specific initial conditions may require to take actions that disable additional conditions (e.g., conditions that are not part of any attack path). Hence, a *hardening strategy* must be obtained in terms of allowable actions, and a cost model that takes into account the impact of hardening actions. A *hardening cost function* is any function $cost : \mathcal{S} \to \mathbb{R}^+$ that satisfies the following conditions:

$$cost(\emptyset) = 0 \tag{6.9}$$

$$(\forall S_1, S_2 \in \mathcal{S})(C(S_1) \subseteq C(S_2) \implies cost(S_1) \leq cost(S_2)) \tag{6.10}$$

$$(\forall S_1, S_2 \in \mathcal{S})(cost(S_1 \cup S_2) \leq cost(S_1) + cost(S_2)) \tag{6.11}$$

where $\mathcal{S}$ denotes the set of all possible strategies and $C(S)$ denotes the set of all conditions disabled under strategy S. We note that many different cost functions can be defined. For example, a simple cost function can be a count on the number of initial conditions that are removed under a hardening strategy. If we assume that the cost of $cost(\{stop\_ftp(h_4)\}) = 20$, $cost(\{block\_host(h_3)\}) = 10$ and $cost(\{stop\_sshd(h_1)\}) = 15$, then the optimal strategy with respect to $root(h_4)$ is $block\_host(h_3)$.

Mission protection solution. Most hardening techniques starts from the target conditions and move backwards in the attack graph to make logical inferences. Such backward search schemes typically face combinatorial explosion issues. Therefore, the definition of a scalable scheme to build hardening strategies is necessary.

Starting from initial conditions, the hardening scheme in [2] traverses the attack graph forward. A key advantage of traversing the attack graph forward is that in a single pass, the algorithm can compute hardening strategies with respect to any condition. Given a set C_t of target conditions, add a dummy exploit e_i for each condition $c_i \in C_t$. The exploit e_i then has c_i as its only precondition. Then, add a dummy target condition c_t, such that all the dummy exploits e_i having c_t are their only postcondition. Since c_t is reachable from any dummy exploit e_i, we need to prevent all such exploits. This can be achieved by disabling the corresponding preconditions; that is, harden the infrastructure with respect to all target conditions in C_t. We note that, given a target condition c_t, the attack graph is a tree rooted at c_t, having initial conditions as its leaf nodes.

The hardening algorithm first performs a topological sort of the nodes in the attack graph, and pushes them into a queue, with initial conditions at the front of the queue. An element q is popped from the queue until the queue is empty. This gives rise to three conditions on q:

- If q is an *initial condition*, it is associated with a set of strategies $\sigma(q)$ such that each strategy contains one of the allowable actions that disables q.
- If q is an *exploit*, it is associated with a set of strategies $\sigma(q)$ that is the union of the sets of strategies for each condition c required by q. In this context, an exploit can be prevented by disabling any of the required conditions.
- If q is an *intermediate condition*, it is associated with a set of strategies $\sigma(q)$ such that each strategy is the union of a strategy for each of the exploits that imply q.

In order to disable an intermediate condition, all the exploits that imply it must be prevented. This scheme, under reasonable assumptions, have the approximation ratio that is

bounded $n^{d/2}$, where n is the maximum in-degree of nodes in the graph and d is the depth of the graph.

Example 6.5.2. *Consider again the attack graph of Figure 6.11, and assume that the cost of actions stop_ftp(h_4), block_host(h_3), and stop_sshd(h_1) is 20, 10, and 15 respectively. After executing the topological sort and examining initial conditions, using k=1, we obtain the following intermediate results:*

- $\sigma(ftp(h_1, h_4)) = \{\{stop\_ftp(h_4)\}\}$
- $\sigma(ftp(h_3, h_1)) = \{\{block\_host(h_3)\}\}$
- $\sigma(sshd(h_3, h_1)) = \{\{block\_host(h_3)\}\}$
- $\sigma(ftp(h_3, h_4)) = \{\{block\_host(h_3)\}\}$

When the algorithm examines the exploit $rsh(h_1, h_4)$, we obtain $\sigma(rsh(h_1, h_4)) = \{\{stop\_ftp(h_4)\}, \{block\_host(h_3)\}\}$ before pruning. After pruning, we obtain $\sigma(rsh(h_1, h_4)) = \{\{block\_host(h_3)\}\}$ since it has lower cost. The algorithm finally returns $\sigma(rsh(h_1, h_4)) = \{\{block\_host(h_3)\}\}$ as the hardening strategy, which is the optimal solution, as discussed above.

6.6 Chapter Summary

We presented a simple yet powerful technique to deploy and execute users' applications in the Cloud infrastructure with high levels of dependability. In particular, we formulated the secure mission deployment problem as a task allocation problem whose objective is to minimize the application's exposure to residual vulnerabilities, and presented an A* based algorithm to solve it. We showed experimentally that our algorithm scales linearly with the size of both missions and infrastructures, and provides solutions with with good approximation guarantees. We then presented a scheme that transforms A* to an elastic task allocation algorithm based on the Anytime processing approach. We showed experimentally that the benefit of having an elastic algorithm introduces low overhead with respect to traditional A*. Finally, we discussed our elastic redeployment algorithm that adapts the current allocation of the mission in order to minimize the affect of system changes at runtime.

For a given request, after translating high level user requirements to low level mechanisms, and specifying dependability constraints (Chapter 5), the resource management algorithms of this chapter can be applied to deliver high levels of reliability, availability, and security to users' applications. In addition, elastic algorithms and resource protection solutions make our solution suitable for mission-critical applications. This solution clearly denotes the third, and the highest, level of dependability offered in this thesis. We note that the approach of formulating secure application deployment as a task allocation

problem and elastic nature of resource management algorithms has not been well studied in the literature.

7 Conclusions

In this thesis, we addressed the problem of dependability in Cloud computing, in order to reduce the risks of using SaaS, PaaS and IaaS services for building, deploying, and executing applications. After a brief introduction and a discussion of related work, we focused on three specific modules: *i)* dependability certification of services, *ii)* the notion of providing dependability as a service, and *iii)* secure application deployment and execution. Each module makes different assumptions on the system context, users' requirements and desired dependability features, and together improve dependability progressively at three different levels. A given module can be applied individually or be complemented with one another based on users' specific dependability goals. For example, a user can simply implement the functional aspects of her application and engage with the dependability service provider to increase its reliability, security, and availability. She can then obtain a dependability certificate for her application to build a trustworthy relationship with her customers. In this chapter, we shortly summarize the original contributions of this thesis and we outline some future work.

7.1 Summary of the Contributions

The contributions of this thesis is threefold.

Dependability assurance of services. We present a dependability certification scheme that, starting from the STS-based model of the service, generates a certification model in the form of a discrete-time Markov chain. The certification model is used to validate whether the service supports a given dependability property with a given level of assurance. The result of property validation and certification is a machine-readable certificate that represents the reasons why the service supports a dependability property and serves as a proof to the users that appropriate dependability mechanisms have been used while building it. To complement the dynamic nature of service-based infrastructures, the certificate validity is continuously verified using runtime monitoring, making the certificate usable both at discovery and run time. Our certification scheme

allows users to select services with a set of certified dependability properties, and supports dependability certification of composite services.

Dependability management. We advocate the new dimension wherein users' applications can transparently obtain required dependability properties, from a third party, as an additional service. Our main contributions in this aspect consists in designing a framework that *i)* encapsulates all the components necessary to offer dependability as a service to the users and *ii)* integrates easily within existing Cloud infrastructures. We provide an approach to measure the effectiveness of each dependability module in different configurations, and its use to define a search algorithm that identifies low level mechanisms based on users' high level requirements. Furthermore, we categorize and formalize several constraints that allow enforcement of deployment conditions inherent to the low level mechanisms selected according to the users' requirements. Such constraints are then used to solve the overall problem of resource allocation in the Clouds. Finally, since Cloud computing environment is highly dynamic, we present an approach to adapt the current allocation of the application when system changes affect its desired fault tolerance and performance. This allows our solution to mask system changes, thus ensuring delivery of a solution that maintains users' requirements also during runtime.

Secure application execution. We present a solution wherein, first, the current vulnerability distribution of the Cloud is considered and users' applications are deployed in the most secure manner possible. Then, network hardening techniques are applied to protect the deployed applications from possible cyber-attacks. Our main contribution in this respect consists in modeling the secure application deployment problem as a task allocation problem with an application-centric, security-oriented objective, subject to dependability constraints, and solving it using A* based search algorithm. Furthermore, in contrast to existing resource management algorithms, we present an approach to build interruptible, elastic algorithms whose optimality improves as the processing time increases, converging to an optimal solution is eventually. In particular, we present two elastic algorithms: first, for a given user request, our algorithm computes an execution plan for the user's application so as to minimize the application's exposure to existing vulnerabilities; second, our elastic redeployment algorithm adapts the application's current allocation to minimize the impact of system changes.

7.2 Future Work

The research described in this thesis can be extended along several directions.

7.2.1 Dependability Certification of Services

Error tolerant modeling. The monitoring data may be error-prone due to the dynamic nature of the Cloud computing environment. In this context, the online validation of an issued certificate may provide incorrect results. Therefore, as part of our future work, we will consider the formulation of an approach allowing to manage errors in the monitoring data and accordingly adapt our service and certification models. For example, we will extend our STS based models with additional Bayesian networks. Furthermore, starting from the STS based model of the service, we will develop techniques to automate the process of obtaining the certification model so as to avoid any human errors.

Certification of Cloud computing services. At present, most service providers are reluctant to take full responsibility of the dependability of their services once the services are uploaded and offered through a cloud. Also, Cloud computing providers refrain from accepting the liability for dependability flaws. This reluctance is due to the fact that the provision and dependability of a Cloud service is sensitive to changes in the environment, as well as to potential interference between the features and behavior of all the inter-dependent services in all layers of the Cloud stack. Our future work will focus on developing models that can integrate the failure behavior of services deployed in a Cloud environment.

7.2.2 Dependability Management

Composition of dependability solutions. Our adaptive resource management algorithm changes the number of replicas and their allocation to mask the affects of system failures; the simulation results show significant improvement over static solutions. Similarly, our future work will focus on designing an automated, modular approach to compose dependability solutions. We will implement dependability mechanisms as independent modules (dep_units) and, according to the user's requirements, compose a set of modules to form a complete solution. This will allow users to dynamically change the dimension and intensity of the dependability support based on their business goals. The approach of generating dynamic dependability support has not been studied in the literature.

Dependability Manager deployment. Our simulation results so far considered evaluation of individual techniques such as virtual machine consolidation and provisioning. As part of the future work, we must implement each component of the framework as individual web services and evaluate the effectiveness of the overall approach. This would also require development of additional techniques to address fault detection/prediction and system recovery. Another interesting work is to study the cost benefits of adopting our approach both for the service provider and the users.

7.2.3 Secure Application Execution

Virtual machine images selection. During secure application deployment, we must first map each task to an available VM image and then to a physical host. Existing Cloud computing services usually require users to manually select VM images from a repository. Typically, users can also upload and share their VM images with other customers. This feature exacerbates the security problems in public Cloud services, and such problems cannot be identified by the users in a straightforward manner during image selection. For example, Balduzzi et al. [18] studied the vulnerability issues in Amazon EC2 service by analyzing over 5,000 public images; using the Nessus vulnerability scanner, they identified that 98% of Windows AMIs (Amazon Machine Images) and 58% of Linux AMIs had software with critical vulnerabilities. This implies that an automated security-driven search scheme is required to deploy applications' tasks to appropriate VM images.

Dynamic application deployment. Instead of allocating resources to tasks for the entire duration of application execution, we must consider the execution time of each task and perform allocation only for necessary periods of time while minimizing its exposure to the vulnerabilities. The application model must be extended to include the start time and a deadline for each task so that the target execution timeline can be included, targeting real-time applications.

Incremental vulnerability analysis. Each allocation introduces a set of new services on a host and increases its vulnerability level. We need an approach to estimate the increase in the vulnerability level so as to facilitate the "what-if" analysis. One possible approach to vulnerability assessment is by means of attack graphs, and a naive method is to discard the original attack graph and perform re-computation from scratch using the new data. However, such re-computation is wasteful since typically the changes are small, resulting in information that is not very different from the original one. Therefore, we need to take an incremental approach that *i)* identifies the portions of the attack graph that have changed due to an event, *i)* re-computes the vulnerability information only in the changed portion, and *i)* combines the new and original information to provide updated results.

Bibliography

[1] B. Addis, D. Ardagna, B. Panicucci, and L. Zhang, "Autonomic management of cloud service centers with availability guarantees," in *Proc. of 3rd International Conference on Cloud Computing*, Miami, FL, USA, Jul 2010, pp. 220–227.

[2] M. Albanese, S. Jajodia, and S. Noel, "Time-efficient and cost-effective network hardening using attack graphs," in *42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Boston, MA, USA, Jun 2012, pp. 1–12.

[3] M. Albanese, S. Jajodia, A. Pugliese, and V. Subrahmanian, "Scalable analysis of attack scenarios," in *Proc. of 16th European Symposium on Research in Computer Security*, Leuven, Belgium.

[4] M. Albanese, S. Jajodia, R. Jhawar, and V. Piuri, "Reliable Mission Deployment in Vulnerable Distributed Systems," in *Proc. of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop*, Budapest, Hungary, Jun 2013, pp. 1–8.

[5] M. Albanese, S. Jajodia, R. Jhawar, and V. Piuri, "Securing Mission-Centric Operations in the Cloud," in *Secure Cloud Computing*, S. Jajodia, K. Kant, P. Samarati, A. Singhal, V. Swarup, and C. Wang, Eds. Springer, 2014, pp. 239–259.

[6] A. Alves and et al., *Web Services Business Process Execution Language Version 2.0*, OASIS, April 2007, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[7] M. Anisetti, C. Ardagna, and E. Damiani, "Fine-grained modeling of web services for test-based security certification," in *Proc. of 8th International Conference on Services Computing*, Washington, DC, USA, Jul 2011, pp. 456–463.

[8] M. Anisetti, C. Ardagna, and E. Damiani, "Security certification of composite services: A test-based approach," in *Proc. of 20th IEEE International Conference on Web Services*, Santa Clara, CA, USA, Jun–Jul 2013, pp. 475–482.

[9] M. Anisetti, C. Ardagna, E. Damiani, and J. Maggesi, "Security certification-aware service discovery and selection," in *Proc. of 5th International Conference on Service-Oriented Computing and Applications*, Taipei, Taiwan, Dec 2012, pp. 1–8.

[10] M. Anisetti, C. Ardagna, E. Damiani, and F. Saonara, "A test-based security certification scheme for web services," *ACM Transactions on the Web*, vol. 7, no. 2, p. 5, 2013.

[11] Apache Sandesha2. http://axis.apache.org/axis2/java/sandesha/.

[12] Apache axis2/java. http://axis.apache.org/axis2/java/core/.

[13] C. Ardagna, M. Cremonini, E. Damiani, S. De Capitani di Vimercati, and P. Samarati, "Supporting location-based conditions in access control policies," in *Proc. of the ACM Symposium on Information, Computer and Communications Security*, Taipei, Taiwan, Mar 2006, pp. 212–222.

[14] C. A. Ardagna, R. Jhawar, and V. Piuri, "Dependability Certification of Services: A Model-Based Approach," *Computing*, pp. 1–28, Oct 2013.

[15] C. Ardagna, E. Damiani, R. Jhawar, and V. Piuri, "A Model-Based Approach to Reliability Certification of Services," in *Proc. of the 6th IEEE International Conference on Digital Ecosystem Technologies - Complex Environment Engineering*, Campione d'Italia, Italy, Jun 2012, pp. 1–6.

[16] V. Atluri and H. Shin, "Efficient enforcement of security policies based on tracking of mobile users," in *Proc. of the 20th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, Sophia Antipolis, France, Jul-Aug 2006, pp. 237–251.

[17] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[18] M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro, "A security analysis of amazon's elastic compute cloud service," in *Proc. of the 27th Annual ACM Symposium on Applied Computing*, Trento, Italy, 2012, pp. 1427–1434.

[19] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future Generation Computer Systems*, vol. 28, no. 5, pp. 755–768, May 2012.

[20] L. Bentakouk, P. Poizat, and F. Zaïdi, "A formal framework for service orchestration testing based on symbolic transition systems," in *Proc. of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems*, Eindhoven, The Netherlands, Nov 2009, pp. 16–32.

[21] L. Bentakouk, P. Poizat, and F. Zaïdi, "Checking the behavioral conformance of web services with symbolic testing and an SMT solver," in *Proc. of 5th International Conference on Tests and Proofs*, Zurich, Switzerland, Jun 2011, pp. 33–50.

[22] E. Bin, O. Biran, O. Boni, E. Hadad, E. Kolodner, Y. Moatti, and D. Lorenz, "Guaranteeing high availability goals for virtual machine placement," in *Proc. of 31st International Conference on Distributed Computing Systems*, Minneapolis, MN, USA, Jun 2011, pp. 700–709.

[23] I. Buckley and et al., "Towards pattern-based reliability certification of services," in *Proc. of 1st International Symposium on Secure Virtual Infrastructures*, Crete, Greece, Oct 2011, pp. 560–576.

[24] R. Buyya, S. K. Garg, and R. N. Calheiros, "Sla-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions," in *Proc. of the International Conference on Cloud and Service Computing*, Hong Kong, China, Dec 2011, pp. 1–10.

[25] C. Clark et al., "Live migration of virtual machines," in *Proc. of the 2nd Symposium on Networked Systems Design and Implementation - Volume 2*, Boston, MA, USA, May 2005, pp. 273–286.

[26] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proc. of the 3rd symposium on Operating Systems Design and Implementation*, New Orleans, LA, USA, Feb 1999, pp. 173–186.

[27] R. C. Cheung, "A user-oriented software reliability model," *IEEE Transactions on Software Engineering*, vol. 6, pp. 118–125, Mar 1980.

[28] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, and P. Samarati, *Security in Decentralized Data Management.* K-Anonymity. Springer, 2007.

[29] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: high availability via asynchronous virtual machine replication," in *Proc. of the 5th USENIX Symposium on Networked Systems Design and Implementation*, San Francisco, USA, Apr 2008, pp. 161–174.

[30] I. Cunha, J. Almeida, V. Almeida, and M. Santos, "Self-adaptive capacity management for multi-tier virtualized environments," in *Proc. of 10th IFIP/IEEE International Symposium on Integrated Network Management*, Munich, Germany, May 2007, pp. 129–138.

[31] E. Damiani, C. Ardagna, and N. El Ioini, Eds., *Open source systems security certification.* New York, NY, USA: Springer, 2009.

[32] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati, "A fine-grained access control system for XML documents," *ACM Transactions on Information and System Security*, vol. 5, no. 2, pp. 169–202, May 2002.

[33] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati, "Securing SOAP e-services," *International Journal of Information Security*, vol. 1, no. 2, pp. 100–115, Feb 2002.

[34] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, and G. Livraga, "Enforcing subscription-based authorization policies in cloud scenarios," in *Proc. of the 26th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy*, Paris, France, July 2012, pp. 314–329.

[35] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, G. Livraga, S. Paraboschi, and P. Samarati, "Enforcing dynamic write privileges in data outsourcing," *Computers & Security*, vol. 39, no. A, pp. 47–63, 2013.

[36] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, G. Pelosi, and P. Samarati, "Encryption-based policy enforcement for cloud storage," in *Proc. of the 1st ICDCS Workshop on Security and Privacy in Cloud Computing*, Genova, Italy, Jun 2010, pp. 42–51.

[37] S. De Capitani di Vimercati, S. Foresti, G. Livraga, and P. Samarati, "Protecting privacy in data release," in *Foundations of Security Analysis and Design VI*, A. Aldini and R. Gorrieri, Eds. Springer, 2011.

[38] S. De Capitani di Vimercati, S. Foresti, G. Livraga, and P. Samarati, "Data privacy: Definitions and techniques," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 20, no. 6, pp. 793–817, Dec 2012.

[39] S. De Capitani di Vimercati, S. Foresti, and P. Samarati, "Managing and accessing data in the cloud: Privacy risks and approaches," in *Proc. of the 7th International Conference on Risks and Security of Internet and Systems*, Cork, Ireland, Oct 2012, pp. 1–9.

[40] S. De Capitani di Vimercati, P. Samarati, and S. Jajodia, "Policies, models, and languages for access control," in *Proc. of the Workshop on Databases in Networked Information Systems*, Aizu-Wakamatsu, Japan, Mar 2005.

[41] J. Deng, S.-H. Huang, Y. Han, and J. Deng, "Fault-tolerant and reliable computation in cloud computing," in *Proc. of IEEE Global Communications Conference Workshops*, Miami, FL, USA, Dec 2010, pp. 1601–1605.

[42] Z. Ding, M. Jiang, and A. Kandel, "Port-based reliability computing for service composition," *IEEE Transactions on Services Computing*, vol. 5, no. 3, pp. 422–436, 2012.

[43] F. Distante and V. Piuri, "Hill-climbing Heuristics for Optimal Hardware Dimensioning and Software Allocation in Fault-tolerant Distributed Systems," *IEEE Transactions on Reliability*, vol. 38, no. 1, pp. 28–39, Apr 1989.

[44] Amazon elastic compute cloud. http://aws.amazon.com/ec2/.

[45] Eucalyptus systems. http://www.eucalyptus.com/.

[46] E. Farr, R. Harper, L. Spainhower, and J. Xenidis, "A case for high availability in a virtualized environment," *Proc. of 7th International Conference on Availability, Reliability and Security*, pp. 675–682, Mar 2008.

[47] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi, "Enhanced modeling and solution of layered queueing networks," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 148–161, Mar 2009.

[48] L. Frantzen, J. Tretmans, and R. de Vries, "Towards model-based testing of web services," in *Proc. of the International Workshop on Web Services - Modeling and Testing*, Palermo, Italy, Jun 2006, pp. 67–82.

[49] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, Oct 2003.

[50] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of Consistent, Available, Partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun 2002.

[51] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," *ACM Computer Communication Review*, vol. 41, no. 4, pp. 350–361, 2011.

[52] R. Guerraoui and M. Yabandeh, "Independent faults in the cloud," in *Proc. of 4th International Workshop on Large Scale Distributed Systems and Middleware*, Zurich, Switzerland, Jul 2010, pp. 12–17.

[53] E. Hansen and R. Zhou, "Anytime heuristic search," *Journal of Artificial Intelligence Research*, vol. 28, no. 1, pp. 267–297, Mar 2007.

[54] F. Hermenier, J. Lawall, J. Menaud, and G. Muller, "Dynamic Consolidation of Highly Available Web Applications," INRIA, Tech. Rep. RR-7545, Feb 2011.

[55] D. Herrmann, *Using the common criteria for IT security evaluation.* Auerbach Publications, 2002.

[56] U. Hoelzle and L. A. Barroso, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 1st ed. Morgan and Claypool Publishers, 2009.

[57] S. Iyer, M. Nakayama, and A. Gerbessiotis, "A markovian dependability model with cascading failures," *IEEE Transactions on Computers*, vol. 58, pp. 1238–1249, Sep 2009.

[58] S. Jajodia, S. Noel, P. Kalapa, M. Albanese, and J. Williams, "Cauldron mission-centric cyber situational awareness with defense in depth," in *Proc. of Military Communications Conference*, Baltimore, MD, USA, Nov 2011, pp. 1339–1344.

[59] G. Jakobson, "Mission cyber security situation assessment using impact dependency graphs," in *Proc. of 14th International Conference on Information Fusion*, Chicago, IL, USA, Jul 2011, pp. 1–8.

[60] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, I. Whally, and E. Snible, "Improving performance and availability of services hosted on iaas clouds with structural constraint-aware virtual machine placement," in *Proc. of IEEE International Conference on Services Computing*, Washington, DC, USA, Jul 2011, pp. 72–79.

[61] P. A. Jensen. (2011) Operations Research Models and Methods – Markov Analysis Tools. Available at www.me.utexas.edu/jensen/ormm/excel/markov.html.

[62] R. Jhawar, P. Inglesant, N. Courtois, and M. Sasse, "Make mine a quadruple: Strengthening the security of graphical one-time PIN authentication," in *Proc. of 5th International Conference on Network and System Security*, Milan, Italy, Sep 2011, pp. 81–88.

[63] R. Jhawar and V. Piuri, "Fault Tolerance Management in IaaS Clouds," in *Proc. of the 1st IEEE-AESS Conference in Europe about Space and Satellite Telecommunications*, Rome, Italy, Oct 2012, pp. 1–6.

[64] R. Jhawar, V. Piuri, and M. Santambrogio, "A Comprehensive Conceptual System-Level Approach to Fault Tolerance in Cloud Computing," in *Proc. of IEEE International Systems Conference*, Vancouver, BC, Canada, Mar 2012, pp. 1–5.

[65] R. Jhawar and V. Piuri, "Adaptive Resource Management for Balancing Availability and Performance in Cloud Computing," in *Proc. of the 10th International Conference on Security and Cryptography*, Reykjavik, Iceland, Jul 2013, pp. 254–264.

[66] R. Jhawar and V. Piuri, "Fault Tolerance and Resilience in Cloud Computing Environments," in *Computer and Information Security Handbook, 2nd Edition.* Morgan Kaufmann, 2013, pp. 125–141.

[67] R. Jhawar and V. Piuri, "Dependability-oriented resource management schemes for cloud computing data centers," in *Handbook on Data Centers*, S. U. Khan and A. Y. Zomaya, Eds. Springer, 2014, (to appear).

[68] R. Jhawar, V. Piuri, and P. Samarati, "Supporting Security Requirements for Resource Management in Cloud Computing," in *Proc. of the 15th IEEE International Conference on Computational Science and Engineering*, Paphos, Cyprus, Dec 2012, pp. 170–177.

[69] R. Jhawar, V. Piuri, and M. Santambrogio, "Fault Tolerance Management in Cloud Computing: A System-Level Perspective," *IEEE Systems Journal*, vol. 7, no. 2, pp. 288–297, June 2013.

[70] G. Jung, K. Joshi, M. Hiltunen, R. Schlichting, and C. Pu, "Performance and availability aware regeneration for cloud based multitier applications," in *Proc. of 2010 IEEE/IFIP International Conference on Dependable Systems and Networks*, Chicago, IL, USA, July 2010, pp. 497–506.

[71] M. Kafil and I. Ahmad, "Optimal Task Assignment in Heterogeneous Distributed Computing Systems," *IEEE Concurrency*, vol. 6, no. 3, pp. 42–51, Jul 1998.

[72] S. Katsikas, J. Lopez, and G. Pernul, "Security, trust and privacy in digital business," *International Journal of Computer Systems, Science & Engineering, CRL Publishing*, Nov 2005.

[73] C. Keum, S. Kang, I.-Y. Ko, J. Baik, and Y.-I. Choi, "Generating test cases for web services using extended finite state machine," in *Proc. of 18th IFIP International Conference on Testing Communicating Systems*, New York, NY, USA, May 2006, pp. 103–117.

[74] S. Kim, F. Machida, and K. Trivedi, "Availability modeling and analysis of virtualized system," in *Proc. of 15th IEEE Pacific Rim International Symposium on Dependable Computing*, Shanghai, China, Nov 2009, pp. 365–371.

[75] S. Koenig and M. Likhachev, "Improved fast replanning for robot navigation in unknown terrain," in *Proc. of the International Conference on Robotics and Automation*, Washington, DC, USA, May 2002, pp. 968–975.

[76] G. Koslovski, W.-L. Yeow, C. Westphal, T. T. Huu, J. Montagnat, and P. Vicat-Blanc, "Reliability support in virtual infrastructures," in *Proc. of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, Indianapolis, USA, Nov 2010, pp. 49–58.

[77] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 45–58, Oct 2007.

[78] K. Kourai and S. Chiba, "A fast rejuvenation technique for server consolidation with virtual machines," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Edinburgh, UK, Jun 2007, pp. 245–255.

[79] D. Kourtesis, E. Ramollari, D. Dranidis, and I. Paraskakis, "Increased reliability in SOA environments through registry-based conformance testing of web services," *Production Planning & Control*, vol. 21, no. 2, pp. 130–144, 2010.

[80] M. Likhachev, D. I. Ferguson, G. J. Gordon, A. Stentz, and S. Thrun, "Anytime Dynamic A*: An Anytime, Replanning Algorithm," in *Proc. of International Conference on Automated Planning and Scheduling/Artificial Intelligence Planning Systems*, Monterey, USA, Jun 2005, pp. 262–271.

[81] M. Likhachev, G. Gordon, and S. Thrun, "ARA*: Anytime A* with Provable Bounds on Sub-Optimality," in *Proc. of Conference on Neural Information Processing Systems*, Vancouver, Canada, Dec 2003.

[82] M. Armbrust et al., "Above the clouds: A berkeley view of cloud computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009.

[83] F. Machida, M. Kawato, and Y. Maeno, "Redundant virtual machine placement for fault-tolerant consolidated server clusters," in *Proc. of 2010 IEEE Network Operations and Management Symposium*, Osaka, Japan, Apr 2010, pp. 32–39.

[84] Y. Mao, C. Liu, J. E. van der Merwe, and M. Fernandez, "Cloud resource orchestration: A data-centric approach," in *Proc. of the 5th Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, USA, Jan 2011, pp. 241–248.

[85] R. Mateescu and S. Rampacek, "Formal modeling and discrete-time analysis of BPEL web services," in *Advances in Enterprise Engineering I*, ser. Lecture Notes in Business Information Processing. Springer, 2008, vol. 10, pp. 179–193.

[86] V. Mehta, C. Bartzis, H. Zhu, E. Clarke, and J. Wing, "Ranking attack graphs," in *Proc. of the 9th international conference on Recent Advances in Intrusion Detection*, Hamburg, Germany, Sep 2006, pp. 127–144.

[87] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *Proc. of 29th IEEE Conference on Computer Communications*, San Diego, California, USA, Mar 2010, pp. 1–9.

[88] K. Mills, J. Filliben, and C. Dabrowski, "Comparing VM-Placement Algorithms for On-Demand Clouds," in *Proc. of the 2011 IEEE 3rd International Conference on Cloud Computing Technology and Science*, Washington, DC, USA, Jul 2011, pp. 91–98.

[89] K. Mishra and K. Trivedi, "Model based approach for autonomic availability management," in *Service Availability*, D. Penkler, M. Reitenspiess, and F. Tam, Eds. Springer, 2006, vol. 4328, pp. 1–16.

[90] J. Muppala, M. Malhotra, and K. Trivedi, "Markov dependability models of complex systems: Analysis techniques," *Reliability and Maintenance of Complex Systems, NATO ASI Series F: Computer and Systems Sciences*, vol. 154, pp. 442–486, 1996.

[91] S. Mustafiz, X. Sun, J. Kienzle, and H. Vangheluwe, "Model-driven assessment of system dependability," *Software and System Modeling*, vol. 7, no. 4, pp. 487–502, 2008.

[92] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive fault tolerance for hpc with xen virtualization," in *Proc. of the 21st annual international conference on Supercomputing*, Seattle, Washington, May–Jun 2007, pp. 23–32.

[93] P. Narasimhan, K. Kihlstrom, L. Moser, and P. Melliar-Smith, "Providing support for survivable corba applications with the immune system," in *Proc. of the 19th IEEE International Conference on Distributed Computing Systems*, Austin, TX, USA, May 1999, pp. 507–516.

[94] Open nebula project. http://opennebula.org/.

[95] J. Ni, N. Li, and W. H. Winsborough, "Automated trust negotiation using cryptographic credentials," in *Proc. of the 12th ACM Conference on Computer and Communications Security*, Alexandria, VA, USA, Nov 2005, pp. 46–57.

[96] N. J. Nilsson, *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill Pub. Co., 1971.

[97] Nimbus project. http://www.nimbusproject.org/.

[98] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus Open-Source Cloud-Computing System," in *9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, Shanghai, China, May 2009, pp. 124–131.

[99] M. Papazoglou, "Web services and business transactions," *World Wide Web*, vol. 6, pp. 49–91, Mar 2003.

[100] J. Pathak, S. Basu, and V. Honavar, "Modeling web service composition using symbolic transition systems," in *Proc. of AAAI Workshop on AI-Driven Technologies for Service-Oriented Computing*, Boston, MA, USA, Jul 2006, pp. 44–51.

[101] V. Piuri, "Design of fault-tolerant distributed control systems," *IEEE Transactions on Instrumentation and Measurement*, vol. 43, no. 2, pp. 257–264, Apr 1994.

[102] I. Pohl, "First results on the effect of error in heuristic search," *Machine Intelligence*, vol. 5, pp. 219–236, 1970.

[103] C. Pu, J. Noe, and A. Proudfoot, "Regeneration of replicated objects: a technique and its eden implementation," *IEEE Transactions on Software Engineering*, vol. 14, no. 7, pp. 936–945, 1988.

[104] H. Qian, D. Medhi, and T. Trivedi, "A hierarchical model to evaluate quality of experience of online services hosted by cloud computing," in *Proc. of IFIP/IEEE International Symposium on Integrated Network Management*, Dublin, Ireland, May 2011, pp. 105–112.

[105] E. Riccobene, P. Potena, and P. Scandurra, "Reliability prediction for service component architectures with the SCA-ASM component model," in *Proc. of 38th EUROMICRO Conference on Software Engineering and Advanced Applications*, Cesme, Izmir, Turkey, Sep 2012, pp. 125–132.

[106] L. U. Rina Panigrahy, Kunal Talwar and U. Wieder, "Heuristics for vector bin packing," 2011, Microsoft Research, (unpublished).

[107] S. Salva, P. Laurencot, and I. Rabhi, "An approach dedicated for web service security testing," in *Proc. of 5th International Conference on Software Engineering Advances*, Nice, France, Aug 2010, pp. 494–500.

[108] S. Salva and I. Rabhi, "Automatic web service robustness testing from WSDL descriptions," in *Proc. of 12th European Workshop on Dependable Computing*, Toulouse, France, May 2009, pp. 1–8.

[109] P. Samarati, "Protecting respondents' identities in microdata release," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 6, pp. 1010–1027, Nov 2001.

[110] P. Samarati and S. De Capitani di Vimercati, "Access control: Policies, models, and mechanisms," in *Foundations of Security Analysis and Design*, ser. LNCS 2171, R. Focardi and R. Gorrieri, Eds. Springer-Verlag, 2001.

[111] P. Samarati and S. De Capitani di Vimercati, "Data Protection in Outsourcing Scenarios: Issues and Directions," in *Proc. of the 5th ACM Symposium on Information, Computer and Communications Security*, Beijing, China, Apr 2010, pp. 1–14.

[112] G. T. Santos, L. C. Lung, and C. Montez, "FTWeb: A Fault Tolerant Infrastructure for Web Services," in *Proc. of the 9th IEEE International EDOC Enterprise Computing Conference*, Enschede, The Netherlands, Sep 2005, pp. 95–105.

[113] C.-C. Shen and W.-H. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE Transactions on Computers*, vol. C-34, no. 3, pp. 197–203.

[114] L. Shi, B. Butler, D. Botvich, and B. Jennings, "Provisioning of requests for virtual machine sets with placement constraints in iaas clouds," in *Proc. of IFIP/IEEE International Symposium on Integrated Network Management*, Ghent, Belgium, May 2013, pp. 499–505.

[115] K. Shin, C. M. Krishna, and Y. hang Lee, "Optimal dynamic control of resources in a distributed system," *IEEE Transactions on Software Engineering*, vol. 15, no. 10, pp. 1188–1198, Oct 1989.

[116] L. M. Silva, J. Alonso, and J. Torres, "Using virtualization to improve software rejuvenation," *IEEE Transactions on Computers*, vol. 58, no. 11, pp. 1525–1538, Nov 2009.

[117] A. Singh, M. Korupolu, and D. Mohapatra, "Server-storage virtualization: Integration and load balancing in data centers," in *Proc. of International Conference for High Performance Computing, Networking, Storage and Analysis*, Austin, TX, USA, Nov 2008, pp. 1–12.

[118] W. E. Smith, K. S. Trivedi, L. A. Tomek, and J. Ackaret, "Availability analysis of blade server systems," *IBM Systems Journal*, vol. 47, no. 4, pp. 621–640, 2008.

[119] Y. Tamura, K. Sato, S. Kihara, and S. Moriai, "Kemari: Virtual Machine Synchronization for Fault Tolerance," in *Proc. of USENIX Annual Technical Conference*, Boston, MA, USA, 2008.

[120] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici, "A scalable application placement controller for enterprise data centers," in *Proc. of the 16th international conference on World Wide Web*, Banff, Alberta, Canada, May 2007, pp. 331–340.

[121] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici, "A scalable application placement controller for enterprise data centers," in *Proc. of 16th International conference on World Wide Web*, Alberta, Canada, May 2007, pp. 331–340.

[122] T. Thein and J. S. Park, "Availability analysis of application servers using software rejuvenation and virtualization," *Journal of Computer Science and Technology*, vol. 24, no. 2, pp. 339–346, Mar 2009.

[123] T. Thein, S. D. Chi, and J. S. Park, "Availability modeling and analysis on virtualized clustering with rejuvenation," *Journal of Computer Science and Network Security*, vol. 8, no. 9, pp. 72–80, Mar 2008.

[124] J. Tretmans, "Model-based testing and some steps towards test-based modelling," in *Proc. of 11th International School on Formal Methods for Eternal Networked Software Systems*, Bertinoro, Italy, Jun 2011, pp. 297–326.

[125] K. S. Trivedi, K. Vaidyanathan, and K. Goseva-Popstojanova, "Modeling and analysis of software aging and rejuvenation," in *Proc. of the 33rd Annual Simulation Symposium*, Washington, DC, USA, Apr 2000, pp. 270–279.

[126] K. Trivedi, D. S. Kim, A. Roy, and D. Medhi, "Dependability and security models," in *Proc. of 7th International Workshop on Design of Reliable Communication Networks*, Washington, DC, USA, Oct 2009, pp. 11–20.

[127] A. Undheim, A. Chilwan, and P. Heegaard, "Differentiated availability in cloud computing slas," in *Proc. of 12th IEEE/ACM International Conference on Grid Computing*, Lyon, France, Sep 2011, pp. 129–136.

[128] *Department Of Defense Trusted Computer System Evaluation Criteria*, USA Department of Defence, Dec 1985, http://csrc.nist.gov/publications/secpubs/rainbow/std001.txt.

[129] W. van der Aalst, N. Lohmann, and M. La Rosa, "Ensuring correctness during process configuration via partner synthesis," *Journal of Information Systems*, vol. 37, no. 6, pp. 574–592, Sep 2012.

[130] E. van Veenendaal, *Standard glossary of terms used in Software Testing Version 2.2*, International Software Testing Qualifications Board, Oct 2012, http://www.astqb.org/documents/ISTQB_glossary_of_testing_terms_2.2.pdf, Accessed in August 2013.

[131] W. E. Vesely and N. H. Roberts, *Fault Tree Handbook*. Government Printing Office: U.S. Nuclear Regulatory Commission, 1987.

[132] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *Proc. of the 1st ACM symposium on Cloud computing*, Indianapolis, IN, USA, Jun 2010, pp. 193–204.

[133] VMware, "White paper: Vmware high availability concepts, implementation and best practices," 2007.

[134] VMware, "White paper: Protecting mission-critical workloads with vmware fault tolerance," 2009.

[135] L. Wang, M. Kunze, J. Tao, and G. von Laszewski, "Towards building a cloud for scientific applications," *Advances in Engineering Software*, vol. 42, no. 9, pp. 714–722, Sep 2011.

[136] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Sandpiper: Black-box and gray-box resource management for virtual machines," *Computer Networks*, vol. 53, no. 17, Dec 2009.

[137] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet, "ZZ and the art of practical BFT execution," in *Proc. of 6th Conference on Computer Systems*, Salzburg, Austria, Apr 2011, pp. 123–138.

[138] OASIS Web Services Reliable Messaging (WSRM). https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm.

[139] B. Yang, X. Xu, F. Tan, and D.-H. Park, "An utility-based job scheduling algorithm for cloud computing considering reliability factor," in *Proc. of International Conference on Cloud and Service Computing*, Hong Kong, China, Dec 2011, pp. 95–102.

[140] Y. Zhang, Z. Zheng, and M. Lyu, "Bftcloud: A byzantine fault tolerance framework for voluntary-resource cloud computing," in *Proc. of IEEE International Conference on Cloud Computing*, Washington, DC, USA, July 2011, pp. 444–451.

[141] W. Zhao, "BFT-WS: A Byzantine Fault Tolerance Framework for Web Services," in *Proc. of the 11th International Enterprise Distributed Object Computing Conference Workshop*, Annapolis, MD, USA, Oct 2007, pp. 89–96.

[142] W. Zhao, P. M. Melliar-Smith, and L. E. Moser, "Fault Tolerance Middleware for Cloud Computing," in *Proc. of the 3rd International Conference on Cloud Computing*, Miami, FL, USA, Jul 2010, pp. 67–74.

[143] Z. Zheng, M. Li, X. Xiao, and J. Wang, "Coordinated resource provisioning and maintenance scheduling in cloud data centers," in *Proc. of IEEE International Conference on Computer Communications*, Turin, Italy, April 2013, pp. 345–349.

A

Publications

Refereed Papers in International Journals

(1) Dependability Certification of Services: A Model-Based Approach.

(co-authors: C. A. Ardagna, V. Piuri)

in Springer Computing Journal, 2013

Abstract. The advances and success of the Service-Oriented Architecture (SOA) paradigm have produced a revolution in ICT, particularly, in the way in which software applications are implemented and distributed. Today, applications are increasingly provisioned and consumed as web services over the Internet, and business processes are implemented by dynamically composing loosely-coupled applications provided by different suppliers. In this highly dynamic context, clients (e.g., business owners or users selecting a service) are concerned about the dependability of their services and business processes.

In this paper, we define a certification scheme that allows to verify the dependability properties of services and business processes. Our certification scheme relies on discrete-time Markov chains and awards machine-readable dependability certificates to services, whose validity is continuously verified using run-time monitoring. Our solution can be integrated within existing SOAs, to extend the discovery and selection process with dependability requirements and certificates, and to support a dependability-aware service composition.

(2) Fault Tolerance Management in Cloud Computing: A System-Level Perspective.

(co-authors: V. Piuri, M. Santambrogio)

in IEEE Systems Journal, 2013

Abstract. The increasing popularity of Cloud computing as an attractive alternative to classic information processing systems has increased the importance of its

correct and continuous operation even in the presence of faulty components. In this paper, we introduce an innovative, system-level, modular perspective on creating and managing fault tolerance in Clouds. We propose a comprehensive high-level approach to shading the implementation details of the fault tolerance techniques to application developers and users by means of a dedicated service layer. In particular, the service layer allows the user to specify and apply the desired level of fault tolerance, and does not require knowledge about the fault tolerance techniques that are available in the envisioned Cloud and their implementations.

Refereed Papers in International Conferences and Workshops

(3) Adaptive Resource Management for Balancing Availability and Performance in Cloud Computing.

(co-authors: V. Piuri)

in Proc. of the 10th International Conference on Security and Cryptography (SECRYPT 2013)

Abstract. Security, availability and performance are critical to meet service level agreements in most Cloud computing services. In this paper, we build on the virtual machine technology that allows software components to be cheaply moved, replicated, and allocated on the hardware infrastructure to devise a solution that ensures users availability and performance requirements in Cloud environments. To deal with failures and vulnerabilities also due to cyber-attacks, we formulate the availability and performance attributes in the users perspective and show that the two attributes may often be competing for a given application. We then present a heuristics-based approach that restores application's requirements in the failure and recovery events. Our algorithm uses Markov chains and queuing networks to estimate the availability and performance of different deployment contexts, and generates a set of actions to re-deploy a given application. By simulation, we show that our proposed approach improves the availability and lowers the degradation of system's response time compared to traditional static schemes.

(4) Reliable Mission Deployment in Vulnerable Distributed Systems.

(co-authors: M. Albanese, S. Jajodia, V. Piuri)

in Proc. of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-RSDA 2013)

Abstract. Recent years have seen a growing interest in mission-centric operation of large-scale distributed systems. However, due to their complexity, these systems are prone to failures and vulnerable to a wide range of cyber-attacks. Current solutions

focus either on the infrastructure itself or on mission analysis, but fail to consider information about the complex interdependencies existing between system components and mission tasks. In this paper, we take a different approach, and present a solution for deploying mission tasks in a distributed computing environment in a way that minimizes a mission's exposure to vulnerabilities by taking into account available information about vulnerabilities and dependencies. We model the mission deployment problem as a task allocation problem, subject to various dependability constraints. The proposed solution is based on the A* algorithm for searching the solution space, but we also introduce a heuristic to significantly improve the search performance. We validate our approach, and show that our algorithm scales linearly with the size of both missions and networks.

(5) Supporting Security Requirements for Resource Management in Cloud Computing.

(co-authors: V. Piuri, P. Samarati)

in Proc. of the 15th IEEE International Conference on Computational Science and Engineering (CSE 2012)

Abstract. We address the problem of guaranteeing security, with additional consideration on reliability and availability issues, in the management of resources in Cloud environments. We investigate and formulate different requirements that users or service providers may wish to specify. Our framework allows providers to impose restrictions on the allocations to be made to their hosts and users to express constraints on the placement of their virtual machines (VMs). User's placement constraints may impose restrictions in performing allocation to specific locations, within certain boundaries, or depending on some conditions (e.g., requiring a VM to be allocated to a different host wrt other VMs). Our approach for VM allocation goes beyond the classical (performance/cost-oriented) resource consumption to incorporate the security requirements specified by users and providers.

(6) Fault Tolerance Management in IaaS Clouds.

(co-authors: V. Piuri)

in Proc. of the 1st IEEE-AESS Conference in Europe about Space and Satellite Telecommunications (ESTEL 2012)

Abstract. Fault tolerance, reliability and availability in Cloud computing are critical to ensure correct and continuous system operation also in the presence of failures. In this paper, we present an approach to evaluate fault tolerance mechanisms that use the virtualization technology to transparently increase the reliability and availability of applications deployed in the virtual machines in a Cloud. In contrast to several existing solutions that assume independent failures, we take into account the

failure behavior of various server components, network and power distribution in a typical Cloud computing infrastructure, the correlation between individual failures, and the impact of each failure on user's applications. We use this evaluation to study fault tolerance mechanisms under different deployment contexts, and use it as the basis to develop a methodology for identifying and selecting mechanisms that match user's fault tolerance requirements.

(7) A Model-Based Approach to Reliability Certification of Services.

(co-authors: C.A. Ardagna, E. Damiani, V. Piuri)

in Proc. of the 6th IEEE International Conference on Digital Ecosystem Technologies - Complex Environment Engineering (DEST-CEE 2012)

Abstract. We present a reliability certification scheme in which services are modeled as discrete-time Markov chains. A machine-readable certificate is issued to the service after validating its reliability properties, and validity of the certificate is verified using constant run-time monitoring. In addition, we present a solution that allows users to search and select services with a given set of reliability properties. Our solution is integrated within existing Service-Oriented Architectures (SOAs), and allows validation of users' preferences both at discovery-time and at run-time.

(8) A Comprehensive Conceptual System-Level Approach to Fault Tolerance in Cloud Computing.

(co-authors: V. Piuri, M. Santambrogio)

in Proc. of the IEEE International Systems Conference (SysCon 2012)

Abstract. Fault tolerance, reliability and resilience in Cloud Computing are of paramount importance to ensure continuous operation and correct results, even in the presence of a given maximum amount of faulty components. Most existing research and implementations focus on architecture-specific solutions to introduce fault tolerance. This implies that users must tailor their applications by taking into account environment-specific fault tolerant features. Such a need results in non transparent and inflexible Cloud environments, requiring too much effort to developers and users. This paper introduces an innovative perspective on creating and managing fault tolerance that shades the implementation details of the reliability techniques from the users by means of a dedicated service layer. This allows users to specify and apply the desired level of fault tolerance without requiring any knowledge about its implementation.

(9) Make Mine a Quadruple: Strengthening the Security of Graphical One-Time PIN authentication.

(co-authors: P. Inglesant, N. Courtois, A. Sasse)

in Proc. of the 5th International Conference on Network and System Security (NSS 2011)

Abstract. Secure and reliable authentication is an essential prerequisite for many online systems, yet achieving this in a way which is acceptable to customers remains a challenge. GrIDsure, a one-time PIN scheme using random grids and personal patterns, has been proposed as a way to overcome some of these challenges. We present an analytical study which demonstrates that GrIDsure in its current form is vulnerable to interception. To strengthen the scheme, we propose a way to fortify GrIDsure against Man-in-the-Middle attacks through *i)* an additional secret transmitted out-of-band and *ii)* multiple patterns. Since the need to recall multiple patterns increases user workload, we evaluated user performance with multiple captures with 26 participants making 15 authentication attempts each over a 3-week period. In contrast with other research into the use of multiple graphical passwords, we find no significant difference in the usability of GrIDsure with single and with multiple patterns.

Chapters in Books

(10) Securing Mission-Centric Operations in the Cloud.

(co-authors: M. Albanese, S. Jajodia, V. Piuri)

in Secure Cloud Computing, (S. Jajodia, K. Kant, P. Samarati, V. Swarup, C. Wang eds.), Springer, 2013

Abstract. Recent years have seen a growing interest in the use of Cloud Computing facilities to execute critical missions. However, due to their inherent complexity, most Cloud Computing services are vulnerable to multiple types of cyber-attacks and prone to a number of failures. Current solutions focus either on the infrastructure itself or on mission analysis, but fail to consider the complex interdependencies between system components, vulnerabilities, failures, and mission tasks. In this chapter, we propose a different approach, and present a solution for deploying missions in the cloud in a way that minimizes a mission's exposure to vulnerabilities by taking into account available information about vulnerabilities and dependencies. We model the mission deployment problem as a task allocation problem, subject to various dependability constraints, and propose a solution based on the A* algorithm for searching the solution space. Additionally, in order to provide missions with further availability and fault tolerance guarantees, we propose a cost-effective approach to harden the set of computational resources that have been selected for executing a given mission. Finally, we consider offering fault tolerance as a service to users in need of deploying missions in the Cloud. This approach allows missions to obtain required fault tolerance guarantees from a third party in a transparent manner.

(11) Fault Tolerance and Resilience in Cloud Computing Environments.

(co-authors: V. Piuri)

in Computer and Information Security Handbook, 2nd Edition, (J. Vacca ed.), Morgan Kaufmann, 2013

Abstract. The increasing demand for flexibility and scalability in dynamically obtaining and releasing computing resources in a cost-effective and device-independent manner, and easiness in hosting applications without the burden of installation and maintenance has resulted in a wide adoption of the Cloud computing paradigm. While the benefits are immense, this computing paradigm is still vulnerable to a large number of system failures and, as a consequence, there is an increasing concern among users regarding the reliability and availability of Cloud computing services. Fault tolerance and resilience serve as an effective means to address user's reliability and availability concerns. In this chapter, we focus on characterizing the recurrent failures in a typical Cloud computing environment, analyzing the effects of failures on user's applications, and surveying fault tolerance solutions corresponding to each class of failures. We also discuss the perspective of offering fault tolerance as a service to user's applications as one of the effective means to address user's reliability and availability concerns.

(12) Dependability-Oriented Resource Management Schemes for Cloud Computing Data Centers.

(co-authors: V. Piuri)

in Handbook on Data Centers (S. U. Khan, A. Y. Zomaya eds.), Springer, 2014

Abstract. Recent years have seen a growing interest among users to migrate their applications and services to large-scale data center environments. Due to their high complexity, these data centers can experience a large number of failures and security breaches, and consequently, impose numerous challenges in dependable service provisioning to the users. A number of autonomic computing solutions have been developed to provision resources among running applications based on short-term demand estimates. However, existing solutions have largely considered only performance and energy trade-off with a lower emphasis on data center's dependability features. In this chapter, we investigate two aspects of resource management: *i)* dependable application deployment in data centers, and *ii)* adaptive resource management to ensure high availability and performance at runtime. In particular, we discuss dependability-aware, application-centric, resource management policies in Cloud computing data centers.

Papers in preparation for International Journals

(13) Security-Oriented Elastic Resource Management in Cloud Computing.

(co-authors: M. Albanese, S. Jajodia, V. Piuri)

Abstract. Cloud computing services are vulnerable to multiple types of cyber-attacks and prone to a number of failures. As a consequence, there is a growing interest among users to deploy their applications in the Cloud, in the manner that minimizes their application's exposure to the vulnerabilities in the system. Existing task allocation techniques are deterministic in the way that they provide a feasible solution only when the algorithm has been completely executed. However, in the Cloud environment, the processing time available to the resource management algorithms may be very limited and often varying. In this paper, we present an interruptible, elastic, task allocation algorithm whose optimality improves as the execution time provided to it increases. Our design of such algorithm is based on the principles of Anytime processing, built using the A^* search scheme. We validate our approach, and show that elasticity and interruptibility can be obtained with a very low overhead. We also discuss an approach to build an elastic resource management algorithm that dynamically adapts the current allocation of a given application as a response to the changes in the system.