UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

DIVERSITY IN AUTOMATIC CLOUD COMPUTING RESOURCE SELECTION

Vinicius Vielmo Cogo

MESTRADO EM INFORMÁTICA

2011

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

DIVERSITY IN AUTOMATIC CLOUD COMPUTING RESOURCE SELECTION

Vinicius Vielmo Cogo

DISSERTAÇÃO

Projecto orientado pelo Prof. Doutor Marcelo Pasin

MESTRADO EM INFORMÁTICA

2011

Acknowledgments

Initially, I would like to thank my advisor, Professor Marcelo Pasin, for all lessons given to me day after day. Since before I came to Lisbon to the present days he has supported, encouraged and taught me the value in completing each job with confidence and willingness to continue. This continuity always came with new opportunities, which I will follow intending to transform into new achievements for both.

I render also thanks to my colleagues in the Navigators group and the LaSIGE. Their work inspired me and was an important starting point for this work, for example from Professors Phd. Paulo Verissimo, Nuno Neves, Miguel Correia, Anotnio Casimiro, Alysson Bessani, Paulo Sousa, Hans Reiser and Marcelo Pasin, which range from intrusion tolerance basic concepts, to BFT replication, proactive recovery and diversity mechanisms. To all my professors from master and undergraduate courses by all teaching on computer science and regarding life, in special to Andrea Charão, my undergraduate advisor.

Many colleagues became great friends and I would like to thank them also for their support, patience and encouragement. In special to João Antunes, Monica Dixit, Giuliana Veronese, Marcelo Pasin, Patricia Gonçalves, Bruno Vavala and André Nogueira by receiving me as a brother or even as a son. For those who have already left and for those who recently arrived in Lisbon by their fellowship and interesting lunch-time discussions. To those that contributed directly to this thesis, with constructive criticism, reviews and ideas: Marcelo Pasin, André Nogueira, Diego Kreutz, João Antunes, Miguel Garcia and Alysson Bessani.

Many other friends make me feel at home, indirectly contributing to this thesis but directly contributing to this life stage. Mainly to Rudra Dixit, Leandro Palma and Márcia Palma that already left Lisbon, but will be remembered wherever they go. I would like to also thank Emanuel Falcão, Paulo Carapinha, Marcírio Chaves, Cristiane Pedron and Simão Fontes.

Last but not least, I must thank all my family by their unconditional support, patience and encouragement even I being physically away from most of them. To my parents Marco and Sandra, my step-parents Juliana and Luis and my brothers and sister Vitor, Mateus and Vitória. A special thank to Leticia, my fiancé, by her love, happiness, confidence and by being always more than I expect, a long-standing light in my life. To my

grandparents, uncles and cousins for being my motivators and making everything be possible. To all my friends from Brazil that even a little bit far are always encouraging me with force and happiness.

This work was partially supported by the Fundação da Ciência e Tecnologia, through the project PTDC/EIA-CCO/108299/2008 (CloudFIT), by the European Commission through the FP7-ICT program under project TClouds, number 257243, and the Large-Scale Informatic Systems Laboratory (LaSIGE). Experiments presented in this work were carried out using *Quinta*, the Navigators experimental testbed.

A toda gente pá!

Resumo

Obter resultados e comportamentos correctos em computação é uma preocupação de longa data. O excerto seguinte sobre o advento das máquinas de calcular foi escrito em 1834 e ilustra a importância já dada naquela época ao uso de mecanismos para tolerar e identificar erros de cálculo [24]:

> *"A verificação mais correcta e efectiva contra erros que surgem do processo de computação é realizar a mesma computação em máquinas de calcular separadas e independentes; e tal verificação é ainda mais decisiva se os cálculos forem realizados através de métodos diferentes."*

Existem dois mecanismos que surgem desta afirmação e são considerados importantes para obter computações correctas. O primeiro é a *replicação*, a qual consiste em calcular os resultados mais de uma vez e compará-los ou realizar uma votação no final. O segundo é a *diversidade*, a qual consiste em utilizar métodos e componentes distintos em cada computação. Actualmente, ambos integram o grupo de mecanismos para tolerância a faltas e intrusões (FIT), os quais são capazes de tolerar tanto faltas acidentais como maliciosas em sistemas computacionais.

Em termos práticos, um serviço replicado pode tolerar faltas acidentais se existir pelo menos um servidor no seu grupo de réplicas que ainda seja capaz de responder aos pedidos dos clientes. O mesmo serviço replicado pode tolerar faltas maliciosas, normalmente, se a maioria das réplicas responderem correctamente ou concordarem com o resultado dos pedidos dos clientes.

Caso um atacante descubra uma vulnerabilidade que possa ser explorada em um servidor, e a mesma também existir em outras réplicas, então a tolerância a faltas e intrusões do serviço pode ser comprometida. Tal problema é uma limitação conhecida dos mecanismos de replicação frente a vulnerabilidades comuns entre as réplicas. Aumentar a independência de vulnerabilidades é o principal objectivo do mecanismo de diversidade.

A diversidade é um mecanismo que consiste em fornecer e criar diversas combinações de recursos entre os componentes de um sistema. Obtê-la automaticamente é um processo que pode ser decomposto em duas fases: criação e selecção. A primeira consiste em fornecer recursos diferentes o suficiente para serem considerados, combinados e seleccionados

na segunda fase. A obtenção automática de diversidade na fase de selecção de recursos é o nosso principal objectivo nesta dissertação.

Gerir grandes quantidades de recursos computacionais é uma tarefa complexa que pode ser facilitada com o uso de ferramentas automáticas para alocação, utilização e monitorização. Actualmente, pensar na gestão de sistemas distribuídos em larga escala implicitamente leva a considerar ferramentas de *cloud computing* como uma das opções de gestão. O modelo de *cloud computing*, na sua definição mais simples, é um modelo de fornecimento de computação como um serviço de utilidade [20]. Porém tecnicamente, este modelo e seus agentes são fontes infinitas de recursos computacionais, administrados automaticamente e fornecidos publicamente. Neste trabalho, nós consideramos *cloud computing* como o cenário para atingirmos nosso objectivo principal.

Considerando que o fornecedor de um serviço replicado seja cliente de um dado serviço de *cloud*, e que todas as réplicas do serviço são alocadas nesta mesma infraestrutura. Se uma falta, seja ela por paragem ou arbitrária, causar uma interrupção do serviço prestado por essa *cloud*, então o serviço replicado pode falhar na sua totalidade, o que significa que não existe independência de vulnerabilidade entre as réplicas do serviço. Neste caso, existe um ponto único de falha, o provedor de *cloud*, o que leva a indicação da diversidade deste componente uma possível solução para o caso.

O primeiro passo para obter diversidade de provedor é criar novas contas em outros fornecedores. O segundo passo consiste em seleccionar, para cada nova réplica do serviço, um fornecedor disponível que não esteja a ser utilizado pelas outras réplicas. Contudo, seleccionar manualmente um fornecedor de *cloud* para cada nova alocação pode ser inconveniente, ou até mesmo inviável, o que torna imperativo o uso de uma ferramenta automática para selecção de recursos.

Nesta dissertação, nós apresentamos o DiversityAgent, uma biblioteca em Java para obtenção automática de diversidade na selecção de recursos de *cloud computing*. Seus clientes apenas precisam registar quais são os recursos disponíveis, que o DiversityAgent se responsabiliza por seleccionar uma combinação de recursos diferente para cada nova réplica a ser alocada e implantada. Acreditamos nesta ser a primeira biblioteca automática com tal propósito, tendo em vista conformidade, extensibilidade, escalabilidade e outros requisitos. O DiversityAgent foi projectado tendo em vista quatro requisitos funcionais, nove não funcionais e alguns padrões de projecto bastante difundidos. O fluxo do algoritmo principal de selecção de recursos é baseado em uma proposta colaborativa entre as diversidades registadas no momento de cada pedido, o qual será discutido no decorrer deste documento. Também são apresentadas a composição interna do DiversityAgent e os algoritmos de diversidade e controladores para *cloud* implementados.

A biblioteca DiversityAgent é um *software* livre e de código aberto que se encontra disponibilizada no Google Project Hosting [10] sobre a licença GNU Lesser General Public License (LGPL v3.0). Esperamos que a mesma possa contribuir com muitos pro-

jectos do grupo Navigators, assim como externos em busca de solucionar os problemas ainda considerados em aberto na área de gestão de diversidade. Incentivamos o desenvolvimento de novos algoritmos e propriedades de diversidade, assim como novos *drivers* para mais provedores e ferramentas de *cloud* e esperamos poder publicar contribuições da comunidade de software livre para com esta ferramenta em futuras versões oficiais.

Além disso, nós realizamos uma ampla análise de diversidade no cenário de *cloud computing*. Este estudo é composto por uma revisão de taxonomia e discussão sobre cada uma das classificações, onde apontamos as propriedades que actualmente são suportadas pelos fornecedores e ferramentas de *cloud*. Nele, apresentamos também algumas oportunidades para que os agentes de *cloud computing* possam contribuir ainda mais com a área de gestão de diversidade. Mais de cinquenta propriedades foram identificadas, sendo quatro relativas à diversidade de aplicação, catorze à diversidade administrativa, dez de localização geográfica, nove de *software* de suporte, nove de *hardware* e seis relativas à diversidade de segurança. Do total de cinquenta e duas propriedades, apenas oito são completamente suportadas pela versão analisada da ferramenta para *cloud computing* OpenNebula e treze pelo fornecedor de *cloud* Amazon. Ainda em relação à Amazon, outras dezoito propriedades são parcialmente suportadas através do uso de rótulos genéricos, totalizando trinta e uma propriedades suportadas. Os provedores de *cloud computing* podem vir a não concordar em fornecer informações relativas a todas as propriedades definidas nesta dissertação, uma vez que existem riscos comerciais e custos extras em publicar e manter todas informações. Porém, ainda assim consideramos importante para a área de gestão de diversidade a apresentação e discussão do maior número possível de propriedades.

Nós também apresentamos a integração do DiversityAgent com dois casos de uso previstos pelo projecto CloudFIT, assim como os resultados dos experimentos de desempenho e conformidade. O primeiro caso é um serviço *Web* sem estado e o segundo é um serviço baseado em replicação de máquinas de estado. Ambos casos utilizam técnicas de recuperação proactiva e posicionam o DiversityAgent entre o gestor de recursos dos serviços e os provedores de *cloud*, a fim de obter diversidade automaticamente a cada nova troca proactiva de réplicas.

No fim desta dissertação, encontram-se as conclusões obtidas com este trabalho, possíveis trabalhos futuros, além de três apêndices sobre as interfaces públicas, tutoriais de utilização e personalização do DiversityAgent.

Palavras-chave: Diversidade, Tolerância a Intrusões, Gestão de Recursos, Cloud Computing.

Abstract

Obtaining correct results and behaviour on computing is a long-standing concern. Such guarantee can be obtained through fault and intrusion tolerance mechanisms, which aim to tolerate crash and arbitrary faults. Byzantine fault tolerant replication, when combined with proactive recovery techniques can tolerate any number of arbitrary faults during entire system life time. However, common vulnerabilities shared between replicas can compromise such tolerance, rendering diversity as a complementary mechanism.

Diversity is a mechanism that consists in providing and combining diverse resources to increase vulnerability independence between system components. Obtaining diversity automatically is a process that can be decomposed into two phases: creation and selection. The first phase consists in providing enough diverse resources to be considered, combined and selected in second phase.

In this thesis we present the DiversityAgent, a Java library for selecting cloud resources considering multiple diversity properties. Its clients only need to register available resources, then the DiversityAgent assumes the responsibility of selecting appropriate cloud computing resource combination for each server deployment. In order to design the DiversityAgent, we review taxonomies for diversity on computer systems and analyse several diversity group properties supported by cloud providers or tools, and opportunities for cloud computing players contribute with diversity management area.

This document contains a review on basic fault and intrusion tolerance mechanisms, followed by an extensive diversity analysis in cloud computing environments and by the DiversityAgent development. We also present an integration of our component with two use cases foreseen by CloudFIT project, as well as present the results of correctness and performance evaluations. At the end there are the final remarks about this work and possible future work, besides three appendices regarding DiversityAgent public interfaces, usage and customising tutorials.

Keywords: Diversity, Intrusion Tolerance, Resource Management, Cloud Computing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Obtaining correct results and behaviour on computing is a long-standing concern. The following excerpt regarding calculators advent is from 1834 and already illustrates the importance of using mechanisms to tolerate and identify errors [24]:

> *"The most certain and effectual check upon errors that arise in the process of computation, is to cause the same computations to be made by separate and independent computers; and this check is rendered still more decisive if they make their computations by different methods."*

There are two mechanisms that arise from this statement as important approaches for correct computations. The first one is *replication*, which consists in calculating results more than once and comparing them or voting at the end. The second one is *diversity*, which consists in using different methods or components in each calculation. Both approaches integrate the group of mechanisms for fault and intrusion tolerance (FIT) nowadays, which are able to tolerate crash and arbitrary faults on computer systems.

In practical terms, a replicated service can tolerate crash faults if there is, in its group of replicas, at least one server still able to answer client requests. The same replicated service can tolerate arbitrary faults, normally, if the majority of replicas answer correctly each client request.

If an attacker discovers an exploitable vulnerability in a server and the same vulnerability also exists in other service replicas, then the system fault and intrusion tolerance can be compromised. Common vulnerabilities, shared between replicas, are a known limitation of replication mechanisms. Increasing vulnerability and bugs independence between replicas is the main goal of diversity.

Diversity is a mechanism that consists in providing and combining diverse resources among system components. Obtaining it automatically is a process that can be decomposed into two phases: creation and selection. The first phase consists in providing enough diverse resources to be considered, combined and selected in the second phase. Automatic diversity obtention during the resource selection phase is our main objective in this thesis.

Managing large amounts of computing resources is a complex task that can be facilitated by automated tools for resources allocation and deployment. Nowadays, thinking in deploying large distributed systems implicitly means to consider cloud computing tools as one of management possibilities. Cloud computing model, in its simplest definition, is a model to delivery computing as an utility service [20], but technically its players are infinite sources of computational resources, automatically managed and publicly provided. In this work, we consider cloud computing the scenario to achieve our main goal.

Imagine that the owner of a replicated service is a client of a given cloud provider, and all service replicas are deployed in this cloud. If a crash or arbitrary fault causes the disruption of this cloud, the entire replicated service can fail, which means that there are no vulnerability independence between replicas regarding cloud provider. In this case, the single point of failure is the cloud provider, which leads to indicate diversity of cloud provider as a possible solution.

Creating some accounts in other providers is the first step to obtain diversity of cloud providers. The second phase consists in selecting for each replica deployment one of the unused available cloud providers. However, manually choosing a cloud provider for each deployment can be an inconvenient solution, therefore an automatic selection tool is a must either for this diversity of provider or any other diversity.

In this thesis, we present the DiversityAgent, a Java library for selecting cloud resources considering multiple diversity properties. Its users only need to register available resources, that the DiversityAgent takes the responsibility of selecting an appropriated resource combination for each server deployment. Our solution focuses in providing what we believe to be the first automatic software library for this purpose, in the light of correctness, extensibility, effectiveness and other requirements.

In addition, we provide a broad diversity analysis in the cloud computing scenario. Such study is composed by a taxonomy review and a discussion on each diversity classification, pointing properties supported by cloud providers, and opportunities for cloud computing players to contribute with diversity management area through more resource specification. We also present an integration of our component with two use cases foreseen in CloudFIT,[1] the project in which this thesis was developed, as well as the results of DiversityAgent correctness and performance evaluations. At the end, there are three appendices regarding DiversityAgent public interfaces, usage and customising tutorials.

1.1 Objectives

Our main goal is to automatically obtain diversity in resource selection, considering cloud computing our scenario. In order to achieve such objective we defined seven specific tasks, namely:

[1] Available at http://cloudfit.di.fc.ul.pt/. Accessed on March 19, 2012.

- Classifying the different and relevant types of diversities.
- Analysing the current state of the art of and opportunities for diversity in cloud computing environments.
- Defining desired functionalities for an automatic diversity management component.
- Designing and implementing such component.
- Integrating the component in a system that uses replication and proactive techniques.
- Evaluating the component correctness and performance.
- Writing the component documentation.

1.2 Contributions

Our scientific contribution can be divided basically into two main points. The first is to provide an extensive diversity analysis on cloud computing environments, based on a well defined taxonomy. The second is to develop and publish what we believe to be the first software component that allows the automatic obtention of diversity in any component during cloud computing resource selection process.

1.3 Document structure

In face of the expressed objectives, the present work has six chapters, from which this first is a brief introduction to the subsequent topics. Additionally, the context in which this thesis appears and its related work are presented on Chapter 2. Chapter 3 contains an analysis on diversity, including the taxonomy that will be used by the following of this work and opportunities for cloud computing players regarding diversity management. The requirements, design and implementation of a component for automatic diversity obtention are described on Chapter 4. We decided to divide the requirement analysis into three categories: functional, non-functional and architectural analysis. The implementation description is divided into basic internal components, diversity implementations and drivers to cloud interfaces.

Chapter 5 is dedicated to describe use case scenarios proposed by CloudFIT, a research project that funded this thesis, as well as to present how DiversityAgent is integrated in the project architecture, then for any other infrastructure. At the same chapter, an evaluation of component correctness is presented and performance aspects are analysed. Chapter 6 contains the conclusions obtained with this thesis and some opportunities for future work.

Finally, some further technical information is addressed on appendices. Appendix A is dedicated to present DiversityAgent public interfaces, while Appendix B is a tutorial on using such interfaces. Appendix C presents a tutorial on customising DiversityAgent.

Chapter 2

Context and related work

This chapter is dedicated to present basic concepts regarding fault and intrusion tolerance mechanisms and other works related with automatic obtention of diversity. It is intended also to address this thesis context with cloud computing and CloudFIT project, besides to yield a background knowledge about topics that will be discussed on next chapters.

Before presenting fault and intrusion tolerance concepts it is important to introduce the security field. Software security is a computer system area focused on extracting vulnerabilities from software and protecting systems to prevent attacks to become intrusions [31]. Based just on this definition, a system is considered as secure only if intrusions never happen on it.

2.1 Fault and intrusion tolerance

Fault tolerant system is one that tolerates a certain amount of faults without disrupting the delivery of a correct service [1]. But normally only crash faults are considered in fault tolerance .

Intrusion tolerance arises on intersection of fault tolerance and security. An intrusion tolerant system is one that tolerates a certain amount of intrusions without disrupting the delivery of a correct service, neither compromising its security properties (availability, confidentiality, integrity, etc) [31]. Therefore, this fault model includes malicious faults, which are also known as arbitrary or Byzantine faults.

In the simplest comparison, security tries to avoid intrusions while intrusion tolerance tries to reduce intrusions impact. The main differences between fault and intrusion tolerance are presented on Figure 2.1 [31], where the main focus is the sequence of facts that lead to failures in each case. We consider intrusion tolerance our main fault model for DiversityAgent, but it can also be used for fault tolerance in other contexts.

State machine replication [29] is one important mechanism to achieve Byzantine fault tolerance with high throughput [8, 23]. Basically, this mechanism consists in a replicated service developed over a deterministic state machine approach. A service is replicated

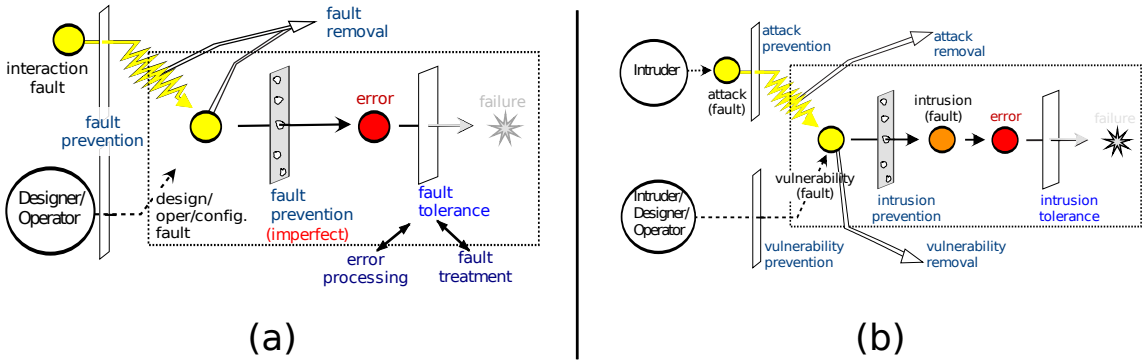

Figure 2.1: Fault-error-failure sequence (a) and the AVI composite fault model (b).

and replicas start from the same initial state upon processing each client request, each replica arrives to the same states and provides the same results. Crash faults are easily tolerated with this mechanism, but arbitrary faults are tolerated through replicas results by voting.

There is a minimal number of replicas needed by BFT replication protocols to tolerate f Byzantine faults. This amount of resources will depend on which protocol is being used, but it can vary from $3f + 1$ [8, 23], to $2f + 1$ [12, 33], or even to $f + 1$ [15, 34]. It is possible to specify that a replicated intrusion tolerant system is a replicated system in which a malicious adversary needs to compromise more than f out of n components during the entire system life time [4].

We introduced a new property at the end of previous paragraph: the time. Allowing the entire system life time to be the window time for attacks can lead to a problem: after f Byzantine faults happen in a service, intrusion tolerance can become compromised. To reduce the size of window, we use proactive recovery [7], which means that we replace each replica from time to time, to recover its initial and correct state.

There is a stateless or just fault tolerant version of proactive recovery, which is based on service redundancy and in this work is called proactive replacement. A proactive replacement means that we replace each redundant server from time to time, where a server is rejuvenated and software aging issues can be avoided.

Both proactive techniques reduce window time available for an attack to happen, but maintain the same limited number of tolerated intrusions on each proactive period. Furthermore, they remove the existence of faults and intrusions on a recovered server. However, if we just rejuvenate the components, vulnerabilities that caused such intrusions remain. Another possible scenario is an adversary acquiring enough knowledge to rapidly compromise more than f recovered servers in less than T time units. With this in mind, we devise a mechanism intended to make an attacker's life even harder, the diversity [30].

Diversity is a fault and intrusion tolerance mechanism that consists in providing and selecting different software or hardware compositions to improve vulnerability indepen-

dence between service instances. It can improve the probability of a vulnerability that caused an intrusion in one server, does not exist in other servers. In other words, an attacker needs to discover more than f different exploits to attack an entire service, and when combining diversity with proactive mechanisms, the attacker has a limited time to perform the entire attack. Common vulnerabilities are one of the main problems of intrusion tolerant systems, thus an extensive usage of diversity is needed [4]. Furthermore, even having good evidences on effectiveness of diversity for operating systems [19] and database management systems [17], it is a half-solved problem on intrusion tolerance scenario, at the same time that diversity management is still an open problem [4].

Regarding our first objective with this work, a diversity analysis, there are two taxonomies for diversity on computer systems[14, 28]. We review their classifications and adapted the scenario of each diversity groups to cloud computing. We also pointed which are the aspects that already are supported by cloud providers, which still remain unsolved and which are the next steps to consolidate diversity through cloud facilities.

Concerning our second objective, an automatic diversity management component, we found basically two automatic diversity management proposals. Regarding operating system diversity, an algorithm proposed by Henriques [18] uses a database table that contains all parameters needed to choose consistently an OS for a requested VM. It is meant to achieve the best combination of all active servers, regarding the highest level of vulnerability and bugs independence through OS diversity. His algorithm considers only diversity of operating system, while DiversityAgent is a library that can be extended to provide diversity algorithms on any diversity group or component. Another difference is regarding the resource selection scenario, where, even both provisioning virtual resource selection, we consider cloud computing environments and he only considers virtual machine monitor (VMM) environments.

N-variant systems [13] is the framework name proposed by Cox et. al., which uses automated diversity to provide high assurance detection and disruption for large classes of attacks. They basically execute a set of automatically diversified variants on the same inputs, and monitor their behaviour to detect divergences. Their algorithm considers only diversity at application level, while once more, DiversityAgent is a library that can be extended to provide diversity algorithms regarding any component. We do not provide behaviour monitoring features, because our focus is only in composing diverse resource combination for service replicas.

2.2 Cloud computing

Cloud computing in its simplest definition is a model to deliver computing as a service [20], and it is the scenario where DiversityAgent will accomplish its goal of automatic obtaining diversity. It is divided into three main service models [27], namely:

- *Software as a service (SaaS):* Refers to the provisioning of any application as a service, rather than a product, running on cloud environments. At this abstraction level, an application can be accessible from various client devices, but clients normally cannot explicitly manage or control underlying cloud infrastructure.

- *Plaftorm as a service (PaaS):* Refers to the provisioning of a computing platform or software components as a service to assist the development and execution of cloud applications. At this abstraction level, clients are software service providers and do not explicitly manage or control underlying cloud infrastructure, but they have control over deployed applications and some hosting configurations.

- *Infrastructure as a service (IaaS):* Refers to the provisioning of computing resources as a service for cloud platforms and applications. At this abstraction level, IaaS customers do not explicitly manage or control underlying physical infrastructure, but they have control over their virtual infrastructure (virtual processing, storage and network resources) and software layers (operating systems and applications).

Infrastructure as a service is the only model where clients can control the resources on a virtual infrastructure. The diversity can be obtained through virtual resource selection. Our positioning is as IaaS clients, where DiversityAgent will prepare diversity requirements to request virtual resource allocation. The DiversityAgent component can be used by PaaS and SaaS providers. We first analyse which diversities can be obtained by IaaS customers and second we provide a component that obtains diversity automatically from IaaS providers on processing virtual resources selection.

Cloud of clouds emerge as an aggregation structure to federate several independent IaaS providers. It keeps all advantages that such federation offers, for example, vendor lock-in prevention. Nowadays, there are cloud brokers that already provide support for resource selection in more than on IaaS provider, but they do not provide any kind of automatic diversity. Broker clients have to specify which IaaS should be used to allocate resources for each service instance. Some examples of cloud brokers are CloudKick, Rightscale, 3tera, Elastra, and Kaavo.

We assume that we should verify the existence of desired properties on cloud tools and providers, since we did not found cloud brokerage tools for automatic selecting diverse resources. DiversityAgent should consider the automatic diversity obtention on all diversity groups in a generic and extensible manner.

2.3 CloudFIT project

This thesis was developed within a research project called CloudFIT, Fault-and-Intrusion Tolerance for Clouds at the University of Lisbon. It is a two-year research project funded

by the national research funding agency (FCT) to define an infrastructure for intrusion tolerant services in a cloud environment.

This section briefly introduces CloudFIT and its system, called FITCH, Fault and Intrusion Tolerant Cloud Computing Hardpan, because it is the structure where DiversityAgent will be integrated. All FITCH components and their relationship with this thesis will be presented on Chapter 5. FITCH was conceived with several components, combining Byzantine Fault Tolerant (BFT) replicated services and hardware virtualization, tolerating intrusions in a subset of replicas of a service, and implementing proactive recovery using replica replacement with diversity. The DiversityAgent will be the component responsible to obtain diversity automatically when selecting resources for proactive requests.

Regarding expected results from the above-mentioned project there are three directly related goals with this thesis:

- The requirement analysis on resource management for intrusion tolerance, such as replica dislocation and diversity;
- The extension of a cloud resource allocation tool in order to incorporate FIT requirements;
- A prototype that integrates a virtualization architecture with the extension of a resource allocation tool. An performance evaluation of this prototype. And an improvements analysis that the proposed architecture yields in terms of intrusion tolerance.

Diversity requirement analysis is addressed on Chapter 5, where the relationship of all diversity groups with cloud computing resource selection is presented . OpenNebula is the cloud resource management tool chosen to be extended by CloudFIT, since it is free and open source software, it was already available when the project started. Furthermore, it was the only tool with matchmaking scheduling policies, which allows for filtering and ranking resources based on requirements. DiversityAgent uses a matchmaking approach in its cloud driver for OpenNebula, and it is presented on Chapter 4. Finally, a brief overview of FITCH prototype integrated with DiversityAgent is presented on Chapter 5.

Chapter 3

Diversity analysis

This chapter presents a taxonomy of various diversity groups that will be used in the remaining ones. In addition, we analyse all diversity groups and correlate each one of them with automatic resource selection through the usage of cloud computing facilities.

3.1 Taxonomy

The first taxonomy created for diversity on computer systems differentiates the level in which diversity is offered. It is divided into the following groups [14]: at level of users or operators, at human-computer interfaces, at application software, at execution level, and finally at hardware or operating system level.

A newer taxonomy was proposed by Obelheiro et al., which is defined in terms of the component in which diversity can exist or be created [28]. It is intended to clearly identify where and how diversity can be obtained, and it is divided into the following groups:

- *Application.* Using diverse implementations for the same software specification.
- *Administrative.* Executing applications in diverse administrative entities.
- *Location.* Using diverse geographic locations to execute an application.
- *Commercial off-the-shelf software.* Using diverse commercial products for the same computing task. It is composed of five subcategories: Database management systems, middleware, virtual machines for bytecode, compilers and libraries.
- *Operating system.* Using different operating systems to execute an application.
- *Security method.* Enforcing security properties through diverse security methods.
- *Hardware.* Executing applications on diverse machines with different physical hardware.

In addition, each diversity can be decomposed into two properties:

- *Axis of diversity:* contains the components where diversity can be introduced (example: operating system);

- *Degree of diversity:* quantifies the possibilities regarding one axis (example: 3, if we consider Windows, GNU/Linux Ubuntu and Solaris);

Regarding the taxonomy to be followed by this work, our approach is similar to Obelheiro's taxonomy, but with some minor modifications, in order to simplify the classification of components. We are not concerned about specific components or software, therefore we merged some groups. The resultant taxonomy is divided into the following groups: *application*, *administrative*, *location*, *support software*, *hardware* and *security*.

The range of components in which diversity can exist or can be created should not be limited. Thus, we propose the usage of *folksonomy*[1] when defining the component from which will be obtained the diversity, provided that it must belong to one of the groups presented in our taxonomy. For example, if one wishes to obtain diversity through different web server implementations, he may define the diversity name as "Web server" and define it to belong to the taxonomy group of "Support software". The diversity groups will be presented in following sections, as well as their relation with automatic resource selection will be discussed.

3.2 Application diversity

The first diversity group to be addressed is the application diversity, which consists basically in *using more than one implementation of the same software specification*. The main goal with this diversity is to increase the probability of creating software whose vulnerabilities (if exist) are completely independent, which means that they are not shared between different implementations.

This idea arose as redundant programming in 1975 [2], and lately it was proposed as N-Version programming in 1977 [3]. The N-Version programming considers N programming teams developing N different application implementations. This methodology contributes to vulnerability independence in some systems [25], where it increases the probability of creating completely independent application versions, but it cannot guarantee vulnerability independence in other cases [21]. With this in mind, caution is appropriated, where controlled experimentation in a realistic environment can be important to define if N-Version programming is useful for the application in question [22].

Registering resources in cloud computing requires some information that are used to describe the respective resources. Such information is normally called metadata, and can also be collected when a new resource is registered. For example, after registering a new

[1] *Folksonomy.* A classification method based on tags created and managed collaboratively by regular people instead of experts.

virtual machine image, a manager software can retrieve some extra information as image size and file type. When metadata is not supported by cloud providers, it is possible to circumvent such limitation through generic tags (if supported) or even composing the resource name by more than one metadata. The application version is mandatory to provide N-Version application diversity on resource selection, but it is not supported on most of cloud providers. The current method used to differ application versions and other VM image metadata is to compose its name with all information needed.

Another existent approach for automated creation of application diversity are transformation techniques, for example, rearranging memory, randomising system calls, instruction set, protocol parameters, among others. In cloud computing, automatic selection of resources using this diversity approach can be achieved similarly to previous one, but for each variant, there is the need of providing metadata regarding transformation methods on VM images. Another possibility for resource selection is using automatic transformation after the deployment through scripts, but this approach is out of this work's scope.

3.3 Administrative diversity

The second diversity group to be analysed is the administrative diversity, which consists in *using more than one administrative entity to run a service or store data*. The fault and intrusion independence provided by this diversity is the prevention of an entire service or data set being affected by any local administrative event.

One of the hottest and most recent examples where this diversity plays an important role is vendor lock-in, which consists in making customers dependent on a specific provider, and where changing it requires substantial costs [5]. There are some possible initiatives to avoid or to reduce costs in vendor lock-in, which encompasses a modular development of drivers for diferent providers, usage of open interfaces, or usage of more than one provider since the beginning. But all these possibilities are cloud-of-clouds scenarios, where cloud providers are completely unreliable, which means that cloud clients are responsible to obtain all diversity of cloud providers.

We developed in this thesis the DiversityAgent, a component that allow cloud clients to automatically obtain diversity at this level, which will be presented on Chapter 4. With this component, administrative diversity (as all other groups) can be naturally provided. It allows to developers of cloud applications to consider, since the beginning, the usage of more than one cloud provider through different cloud drivers implemented on this component, or even using more than one cloud with the same driver already implemented.

Internal events on specific providers is another example where administrative diversity is important for intrusion tolerance. Electrical disturbances and accidents are some of the most expressive ones. Spikes and surges normally cause disturbances if redundant power management are not provided (for example in private clouds). To avoid these disruptions,

a component like DiversityAgent can solve the problem if it considers the usage of more than one cloud provider. This component should consider information regarding physical hosts, racks and clusters during resource selection, for similar reasons.

The reliability of a cloud provider and its tools is the third scenario. Cloud clients could use any form of reliability metrics to decide on resource selection. For example, one could use the amount of failures and uptime as properties to be considered. These values allow to define some traditional fault tolerance metrics like failure rates, mean time to failure (MTTF), mean time between failures (MTBF), and others.

Resource selection regarding this diversity group can provide an improvement on globally distributed services performance, because it can be used to select cloud providers conveniently located in order to reduce network latencies. When using service replacement protocols as in FITCH, performance impact is relevant, either in service level degradation or approximation with end user cases. Considering network latency, round trip time (RTT) or number of hops can be relevant for resource selection, once through comparison between cloud providers could be possible to verify which one is the best option in relation to the mentioned performance metrics.

3.4 Location diversity

The next group is location diversity, which consists in *using geographically distributed resources to run a service or to store data*. The fault and intrusion independence provided by this diversity is the prevention of an entire service or data set being affected by any geographically local event.

The most used examples to express the importance of this diversity are natural disasters, amongst which the earthquake that hit Japan on March, 2011 was one of the most devastating and recent. Asia Cloud Forum[1] published a series of posts about services and resource disruptions caused by this event, where through large companies announcements (like Amazon, Google, Microsoft, Verizon and NTT Com), it is possible to verify the importance of this class of diversity.

Political and legal events are equally important scenarios where diversity of location plays an important role, even if it has been less discussed. Unstable governments, diplomatic positioning and personal data or copyrights prosecutions are some specific examples of scenarios where this diversity can be helpful. An advantage when using location diversity in these scenarios is that any local political or legal event cannot lock in that region an entire service or data set. One way to provide political independent (or almost independent) cloud provisioning would be data centres located in international areas. To exemplify this possibility, Google Inc. proposed a water-based solution for data centres

[1] Available at http://www.asiacloudforum.com/tag/Japan%20earthquake%202011. Accessed on March 19, 2012.

(and registered a patent with this idea), which can be placed in any international Ocean portion [9].

The last scenario, where this diversity can be used, is regarding performance aspects. There is a large number of services that can be deliberately distributed on different specific locations to approximate service and end users, normally reducing network latency between them, similarly to what is presented in Section 3.3. Location diversity can be easily achieved through cloud computing, even with a single public cloud provider. For example, Amazon has data centres in United States, Europe, South America and Asia Pacific.

Considering geographic location of physical resources is the main way to provide location diversity. Currently, geographic location of cloud providers are managed by clients manually or through services that matches IP addresses with their location, but they are unofficial and probably imprecise results. In order to achieve more precision, cloud providers could supply their location with meaningful information, for example, physical coordinates or other geopolitical metadata like city, state, region, country, continent, economic group, political union, among others.

Physical coordinates can be very useful to calculate, in allocation time, the smallest or the highest distance between any two specific points in the world. With this information, it would be possible to choose the farther away resource from some place where happened some recent natural disaster or, in opposite case, the nearest resource from some end user.

Additionally, there is a novel Byzantine fault-tolerant (BFT) protocol called EBAWA [32], which focuses on Wide Area Networks (WANs), instead of Local Area Network (LANs). It requires fewer communication steps, fewer replicas and has better throughput and latency that others in literature (that are mainly focused on LANs). Location diversity on resource selection can improve this protocol considering that it can be possible to correlate geographic location of client requests and service replicas geographic location, in order to always approximate service replicas and regions where there are significant amounts of clients requests.

3.5 Support software diversity

The fourth diversity group to be analysed is support software, which consists in *using diverse versions and implementations of any software that can provide a basis for service applications*. This basis can be composed by many software layers and components, ranging from operating systems to commercial off-the-shelf software, middleware, libraries, compilers, among others. The fault and intrusion independence focused by this diversity is the prevention of an entire service from being affected by any common software vulnerability shared between them.

Measuring the independence level on this diversity group is complex because it has to

consider all known vulnerabilities of each component and correlate them. Even after this correlation, there is the unknown vulnerability problem, which can be exploited through zero-day attacks and are impossible to measure among different components. Our objective with this analysis is not to discuss how much independent are software vulnerabilities, but presenting who already contributed to this area and how this group of diversity can be used on cloud computing resources selection.

Operating system is one of the main (and largest) components that can be used by cloud applications as a support software. There are some studies that analysed OS vulnerabilities and bugs (see *Related Work* chapter on [18]). We believe that their judgement in addition to conclusions from [19] provide good evidences that OS diversity has acceptable degrees of vulnerability independence.

All other components of support software are dependent from the service or application scope, but some of them already were studied to verify their vulnerability independence level. Some examples of software component that normally are considered as support software are Database Management Systems, middleware, web servers, FTP servers, SSH servers, compilers, libraries, etc. Regarding databases, Gashi et. al. [17] presented that there are good evidences that diverse redundancy using this components has acceptable vulnerability independence as well.

One drawback of this diversity is a high cost, in human resources and time, to design and prepare multiple combinations of components. A VM image has to be created and registered by an administrator for each combination to be provided when deploying the application. We believe that, in a near future, there will be good solutions that may provide automatic composition of VM images in execution and allocation time. One example of this kind of project is the OSFarm [6], which contains a service that aims to provide VM images generated on demand.

We propose a large set of metadata that could be supported by cloud providers regarding registered VM images. But nowadays most of them just maintain a minimal set of information like image name, path and owner. To exemplify, information like OS type (GNU/Linux, BSD, MacOS, Microsoft Windows, Solaris, etc), OS name (Ubuntu 11.10, Windows 7, Solaris 11), virtualization type (para or full-virtualized), architecture (i386, i686, x64, SPARC, etc), application and services installed, supported programming languages, among others could be registered for each VM image. All this range of information could provide a good granularity for selecting resources using diversity.

Another important support software on cloud computing that could be diversified is the hypervisor. It is possible to request for each server to be deployed on a different hypervisor if cloud providers offer more than one option, and if there is metadata associated with the images to express which are the hypervisors that can deploy it.

3.6 Hardware diversity

The fifth diversity to be considered in this analysis is the hardware group, which consists in *using physical hosts with different hardware components to allocate the service instances*. The fault and intrusion independence provided by this diversity is the prevention of an entire service or data set being affected by one common hardware vulnerability shared between more than one physical host that are running the virtual machines.

From all components, processors are one of the main targets, and one example of shared bug was the F00F Pentium bug [11], where an execution of one specific instruction was not handled by the exception handler, causing blockage of interrupts handling.

Cloud resource selection can provide hardware diversity only when this information is available. For example, some properties that could be considered are CPU model, architecture and speed. Not just limited to CPU metrics, information about all hardware components could be supported by cloud providers to select different models of, for example, network or video cards, hard disks and Trusted Platform Modules (TPM). Trust Platform Modules are important security components that provide secure generation and storage of cryptographic keys and and offer functionalities for remote attestation of system components.

3.7 Security diversity

The last diversity group to be analysed is the security diversity, which is equally important and consists in *using more than one security method or more than one security policy within a group of cloud providers and replicated service instances*. The fault and intrusion independence provided by this diversity is the prevention of an entire service or data set being affected by one common security vulnerability or security flaw.

The first scenario is the authentication security mechanism, where there is the assurance that a cloud client is the one it claims to be. Considering that each provider can support a specific authentication method, it allows the selection of cloud provider based on different authentication methods in a cloud-of-clouds scenario. All other security methods can be equally used as properties on resource selection using security diversity, ranging from access control, to data integrity and confidentiality.

In addition to security mechanisms, the security policy of a cloud provider can be equally important when selecting resources through security diversity. Such security policies can include different physical access and control methods, which can provide diversity on protection against service disruptions caused by physical attacks to data centres of some specific cloud provider. Choosing different cloud providers means that they can have different security policies. Cloud providers could publish which security policy they

follow (for example, the ISO 27001)[1] , so clients can choose different security policies.

3.8 General considerations

In this section we provided some general comments regarding our diversity analysis and present a summary of all properties identified on this chapter. Our first general consideration is dedicated to reinforce that we did not aim to discuss diversities effectiveness, but we aimed to discuss their meaning, scenarios and identify properties that cloud providers should pay attention for. Cloud providers may not agree to inform all proposed properties, once there are commercial risks and extra costs in publishing and maintaining all information addressed on this thesis, but we consider such discussion an important step on diversity management area.

Our second consideration is that our analysis does not focus on economic constraints applied on diversity obtention. In addition, we consider that any of diversities presented on this analysis can be combined with others. Diversities can also form hierarchical relationships, which means that a diversity with lower hierarchy level depends on resources previously chosen by a diversity algorithm with a superior hierarchical level. Choosing a certain type of component from one diversity group may eliminate the diversity once available in another group. For example, by choosing cloud provider one has to cope with its hardware or locations. The hierarchical order is extremely important when combining diversities, and should be described always from the most to the less important. We explain how this relationship works and should be addressed by our component on next chapter.

Finally, on Table 3.1 we present a summary of properties discussed on this chapter, their relationship with cloud computing resources and if they already are supported by OpenNebula (ONE) tool and by Amazon Web Services (AWS) cloud provider as examples of their existence in cloud environments.

[1] Available at http://www.17799.com/. Accessed on March 19, 2012.

Diversity group	Property description	Cloud resource	ONE	AWS
Application	Image name	VM images	Yes	Yes
	Application name	VM images	No	No
	Application version	VM images	No	No
	Transformation method	VM images	No	No
Administrative	Cloud provider name	Cloud provider	No	No
	Available APIs	Cloud providers	No	No
	Physical host name	Physical hosts	Yes	No
	Rack name	Physical hosts	No	No
	Cluster name	Physical hosts	Yes	No
	Number of failures	Cloud providers	No	No
	Number of power outages	Physical hosts	No	No
	Number of VM failures	Physical hosts	No	No
	Cloud uptime	Cloud providers	No	No
	Host uptime	Physical hosts	No	No
	Autonomous system (AS)	Cloud providers	No	Yes
	Network latency to X	Cloud providers	No	No
	Round trip time to X	Cloud providers	No	No
	Number of hops to X	Cloud providers	No	No
Location	GPS coordinates	Cloud providers	No	No
	Location based on IP	Cloud providers	No	No
	City	Cloud providers	No	No
	State	Cloud providers	No	No
	Region	Cloud providers	No	Yes
	Country	Cloud providers	No	No
	Continent	Cloud providers	No	No
	Economic group	Cloud providers	No	No
	Political union	Cloud providers	No	No
	Geographic distance to X	Cloud providers	No	No
Support software	Image name	VM images	Yes	Yes
	OS type	VM images	No	Yes
	OS name	VM images	No	No
	OS architecture	VM images	No	Yes
	Kernel	VM images	No	Yes
	Virtualization type	VM images	No	Yes
	Application and service	VM images	No	No
	Supported programming languages	VM images	No	No
	Compatible hypervisors	VM images	No	Yes
Hardware	CPU model	Physical hosts	Yes	No
	CPU architecture	Physical hosts	Yes	No
	CPU Speed	Physical Hosts	Yes	No
	Network card model	Physical Hosts	No	No
	Network card speed	Physical hosts	No	No
	Video card model	Physical hosts	No	No
	Hard disk model	Physical hosts	No	No
	Hard disk speed	Physical Hosts	No	No
	Hypervisor	Physical hosts	Yes	No
Security	Cloud provider name	Cloud providers	No	No
	Authentication methods	Cloud providers	No	Yes
	Access control methods	Cloud providers	No	Yes
	Data integrity methods	Cloud providers	No	Yes
	Data confidentiality methods	Cloud providers	No	Yes
	Security policies	Cloud providers	No	No

Table 3.1: Proposed properties for obtaining diversity on resource selection.

Chapter 4

The DiversityAgent

DiversityAgent is a Java library that allows IaaS clients to automatically obtain diversity on cloud resource selection. In this chapter, we present the considered requirements for this component, its design and development. A tutorial on how to use DiversityAgent is available in Appendix B, as well as two use cases are presented in Chapter 5.

4.1 Requirement analysis

4.1.1 Functional analysis

The main challenge for DiversityAgent is the automatic selection of resources considering any diversity group. The first functional requirement, *the automatic selection of resources considering the obtention of any diversity*, could be solved by a generic algorithm with an extensible structure that might allow DiversityAgent users to create and provide their own diversity algorithms. A generic property set could be used in order to receive contributions from each diversity algorithm. Then, it could be sent to cloud providers, which should select and allocate the appropriated resources.

The second functional requirement is the *dynamic interaction with DiversityAgent*, which means that users might add or remove clouds, images, diversities and virtual machines at any time. To solve this requirement, CRUD pattern [26] could be adopted. In doing so, users could dynamically modify service instances and available resources at execution time, without disrupting their service provisioning.

Regarding *information management* aspects, DiversityAgent must maintain only a minimal amount of mandatory information for automatic resource selection. This information should be related with the client service, including regarding cloud providers, VM images, diversities and running service instances. It should not create attributes that do not exist in cloud providers. Just to complement this idea, DiversityAgent might also differ from centralised meta-schedulers (or brokers), which are normally time triggered components that from time to time fetch information from all cloud providers to maintain locally the maximum of global information and to take decisions alone.

In order to achieve *hierarchy between diversities*, DiversityAgent could orchestrate diversity algorithms considering different hierarchical levels. It might allow users to register diversity algorithms, from the most to the less important in the hierarchical sequence of diversity algorithms. Each algorithm must receive the contributions from diversities registered before itself and should check if its properties were not already defined. An example of hierarchical relation between diversities is the case of diversity of cloud provider and physical host. If an user registers the diversity of cloud provider first, then DiversityAgent must request the contribution from this diversity before the contribution from diversity of physical host

4.1.2 Non-functional analysis

Correctness. A correct diversity algorithm is the one that obeys a minimal and a maximal level of diversity obtention. Regarding the minimal level, DiversityAgent would provide diversity selection only if it is currently available, which means that if users do not provide enough diverse resources, DiversityAgent cannot guarantee selecting diverse resources. Regarding the maximal, DiversityAgent must provide always the highest level of diversity as possible, which means accomplishing all possible and diverse combinations before start repeating resource combinations.

Extensibility. DiversityAgent could provide abstract classes to be extended by diversity algorithms and cloud drivers. Another approach that might contribute to this solution is the factory design pattern [16], which correlates tags with the respective class instantiation. This way, in order to users extend DiversityAgent, they could modify just some specific classes and methods, without modifying DiversityAgent's core algorithms. A tutorial on how to customise DiversityAgent is presented in Appendix C.

Scalability. DiversityAgent should maintain just the minimal amount of information necessary regarding client software, which might allow users to deploy large and distributed services without system degradation caused by scalability issues. Furthermore, diversity algorithms should have good or optimal complexity time, in order to remain scalable face to large amounts of available resources.

Maintainability. In order to achieve the highest level of maintainability as possible, our approach must consider simplicity, modularity, flexibility and code documentation.

Security. DiversityAgent must maintain exactly the same authentication methods and credentials provided by cloud providers. Providing our component as a library might be another factor that contributes to this approach, because DiversityAgent should not provide a secure key storage service.

Recovery. DiversityAgent should provide public interfaces to save and recover information about the current resources allocated to client services.

Configuration management. The component should support configuration management for dynamic adaptation, for example, upgrades and downgrades regarding the type

of service instances in execution time. CRUD interfaces could be provided regarding all resources presented on functional requirements, as well as an interface for clients inform predefined properties, for example, the amount of memory and CPU to be allocated for each service instance.

Documentation. All DiversityAgent classes should be documented using JavaDoc specification and tool, allowing users to see internal components properties and methods within an web browser. Furthermore, all tutorials presented in this work appendices must be published on DiversityAgent site [10].

Licensing. Our approach must be focused on free and open source code, where any user can improve the implementation with their own algorithms and drivers. We could provide DiversityAgent under GNU Lesser General Public License (LGPL), which allows using this library even with proprietary software. In addition to the customisation tutorial, there are some notes on publishing users contributions to DiveristyAgent in Appendix C.

4.1.3 Architectural analysis

The architectural analysis consists in presenting DiversityAgent positioning as a component to be integrated in client software architecture. A resource manager normally is responsible for creating, maintaining and removing service instances from the group of active servers. Such maintenance tasks are requests from resource managers to cloud providers in order to create, migrate or delete virtual machines, which can be done through multiple interfaces supported by each provider. DiversityAgent must be a library component that could be instantiated by resource managers to become responsible for preparing such requests and maintaining information regarding the current service instances. Further than this, DiversityAgent must select automatically a diverse resource combination for each new instance deployment. A possible DiversityAgent positioning in relation to client software and cloud providers can be seen in diagram *b* of Figure 4.1. In next sections, the internal components of DiversityAgent are presented and explained.

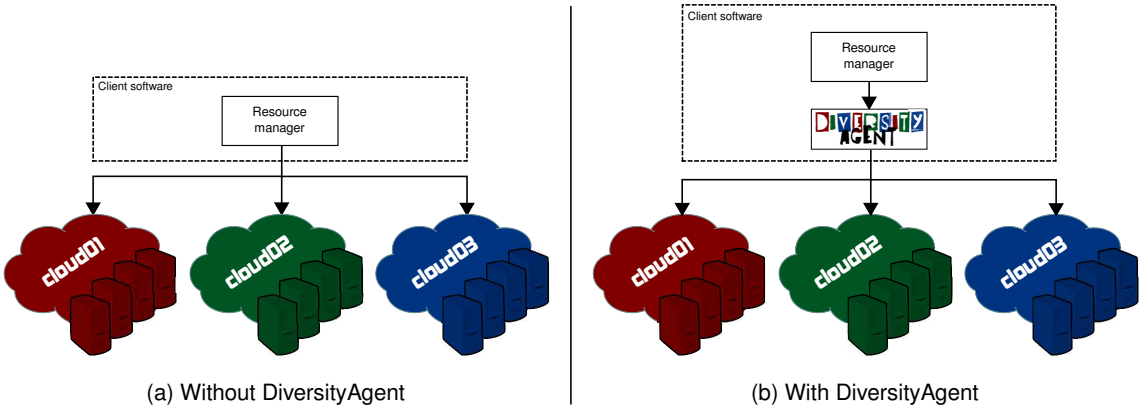

Figure 4.1: DiversityAgent positioning.

4.2 Implementation

In the following descriptions, all components existent on Figure 4.2 will be presented, as well as their importance and contributions to DiversityAgent, a Java library for selecting cloud resources considering multiple diversity properties.

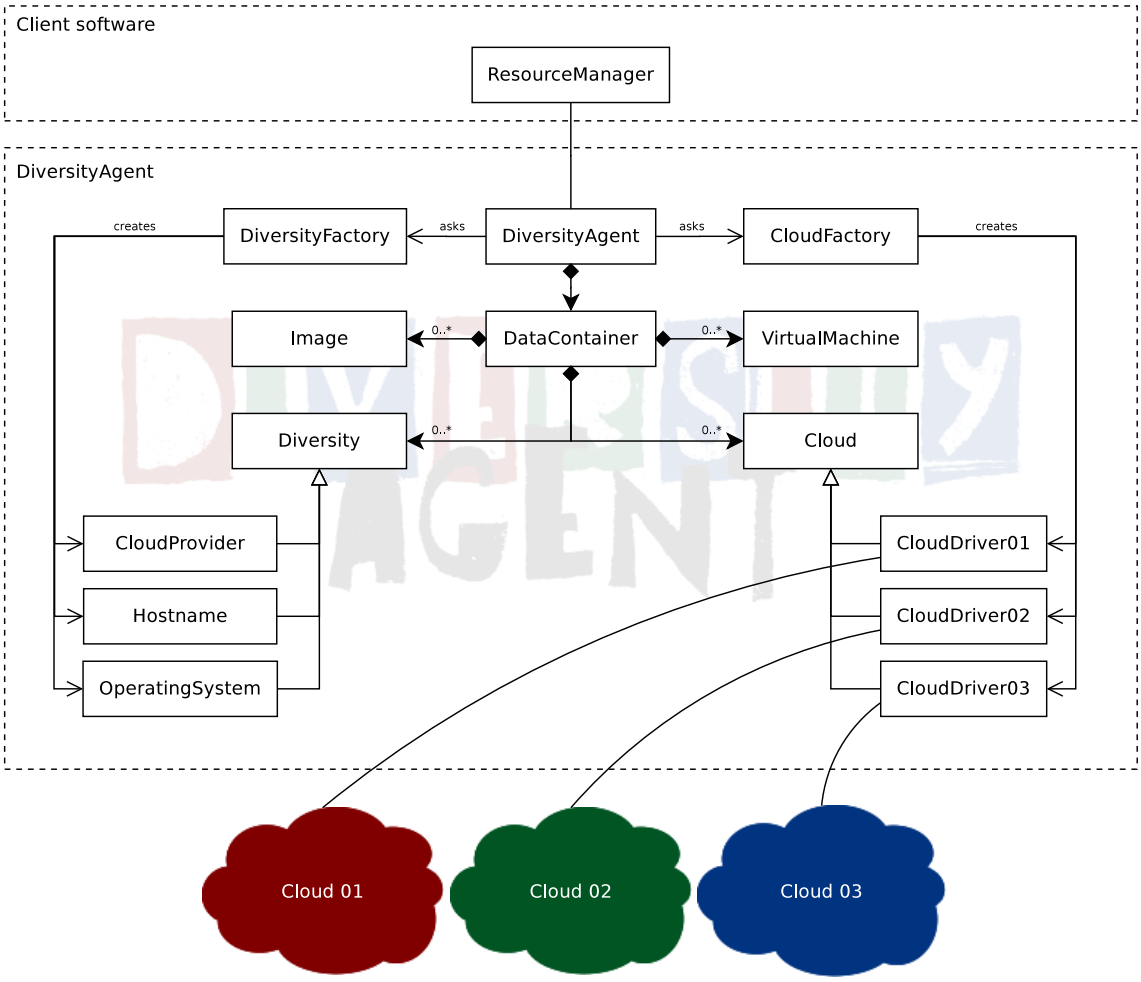

Figure 4.2: DiversityAgent class diagram.

4.2.1 How does it work?

Collaboration is a key word to explain how DiversityAgent works. There are at least three parts involved in its collaborative process, namely: clients, diversity algorithms, and cloud drivers.

When DiversityAgent is instantiated by client resource managers, it creates internally two factories and one data container that will maintain all information about resources currently being used or available to the client service in question.[1] After such initialisa-

[1] A factory is basically a component that contains a method that receives a tag (for example a string) and

tion, DiversityAgent is ready to receive the registration of available resources. Clients are responsible to register all available resources through provided interfaces, as well as, inform which diversities should be considered on resource selection. All DiversityAgent public interfaces are presented in Appendix A,

Once a client resource manager requests the creation of a virtual machine, DiversityAgent creates an empty list of properties (key value pairs) and requires for each diversity registered at that moment its contribution to the set of properties. The VM creation process is presented on Algorithm 1 and the steps followed by each diversity to create a contribution is presented on Algorithm 2.

Algorithm 1: Virtual machine creation process.

output: The created virtual machine identifier (Id) on DiversityAgent.

begin
  $P \leftarrow$ an empty list of properties;
  $DC \leftarrow$ current data container view;
  foreach *diversity* D *registered in DiversityAgent* **do**
    $P \leftarrow$ get contribution of D considering current P and DC;
  end
  $C \leftarrow$ get the selected cloud;
  $Id \leftarrow$ request C to create a VM considering P and DC;
  return Id;
end

Algorithm 2: Get contribution of a diversity algorithm.

input : The current list of properties (P) already defined and data container (DC)
output: The updated list of properties (P) with the contribution of diversity in question.

begin
  `// create the key of <key, value> pair`
  K $\leftarrow$ some key string;
  if $K \ni P$ **then**
    `// create the value of <key, value> pair`
    V $\leftarrow$ select a resource (related with K) based on DC;
    $P \leftarrow P \bigcup < K, V >$
  end
  return P;
end

Once all diversity algorithms return their contributions to the property set, it will be sent to the selected cloud driver. The cloud driver will receive and parse the properties to compose the requests and will send them to the selected cloud provider. Clients will receive a VM identifier from DiversityAgent, which can be used to request more information about the VM or delete it later.

returns a new instance of an abstract class implemented by the class associated with that tag. In DiversityAgent factories were used on diversity algorithms and cloud drivers.

4.2.2 How is it implemented?

A diagram with all internal DiversityAgent classes is presented on Figure 4.2. The following descriptions are dedicated to explain each one of them.

Client software

Client software is a system in which diversity mechanism will be inserted. It is supposed to contain a component responsible to instantiate the DiversityAgent, which here is called *ResourceManager*. This management component registers all available resources to be considered on resource selection process and requests the creation of new service instances. Other components may exist in client software, even representing other FIT mechanisms, but the only steps needed regarding the integration between resource managers and DiversityAgent library are: creating an instance of DiversityAgent, registering available clouds, VM images and diversities on DiversityAgent, and requesting service components to create or delete the service instances, which normally, are virtual machines that contain the service code.

DiversityAgent

DiversityAgent class is the entrance door for interaction with DiversityAgent library. It is instantiated by a resource manager component within the client software. This class has one *DataContainer* instance, two factories and provides all public interfaces for client software. The only functionality besides the previously mentioned is that when a new service instance is requested, this class receives the contributions from diversity algorithms, which will be send to cloud drivers in order to create a new replica with a resource combination diverse from the other replicas.

DataContainer

DataContainer is a class responsible for maintaining all information ever registered about the client service that is using DiversityAgent. It also provides a global view of information for diversity algorithms and cloud drivers similarly to a system snapshots. The container design pattern [16] was used on this class, and it has lists of all registered and active clouds, diversities, images and virtual machines allocated to the client service. All this information can be used by diversity algorithms to choose the appropriated resource that will be selected as the responsible for allocating it.

VirtualMachine

VirtualMachine class is the representation of a VM instance that runs the service code. It is used to maintain internal information about VMs, namely: the identifier on DiversityAgent, identifier on cloud provider, name of selected cloud provider, name of the selected

physical host, IP address of the service instance, VM image and amount of memory. After each new *VirtualMachine* instance be created, it is stored in the current VM list on *DataContainer* instance.

Image

Image class is the representation of a VM image, which in its turn contains an operating system and the service code. This image has to be registered on DiversityAgent component and must already exist in cloud providers, thus it is possible to create new VMs using it. The majority of diversities from support software and application diversity groups can be provided using VM images metadata.

4.2.3 The diversities

Diversity

Diversity is an abstract class that is used as a model to provide new diversity algorithms and has one principal method called *getContribution*, which is implemented to provide their contributions in a standardised way. The *Properties* Java class is used to provide the contributions, in such way that, all information inserted by diversities are expressed in key value pairs, which will be later interpreted by cloud drivers.

On the following paragraphs, the implemented diversity algorithms will be explained, and a tutorial on creating your own diversity algorithms and properties is presented in Appendix C.

DiversityFactory

DiversityFactory is an auxiliary class that facilitates the insertion of new custom diversities, through the factory design pattern [16]. It correlates tags, used to register diversity algorithms on DiversityAgent, and returns an instance of the respective class that implements the required diversity algorithm for each tag.

Diversity of cloud provider

CloudProvider is a *Diversity* class implementation and contains one of the most basic diversity algorithms on cloud computing, which creates diversity selection between different cloud providers. As discussed on previous chapter, it acts at administrative domain and security diversity groups because by choosing different cloud providers there is a probability of selecting different management human resources, administrative domains and security policies.

This diversity algorithm has three possible selection policies: round robin, current usage and historical usage. The first one simply selects the next provider from a circular

list of available cloud providers. The second consists in choosing the least used cloud provider at the moment and the third consists in selecting the least used of all times. The expected behaviour is always to find a cloud provider if at least one is registered and, in case of a tie on current or historical usage of cloud providers, the first analysed cloud with the smallest number of service instances running will be chosen.

The contribution of this diversity implementation to the property set is the property indexed by *cloud.name* key and value equal to the name of selected cloud provider. If this property has already been defined, this algorithm does nothing.

Diversity of Hostname

Hostname is the second *Diversity* implementation and aims to contribute with the previous one, increasing the diversity at administrative level, as well as contribute to security and hardware diversity groups. It allows the selection of different physical hosts that with some probability will be vulnerability and bugs independent in some contexts like natural disasters, unauthorised physical access or even physical hardware issues.

This diversity algorithm consists in selecting any physical host that is not being used by the service in question, within a previously selected cloud provider. As we want to avoid meta-scheduling approach of fetching all information from all providers, we just create a property indexed by the *host.name.differ* key and populate its value with all hosts from the chosen cloud provider that are being used by the client service. This way, the corresponding cloud driver has to know how to require any host different from those specified by the mentioned property.

The expected behaviour of this algorithm, in contribution with cloud driver algorithm for this property, is to select any physical host that is not being used by the client service. If all physical hosts of a cloud provider are being used by the service, the algorithm has to allocate a VM in the least used physical host, which means that the host that allocated less VMs for the service will be chosen.

This diversity algorithm requires a previously selected cloud provider to choose a physical host within it. With this in mind, the algorithm first verifies if the *cloud.name* property already was created on current property set. If it was not created, the algorithm chooses one cloud provider randomly, and then executes the algorithm of physical host selection. If *host.name.differ* property has already been defined, this algorithm does nothing.

Diversity of Operating System

OperatingSystem is the third and last *Diversity* implemented for this thesis and acts at diversity group of support software, which aims to select an operating system that is not being used by other current service instances.

The importance of this diversity implementation already was discussed in Section 3.5, but as discussed on [19], there are some good evidences that OS diversity plays an important role when one wants to achieve vulnerability independence.

The expected behaviour of this diversity algorithm is to select a VM image that contains an operating system that was never used on service instances, or the less used one on each cloud and physical host. The algorithm first verifies if the received set of properties already contains a selected VM image, where in this case, this algorithm does nothing. In opposite case, the algorithm verifies if some cloud provider already was chosen, where in this case, the algorithm chooses the least frequently used image on that cloud or host.

The contribution of this algorithm is presented in a property indexed by *disk.image.name* and its value is the name of the selected VM image. To achieve different OS selection, the creation of an image metadata that contains an operating system name and version is important, as discussed in Section 3.5. In our case, the information regarding the operating system of an image is passed during image registering, but the cloud provider should support this metadata to inform its clients.

An algorithm proposed by Henriques [18] uses a database table that contains all parameters needed to choose consistently an OS for a requested VM. It is meant to achieve the best combination of all active servers, regarding the highest level of vulnerability and bugs independence through OS diversity. His algorithm can be integrated with DiversityAgent as a new algorithm version for diversity of operating system.

Properties supported by DiversityAgent

Currently, DiversityAgent supports the following properties on diversity algorithms: *cloud.name, disk.image.name, host.name.equal* and *host.name.differ*. The *cloud.name* property is used to inform the cloud provider name that will be responsible to allocate the next virtual machine. The *disk.image.name* property is used to present the VM image name to be used on the next VM. The *host.name.equal* property is used to inform that a specific physical host was selected, and the *host.name.differ* property is used to inform which are the physical hosts that should not be considered on next resource allocation.

Regarding configuration properties, DiversityAgent supports the following: *vm.name.prefix*, *vm.name.suffix*, *vm.cpu*, *vm.vcpu*, *vm.memory*, and *vm.network*. The *vm.name.prefix* and *vm.name.suffix* properties are used by cloud drivers to register the name of service instances on cloud providers. The VM name is composed by its prefix, its identifier number on agent and its suffix. The *vm.cpu* and *vm.vcpu* properties are used to inform how many physical and virtual CPUs will to be allocated to each service instance, where the virtual CPUs are what will be informed as allocated to service instances. The *vm.memory* is used to inform how many megabytes (MB) of memory will be allocated for each service instance and the *vm.network* is used to inform the virtual network name which will have an interface connected with service instance.

4.2.4 Cloud drivers

Cloud

Cloud is also an abstract class that contains abstract methods to be implemented by custom cloud drivers. This class has one abstract method for each CRUD method provided by *DiversityAgent* class and is responsible to translate the defined property set to the requests that will be sent to cloud providers. On Appendix C a tutorial on how creating new cloud drivers is presented.

CloudFactory

CloudFactory is an auxiliary class, similar to *DiversityFactory*, which was also developed based on factory design pattern [16]. It facilitates the insertion of new custom cloud drivers. It correlates tags, used to register clouds on DiversityAgent, and returns an instance of respective class for each tag.

OpenNebula

Due to our project context and background knowledge on OpenNebula, we have chosen it as our first cloud driver implementation. The driver for OpenNebula uses Java OCA (OpenNebula Cloud API), which is a wrapper for OpenNebula requests that are based on XML-RPC methods.

The class that implements OpenNebula driver contains just one OpenNebula client instance and the implementation of some abstract methods proposed on *Cloud* class to create and delete virtual machines. Besides these methods, this class has some methods regarding fetching VM metadata and status, which are achieved through the response of XML-RPC methods.

The method responsible for creating VMs has to parse the defined property set and create the appropriated request to be sent to cloud provider. In OpenNebula's case, a VM template is created, similar to the presented on Listing 4.1, which contains information regarding the amount of memory and CPU to be allocated, as well as the VM image name to be used, network interfaces, physical hosts restrictions and other specific information to the hypervisor.

After VM template creation, it is sent to the cloud provider, which will answer with an identifier on cloud provider for the new service instance. Once the answer is under the cloud driver control, it is possible to register that VM identifier, as well as verify if it is already running and get its IP address and the host name that allocated the VM.

[1] Available at http://opennebula.org/documentation:archives:rel2.0:java. Accessed on March 19, 2012.

Listing 4.1: OpenNebula VM template example.

```
NAME = my-server-name
MEMORY = 1024
CPU = 2
DISK = [ IMAGE = "some-image-name" ]
NIC = [ NETWORK = "my-network-name" ]
REQUIREMENTS = "HOSTNAME != \"some-host-already-used\""
RAW = [
        TYPE = "xen",
        DATA = "builder = \"hvm\"
                shadow_memory = 8
                boot = \"c\""
                device_model = \"/usr/lib/xen-default/bin/
                   qemu-dm\"
]
```

Chapter 5

Integration and evaluation

In this chapter, we aim to present the DiversityAgent integration with two use cases foreseen in CloudFIT and its evaluation regarding correctness and performance terms.

5.1 Integration with CloudFIT use cases

As presented in Section 2.3, CloudFIT is the research project in which this thesis was developed. FITCH is a system structure proposed by CloudFIT, where DiversityAgent will be integrated with a resource manager component in order to automatically obtain diversity during resource selection process. An overview of FITCH is presented in Figure 5.1. There are two use cases foreseen by CloudFIT to FITCH, which are client-server architectures differentiated by the system model assumptions and mechanisms.

A replicated stateless web service is our first use case, where each client request is processed independently, unrelated to any other requests previously sent to the service instance in question or other instances. It is composed by some minimal amount of redundant servers, which has exactly the same service implementation, and are orchestrated by a load balancer component (Gateway component in Figure 5.1), which forwards clients requests to redundant servers. The amount of redundant servers (n) depends on the number of faults (f) that one wants to tolerate and is given by the following expression for crash faults:

$$n \geqslant f + 1$$

With this in mind, to tolerate one fault, a system needs at least two redundant servers. To tolerate two faults, a system needs at least three redundant servers and so on. The basic idea is that even with faults, there is always at least one more server running and answering client requests.

A service based on Byzantine fault tolerant state machine replication is our second use case, where each request is processed in parallel by multiple service replicas and the majority of replicas answers are compared to achieve a correct service answer. The number of replicas (n) also depends on the number of faults (f), both crash and arbitrary,

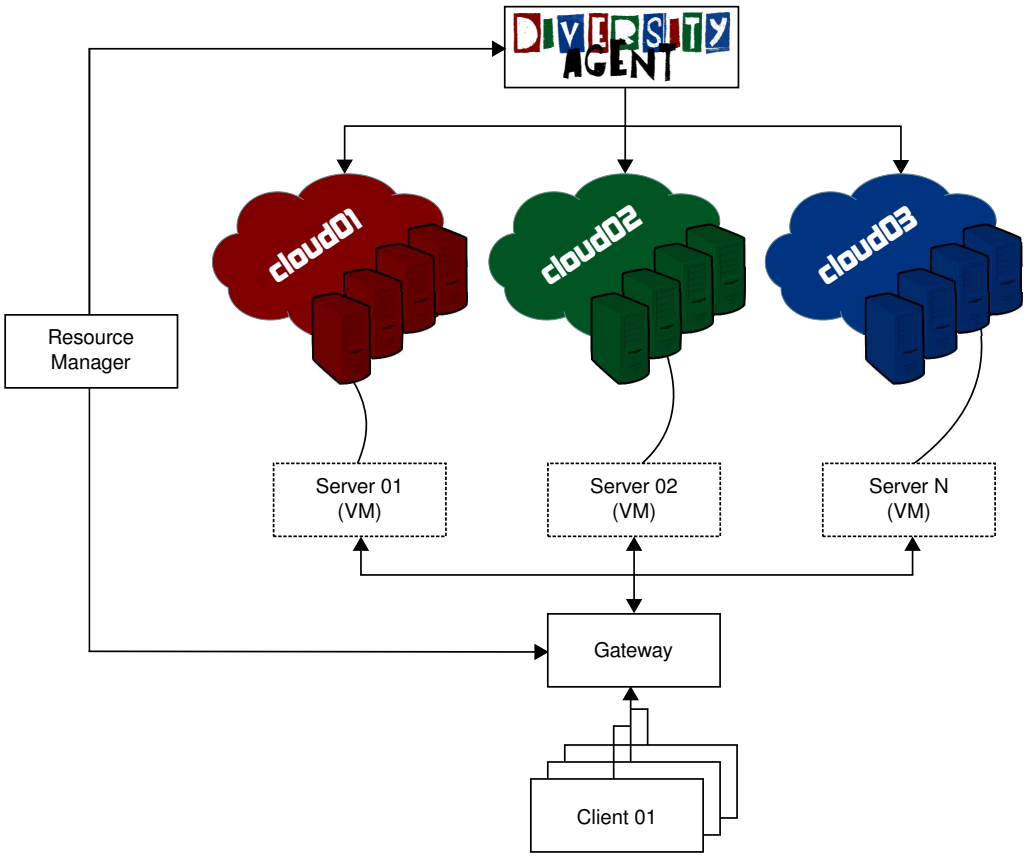

Figure 5.1: DiversityAgent integrated with CloudFIT use cases.

that one wants to tolerate and is given basically by the following expression:

$$n \geqslant 3f + 1$$

Based on this expression, to tolerate one Byzantine fault, a system needs at least four service replicas. To tolerate two arbitrary faults, a system needs at least seven service replicas and so on. The basic idea is that even with Byzantine faults, the majority of replicas are always running and answering correctly each client request.

A proactive recovery mechanism is employed in both use cases to tolerate any number of faults in entire service life time, instead of just f faults. It consists in replacing, from time to time, an old server by a new and clean one. Based on this approach, the window time available for an attacker to try to stop or disrupt a service is reduced from the entire service life time to the proactive replacement time of all servers. In the first use case, the protocol consists in terminating all pending requests, creating a new redundant server from a safe point, adding this new server to a list of servers on load balancer, removing the old replica from load balancer list and deleting the server instance on cloud provider. In the second use case, the replacement consists in terminating all pending requests, creating a new service replica from a clean state, making it to join the replica group through some

state machine replication protocol, removing the old replica from the service group and deleting the server instance in the cloud provider.

The resource manager, in Figure 5.1, is responsible to orchestrate FITCH. It has a time-triggered process that requests server replacements and has another process that controls actions foreseen by protocols when replacements happen. In a case without DiversityAgent, the resource manager should also have a process responsible for choosing resources that will be allocated for each new replica on replacements.

The DiversityAgent is placed between resource manager and cloud providers. It receives proactive requests and automatically obtains diversity for FITCH. Using this component is a modular approach to separate everything related with diversity from other mechanisms, beyond reducing the size of resource manager component. Diversities regarding cloud providers, physical hosts and operating systems were implemented because it demonstrates that it is possible to obtain diversity from more than one diversity group automatically. Other diversity algorithms can be developed through DiversityAgent as presented in Appendix C.

5.2 Evaluation

In this section we present our experimental environment, and an evaluation of DiversityAgent correctness and performance.

5.2.1 Experimental Environment

All physical resources for the experimental environment used on tests that will be presented on next sections belongs to *Quinta*, the cluster of Navigators research team. On Table 5.1 the physical hardware used on evaluations are described.

Regarding software components used on this thesis, the first to be presented is OpenNebula (version 2.0.1), which was our cloud computing management tool. This specific version was the first that handled scheduling policies through rank and requirements, which was necessary when expressing the decisions of diversity algorithms.

The main programming language used in this thesis was Java (version 6, from OpenJDK Runtime Environment and 64-bit Server VM 1.6.0_18), which is the language used on all DiversityAgent components.

5.2.2 Correctness Test

The correctness test consists in verifying if DiversityAgent component provides diversity to the available extent. Our methodology is presented through mathematical proofs and test executions, in order to match practical with theoretical expected correctness.

Component	Quantity	Description
Client software	1	Dell PowerEdge 850 Intel Pentium 4 CPU 2.80GHz 1 CPU per node, 1 core per CPU, 2 threads per core 2.8 GHz / 1 MB L2 cache 2 GB RAM (2x1GB) / DIMM Synchronous 533MHz (1.9ns) 2 x Broadcom NetXtreme BCM5721 Gigabit Ethernet Hard disk 80 GB / SCSI
Cloud providers front end	3	
Physical cloud hosts	5	Dell PowerEdge R410 Intel Xeon E5520 2 CPU per node, 4 core per CPU, 2 threads per core 2.27 GHz / 1 MB L2 cache / 8 MB L3 cache 32 GB (8x4GB) / DIMM Synchronous 1066 MHz (0.9 ns) 2 x Broadcom NetXtreme II BCM5716 Gigabit Ethernet 2 x Intel Ethernet interface Hard disk 146 GB / SCSI

Table 5.1: Experimental environment hardware description.

The mathematical proof result in two main theorems: one to settle correctness for a single diversity algorithm, and one to settle correctness when combining diversity algorithms. Most of discussion are based on combinatorics and its properties applied on obtaining diversity from resource selection process.

Definition 1 *Combination, in mathematical combinatorics, is the process of selecting a subset of elements from a finite set, where order does not matter and the number of combinations is given by the following binomial coefficient:*

$$C_p^n = \frac{n!}{p! \times (n-p)!}, \tag{5.1}$$

where n *represents the amount of elements on finite set, and* p *represents the quantity of elements that will compose the new subset.*

Lemma 1 *The quantity of possible combinations of only one element from a finite set is equal to the amount of elements that compose the entire set.*

Proof: From Expression 5.1, we have that *n* is the amount of elements on finite set, which will follow being *n*. And we have also that *p* represents the amount of elements that will compose the new subset, which in this case is 1. With this in mind, the quantity of possible combinations of only one element is given by the following expression:

$$C_1^n = \frac{n!}{1! \times (n-1)!} = \frac{n \times (n-1)!}{1! \times (n-1)!} = \frac{n}{1!} = \frac{n}{1} = n \tag{5.2}$$

Lemma 2 *The quantity of possible combinations of one element among those provided on diversity in question is equal to the degree of diversity.*

Proof: Considering that the quantity of elements that will compose the new subset is 1, from a finite set with size *n*, and from Expression 5.2, the quantity of possibilities provided on diversity in question is *n*. Finally, from the definition of degree of diversity in Section 3.1, *n* will always be equal to the degree of diversity.

Theorem 1 *If a single Diversity implementation can deploy the same amount of VMs as the degree of diversity provided, without repeating the resource selected in question, then it can be considered as correct.*

Proof: First, as settled on Lemma 2, it is possible to select the same amount of possibilities as the degree of diversity. Second, as our meaning of correctness is the capability of guarantee the highest level of diversity as possible, the Diversity implementation need to provide at least the same amount of VMs as the degree of diversity, without repeating the selected resource. This can only be achieved if and only if an algorithm select all possibilities before start repeating its choices.

Definition 2 *In combinatorics, rule of product is a counting principle that focuses on count the number of ways a task can occur given a series of events, where basically the number of possibilities of each event is multiplied by others, as the following expression:*

$$C_f = C_{p1}^{n1} \times C_{p2}^{n2} ... \times C_{pN}^{nN} \tag{5.3}$$

Lemma 3 *The quantity of all possibilities when combining independent resources through the selection of one element per resource is equal to the product of all degree of diversities in question.*

Proof: From the rule of product (Expression 5.3) and on Lemma 1 we can achieve the following expression:

$$C_f = C_1^{n1} \times C_1^{n2} ... \times C_1^{nN} = n1 \times n2 ... \times nN, \tag{5.4}$$

which proves that the number of combinations of all resources is given by the product of all degrees of independent diversity in question.

Lemma 4 *If two or more resources have hierarchical relations, then just the child components can be considered on rule of product.*

Proof: Based on the existence of inheritance between resources, when selecting some child component, automatically a parent components will be selected, which cannot be considered on rule of product. To exemplify, if we consider 2 cloud providers with 2 physical hosts each and 1 cloud provider with 1 physical host, we have two sets of resources: cloud providers (with 3 elements) and physical hosts (with 5 elements). In this

case, if we use both degrees of diversity on rule of product we should be able to deploy 15 VMs without repeating the resource combination, but this is not possible, because each host is connected just to one cloud provider. The correct in this case is to consider just the number of physical hosts existent, which lead to 5 VM deployments before starting repeating the resource combination, because when selecting the physical host, automatically we are selecting a cloud provider.

Theorem 2 *If any number of independent Diversity implementation can deploy the same amount of VMs as the product of all degrees of diversity provided in question, without repeating the resource combinations, then they can be considered as correct.*

Proof: First, as settled on Lemma 3, it is possible to select the same amount of resource combinations as the product of degrees of diversity in question, if they are not hierarchically related (Lemma 4). Second, as our meaning of correctness is the capability of guarantee the highest level of diversity as possible, the Diversity implementation need to provide at least the same amount of VMs as the product of degrees of diversity, without repeating resource combinations, which can just be achieve if and only if the algorithm select all possibilities before start repeating its choices.

Considering the mathematical proofs previously presented and that all diversities implemented will be used, we choose the following hierarchical diversity order to be used on resource selection: (1) diversity of operating system; (2) diversity of cloud provider; (3) diversity of physical host.

Our system is composed by two cloud providers with two physical hosts each and one cloud provider with only one physical host. In addition, we will provide three VM images with different Operating Systems (Ubuntu Dapper 6.06 LTS, Ubuntu Intrepid 8.10 and Ubuntu Oneiric 11.10). Considering our theoretical correctness analysis, with three cloud providers, five physical hosts and three VM images with different OS. Then DiversityAgent should be able to deploy fifteen virtual machines without repeating the resources combination, remembering that just the number of VM images (3) and physical hosts (5) are considered in the rule of product, due to hierarchical relation between physical hosts and cloud providers.

Our methodology for this test consists in deploy sixteen (16) service instances to verify the correctness of diversity algorithms. This is possible because the previously presented proofs, which leads to conclude that will be possible to select fifteen (15) different compositions before starting repeating some previous configuration.

The algorithm implemented to correctness test is just a simple resource manager that registers the resources available and requests virtual machines creation, presentation or deletion. The basic steps are presented on Algorithm 3.

After the execution of Algorithm 3, we obtain the resource distribution presented on

Algorithm 3: Correctness test algorithm.

```
create a new instance of DiversityAgent;
create clouds (cloud_provider_01, cloud_provider_02, cloud_provider_03);
create images (Ubuntu Dapper, Ubuntu Intrepid, Ubuntu Oneiric);
create diversities (OPERATING_SYSTEM, CLOUD, PHYSICAL_HOST);
for i=0 ; i<16 ; i++ do
    create a virtual machine;
end
for i=0 ; i<16 ; i++ do
    print information about VM(i);
    delete VM(i);
end
```

VM ID	Cloud Provider	ID on Cloud	Host	VM IP Address	OS Name
0	cloud_provider_01	270	s4	192.168.2.33	Ubuntu Dapper 6.06
1	cloud_provider_02	0	s6	192.168.2.39	Ubuntu Intrepid 8.10
2	cloud_provider_03	1935	s7	192.168.2.46	Ubuntu Oneiric 11.10
3	cloud_provider_02	1	s5	192.168.2.40	Ubuntu Dapper 6.06
4	cloud_provider_03	1936	s7	192.168.2.47	Ubuntu Dapper 6.06
5	cloud_provider_01	271	s3	192.168.2.34	Ubuntu Intrepid 8.10
6	cloud_provider_03	1937	s7	192.168.2.48	Ubuntu Intrepid 8.10
7	cloud_provider_01	272	s4	192.168.2.35	Ubuntu Oneiric 11.10
8	cloud_provider_02	2	s6	192.168.2.41	Ubuntu Oneiric 11.10
9	cloud_provider_02	3	s6	192.168.2.42	Ubuntu Dapper 6.06
10	cloud_provider_01	273	s3	192.168.2.36	Ubuntu Dapper 6.06
11	cloud_provider_01	274	s4	192.168.2.37	Ubuntu Intrepid 8.10
12	cloud_provider_02	4	s5	192.168.2.43	Ubuntu Intrepid 8.10
13	cloud_provider_02	5	s5	192.168.2.44	Ubuntu Oneiric 11.10
14	cloud_provider_01	275	s3	192.168.2.38	Ubuntu Oneiric 11.10
15	cloud_provider_01	276	s4	192.168.2.32	Ubuntu Intrepid 8.10

Table 5.2: Resource distribution result of correctness test execution.

Table 5.2, which is also presented hierarchically in Figure 5.2.

Based on this results, the correctness of all diversities acting together is basically proved, since the first 15 deployments used different configurations and on 16^{th} deployment it started to repeat some previous choices.

5.2.3 Performance Test

The performance test consists basically in verifying if DiversityAgent component causes a substantial overhead to client software that use it to obtain diversity through cloud computing resource selection. Our methodology is to follow a mathematical composition of each time element to be considered and present the overhead caused by DiversityAgent

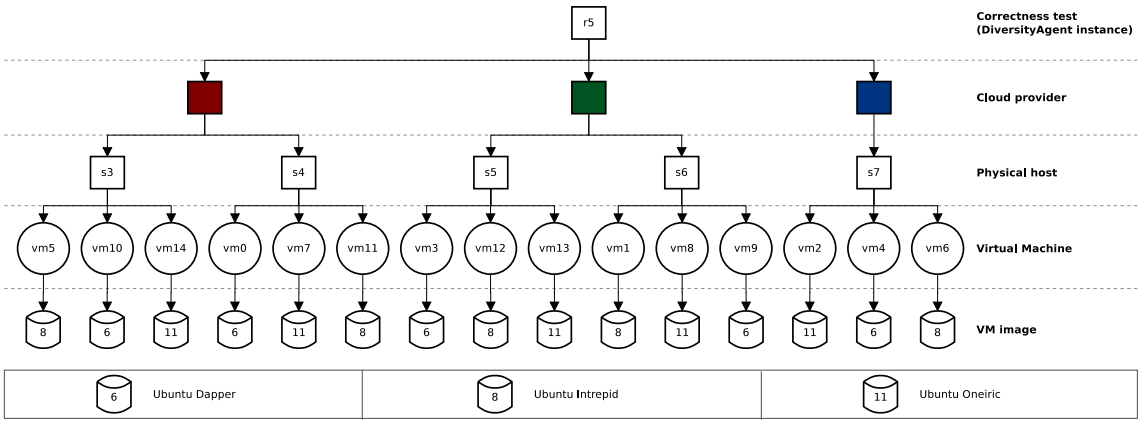

Figure 5.2: Hierarchical view of correctness test result for the first 15 VMs.

component. Our approach was to instrument the source code to obtain the exact and real values for time components representing each task. The complete list of timing metrics that will be considered on this performance test are presented on Table 5.3.

Time	Description
T_a	Time of one VM deployment with DiversityAgent.
T_b	Time of one VM deployment without DiversityAgent.
T_1	Diversity properties preparing time.
T_2	Properties parsing time.
T_3	Cloud request preparing time.
T_4	Cloud asynchronous request time.
T_5	VM pending time.
T_o	The real time representing the overhead.

Table 5.3: Time composition of performance test.

The first element to be calculated is the time spend with a deployment of only one VM using DiversityAgent. This time (T_a) is composed by the sum of all task times (from T_1 to T_5), resulting in Expression 5.5.

$$T_a = T_1 + T_2 + T_3 + T_4 + T_5 \tag{5.5}$$

The second element is exactly the same time spend with a deployment of only one VM, but now without DiversityAgent, which is given by the variable T_b. For this time computing, a client similar to an architecture controller was created, which requests statically the same amount of virtual machines for each cloud provider as the previous case. This metric is given by the sum of some task times (from T_3 to T_5) in Expression 5.6.

$$T_b = T_3 + T_4 + T_5 \tag{5.6}$$

Considering the difference of T_a and T_b definitions, it is already possible to conclude that the only differences between DiversityAgent usage or not are time components represented by T_1 and T_2. Thus, the real time representing the overhead (T_o) is given by the sum of this two elements, as shown in Expression 5.7.

$$T_o = T_1 + T_2 \tag{5.7}$$

Finally, the best overhead presentation is to represent it through the ratio between real overhead time and total virtual machine deployment time, as shown in Expression 5.8.

$$Overhead = \frac{T_o}{T_a} \tag{5.8}$$

After all variables being settled, it is possible to execute the performance test and gather time results of each task. The results of T_a, T_b, T_o and $Overhead$, from fifteen VM creations on each cloud, are presented in Table 5.4, in seconds.

Time		cloud_provider_01	cloud_provider_02	cloud_provider_03
T_a	min	118	110	149
	max	130	114	152
	avg	127	111	150
	$mean$	124	112	150
T_b	min	120	109	149
	max	129	122	152
	avg	124	115	150
	$mean$	124	115	150
T_o	min	0.000038	0.000039	0.000039
	max	0.0047	0.000073	0.00012
	avg	0.000974	0.000048	0.000053
	$mean$	0.0023	0.000056	0.00008
$Overhead$		0.000007	0.0000004	0.0000003

Table 5.4: Performance test results in seconds and final overhead.

Based on this results, it is possible to verify and proof that DiversityAgent does not cause a substantial overhead when a small amount of resources is considered. The overhead caused by our component, considering an average among all clouds, is near to

0.00025% and in the worst case, it is 0.0007%, which also is a good result. High variations on deployment time in the same cloud occurs because deployment time is composed by three time components: scheduling, transferring and booting, which create such divergences. Difference between different clouds can be caused by many internal factors like network or processing overload, service uptime (aging issues) or scheduling tasks.

Analysing algorithm complexity time is important for scalability. Diversity properties creation (T_1) is a task where all registered diversities provide their contributions and the more registered diversities, the more time it takes. In our case, we have three diversity algorithms that separately have their own internal complexity time.

Cloud provider diversity has a linear ($O(n)$) complexity time, once it has to select a cloud provider that has never been used or the less used from the entire set of registered cloud providers. Host name diversity algorithm has a polynomial ($O(m \times n)$) complexity time, but it depends on two resources, the virtual machines running the service and the physical hosts that already were used by client system. The last algorithm, regarding diversity of operating systems, is polynomial ($O(m \times n)$) to select a never used VM image, but if it is not found at first time, other methods for selecting the VM image are executed. For example, find an image that has never been used in some cloud ($O(m \times n \times p)$), find an image that has never been used in some physical host ($O(m \times n^2)$), and select a random image ($O(n/2)$) as last resource.

Properties parsing (T_2) is an algorithm where all properties are translated to cloud request language and the more properties, the more time it takes. Current DiversityAgent version only has OpenNebula cloud driver, which parses all supported properties. From this point of view, the complexity time is linear ($O(n)$), but we have to consider that the *host.name.differ* property is composed by a list of physical hosts that should be avoided on resource selection in question, which leads to a polynomial complexity time of ($O(m \times n)$).

At the end, DiversityAgent has a polynomial complexity time when considering all diversity algorithms implemented, because the property set is composed in a task that is a sequence of all algorithms previously discussed, which leads to the summation of all complexity time. Improving complexity time is a good start point to improve DiversityAgent performance. Previous complexity analysis shows that diversity algorithms are not scalable for really large amounts of resources and servers, but for CloudFIT use cases it was efficient enough. Proving that automatic diversity obtention in cloud computing resource selection is possible was our initial step, but future steps can consider obtaining diversity even more effectively with better complexity time. Resource selection through binary search or balanced search trees can provide logarithmic complexity time, which is a much better result in complexity terms.

Chapter 6

Conclusions

6.1 Final remarks

This thesis focused in obtaining automatically diversity from cloud computing resource selection. Our two main scientific contributions arose from this general goal: an analysis of diversity in cloud computing scenario and a library for automatic obtention of diversity. In addition, some basic mechanisms of fault and intrusion tolerance were presented, the state of the art on diversity in cloud computing was analysed, and a contextualisation of this thesis with CloudFIT project was provided.

Regarding our first contribution, a diversity analysis in cloud computing environments, we proposed some minor modifications to existent diversity taxonomies to obtain a dry classification. Beyond such classification, we analysed all diversity groups occurrence in cloud computing scenario and indicated opportunities for IaaS providers improve diversity management area. More than fifty properties were identified, where four are from application diversity group, fourteen from administrative, ten from geographic location, nine from support software, nine from hardware and six from security diversity group. From the fifty two properties, only eight are completely supported by OpenNebula and thirteen by Amazon. Amazon still partially supports eighteen properties through the usage of generic tags. We believe that there are still a considerable amount of diversities opportunities to be defined and we expect all to be compatible with DiversityAgent.

Comparing both cloud players, Amazon supports more VM image metadata than OpenNebula, but it provides less hardware information. Diversity properties regarding geographic location is almost not supported either by Amazon or by OpenNebula. Amazon supports generic tags for resources, which is a solution that partially support some properties.

Big exchange proposals are not always well regarded by established companies like Amazon. Cloud providers may not agree to inform all proposed properties, once there are commercial risks and extra costs in publishing and maintaining all information addressed on this thesis, but we consider such discussion an important step on diversity management

area. Our aim was to present and discuss as many properties as possible to accomplish an extensive diversity analysis.

Concerning our second contribution, we developed the DiversityAgent, a Java library for automatic diversity obtention during cloud computing resource selection. This component was designed considering four functional, nine non-functional requirements and some well known design patterns. We established a collaborative approach for diversity properties composition, which was presented in details its working processes. In addition, we explained the DiversityAgent internal composition, as well as, the diversities and cloud drivers implemented.

Diversity of cloud providers, physical hosts and operating systems are the three diversities implemented within this thesis. The only completely supported property from this three by OpenNebula is the physical host. Cloud provider is a property that is achieved externally to OpenNebula, because it exist just in federated or cloud-of-clouds scenarios. Operating system name is a property that is not supported by OpenNebula, but we circumvented this limitation in our algorithm through the VM image name composition by more information than the normally required. OpenNebula is the cloud tool chosen by CloudFIT and is the only cloud driver implemented on DiversityAgent. There are tutorials on how to use or customise the DiversityAgent on this document appendices.

DiversityAgent was integrated with two use cases foreseen in the CloudFIT project. The first is basically a stateless web server and the second is a service based on state machine replication. Both use cases consider proactive mechanisms and positioned DiversityAgent between the client resource manager and the cloud providers, in order to obtain diversity automatically by using our component.

A correctness analysis was provided, giving proofs that implemented diversity algorithms can be considered as correct from functional requirements perspective. We believe that there are many other possible correct algorithms, each one depending on correctness meaning and requirements. Our focus was to prove that it is possible to obtain several diversities automatically and provide a generic library for this task. Regarding performance, we verified that even with low overhead in CloudFIT use cases, which considered small amount of resources, the diversity algorithms and cloud drivers can be improved in order to reduce the complexity time. One possible approach for such improvement is consider binary tree search in resource selection process, instead of the current linear model.

DiversityAgent evolution and continuity were also this thesis concerns, once we employed techniques that facilitate DiversityAgent usage, customisation and maintainability. It is a free and open source software available in its page [10] on Google Project Hosting, under the GNU Lesser General Public License (LGPL v3.0). We hope this library may contribute to many future Navigators and external projects to fill some open problems in diversity management area.

6.2 Future work

In this section we present some opportunities of continuity on DiversityAgent development and diversity management area improvements. The first possible step is to transform DiversityAgent in a multi service agent, in order to declare just one component instance for all services of a given client. We decided by this single service approach because the use cases foreseen in CloudFIT just consider one service per time. Other motivation is because it is simpler and easier to promote the first DiversityAgent version, as well as gather contributors to develop extensions for the component.

Regarding component extensions, the second possible future work is one of the most important, which is the creation of more cloud drivers and diversity algorithms. The cloud computing model is a growing market, where each time more players are coming up. With this in mind, we identify the need of implementing more options of cloud drivers to follow cloud computing growth and become a useful solution for many researching projects, as well as a reference for industry products. At the same time, to reach new users and scientific niches we identify the same need of implementing more options of diversity properties and algorithms.

The third possible future step encompasses the promotion of all diversity opportunities pointed by our analysis to the largest number possible of cloud computing players. This is extremely important for diversity management area and its evolution, in order to became a half- or completely solved problem in computing.

The last future work is the integration of our component with tools dedicated to provide automatic creation of diversity, once our solution is focused on obtaining diversity only during resource selection phase. Tools able to automatically create VM images are really important in a partnership with DiversityAgent to provide a complete solution on automatic diversity management. Our tool could select resources that only exist in DiversityAgent and request their on-demand creation on cloud providers.

Appendix A

DiversityAgent public interfaces

This appendix contains the DiversityAgent public interfaces explained.

A.1 Initialising DiversityAgent

There are two main ways to instantiate the DiversityAgent library. The first one is using the simplest constructor, without any argument, as follow:

```
public DiversityAgent();
```

The second is passing as argument a path to a recoverable state file. This file basically consist in a *DataContainer* object that extended the *Serializable* Java class, which was previously stored with the *saveState* interface. The constructor in this case is the following:

```
public DiversityAgent(String theRecoveryStateFile);
```

A.2 Announcing cloud providers

After DiversityAgent initialisation, its clients can register which are the cloud providers that can be used on resource selection process. The interface to register a cloud is the following:

```
/**
 * Register a cloud provider in the current session.
 * @param theCloudName A identifier name for the cloud provider.
 * @param theTool The tag that define which cloud driver will be used.
 * @param theType The cloud deployment model (private, public or hybrid).
 * @param theAddress The IP address to the cloud provider's front-end.
 * @param theUsername The user registered on the cloud provider.
 * @param thePassword The password registered for the user.
 * @return The successfulness of registering the cloud provider.
 */
public boolean createCloud(String theCloudName,
                           String theTool,
                           String theType,
                           String theAddress,
                           String theUsername,
                           String thePassword);
```

As DiversityAgent is a dynamic component, client can remove cloud providers in execution time, through the following interface:

```
/**
 * Remove a cloud provider in the current session.
 * @param theCloudName The identifier name registered to be removed.
 * @return The successfulness of removing the cloud provider.
 */
public boolean deleteCloud(String theCloudName);
```

A.3 Providing VM images

Similarly to cloud provider, DiversityAgent clients have to inform which virtual machine images are registered in which cloud provider, in order to select them when a virtual machine creation is required. The interface to publish a VM image registered in all cloud providers is the following:

```
/**
 * Register a VM image as available in all clouds already registered.
 * @param theImageName The VM image name to be registered.
 * @return The successfulness of registering the VM image.
 */
public boolean createImage(String theImageName);
```

If the client has some images that are not registered in all cloud providers, then he can register VM images in each cloud provider separately, as follow:

```
/**
 * Register a VM image as available in some clouds already registered.
 * @param theImageName The VM image name to be registered.
 * @param theCloudNames The clouds to be linked to the VM image.
 * @return The successfulness of registering the VM image.
 */
public boolean createImage(String theImageName, String theCloudNames[]);
```

The same specific process can be done when deleting VM images from a cloud provider:

```
/**
 * Unregister a image from a specific cloud provider.
 * @param theImageName The VM image name to be unregistered.
 * @param theCloudName The cloud provider that is linked to the image.
 * @return The successfulness of deleting the VM image in that cloud.
 */
public boolean deleteImage(String theImageName, String theCloudName);
```

If the image is registered in all cloud providers, clients can delete the image from all providers at the same time.

```
/**
 * Unregister a image from all registered cloud providers.
 * @param theImageName The VM image name to be unregistered.
 * @return The successfulness of deleting the VM image in all clouds.
 */
public boolean deleteImage(String theImageName);
```

If clients want to delete all image from a specific cloud provider, they can use the following interface:

```
/**
 * Unregister all VM images from a specific cloud provider.
 * @param theCloudName The cloud provider name to be unlinked to all images.
 */
public void deleteAllImagesFromCloud(String theCloudName);
```

A.4 Working with diversities

Obtaining diversity automatically is the main goal of DiversityAgent. Clients have to register which are the diversities that our component should consider on resource selection process. The interface to add diversities is the following:

```
/**
 * Register one or more diversities in the current session.
 * @param theDiversities Group of diversities to be registered in the DataContainer.
 * @return The successfulness of registering the diversities.
 */
public boolean createDiversity(String theDiversities[]);
```

While the interface to delete diversities from the considered range is the following:

```
/**
 * Remove one or more diversities from the current session.
 * @param theDiversities Group of diversities to be removed from the DataContainer.
 * @return The successfulness of removing the diversities.
 */
public boolean deleteDiversity(String theDiversities[]);
```

A.5 VM related requests

Processing resources are selected by DiversityAgent considering diversities previously registered. Clients can request the creation of virtual machines through the following interface:

```
/**
 * Create a new Virtual Machine through the usage of all diversities.
 * registered in current state.
 * @return The successfulness of creating the VM.
 */
public int createVm();
```

The same creation can be done if clients already decided on some resource selection, through the passage of their own property set:

```
/**
 * Create a new Virtual Machine through the usage of all diversities
 * and helped by some properties passed as argument.
 * @param theProperties Some properties to be considered on the resource selection.
 * @return The successfulness of creating the VM.
 */
public int createVm(Properties theProperties);
```

After the VM creation, it is possible to read information about the created server:

```
/**
 * Read the instance of VirtualMachine indexed by the ID on agent.
 * @param theVmIdOnAgent The identifier of the instance to be read.
 * @return The VM instance indexed on agent by the identifier.
 */
public VirtualMachine readVm(int theVmIdOnAgent);
```

As well as it is possible to just fetch the IP address of a specific VM:

```
/**
 * Read the IP address of a specific VirtualMachine.
 * @param theVmIdOnAgent The identifier of the instance to be read.
 * @return The IP address of the specified VM.
 */
public String readVmIpAddress(int theVmIdOnAgent);
```

It is possible to remove a virtual machine from the service group through the following interface:

```
/**
 * Remove a Virtual Machine from the current group.
 * @param theVmIdOnAgent The identifier of the instance to be removed.
 * @return The successfulness of removing the VM.
 */
public boolean deleteVm(int theVmIdOnAgent);
```

A.6 Recovery feature

Recovering service state is an important feature for fault tolerant components, once through the following interface it is possible to recovery a previously saved state:

```
/**
 * Load an instance of DataContainer to the current session.
 * @param theFileName Path to the file to be read in order to load the state.
 * @return The successfulness of loading the state contained in the file.
 */
public boolean loadState(String theFileName);
```

To save the state at any execution time, clients can use the following interface:

```
/**
 * Save the current instance of DataContainer in a file.
 * @param theFileName Path to the file to be created or overwritten.
 * @return The successfulness of saving the state in the file.
 */
public boolean saveState(String theFileName);
```

Appendix B

Using DiversityAgent

This tutorial explains how to use DiversityAgent in client systems. It is divided into three sections, from preparation phase, to basic and advanced usage.

B.1 Preparing

The preparation phase consists in obtaining the required packages and positioning them on your project folder. Obtaining a functional DiversityAgent package is the first step to be done, and can be achieved through two different ways.

B.1.1 Obtaining from DiversityAgent site

Download the latest stable version of DiversityAgent from the Downloads page [10]. It will fetch a previously compiled package, which was created following the same steps of source code based (explained in next section).

```
$ wget http://diversity-agent.googlecode.com/files/DiversityAgent-v0.1b.jar
```

B.1.2 Obtaining from DiversityAgent source code

Checkout the DiversityAgent source code from the SVN repository. It will fetch the current stable version of all DiversityAgent classes source code.

```
$ svn checkout http://diversity-agent.googlecode.com/svn/trunk/ diversity-agent
```

Create a new folder for some required libraries inside the diversity-agent directory and download DiversityAgent dependencies (see Downloads).

```
$ cd diversity-agent
$ mkdir lib
$ cd lib
$ wget http://diversity-agent.googlecode.com/files/opennebula-client-2.0.1.jar
$ wget http://diversity-agent.googlecode.com/files/ws-commons-util-1.0.2.jar
$ wget http://diversity-agent.googlecode.com/files/xmlrpc-common-3.1.2.jar
$ wget http://diversity-agent.googlecode.com/files/xmlrpc-client-3.1.2.jar
$ cd ..
```

Compile all DiversityAgent classes linking them with fetched libraries.

```
$ javac -cp \
"lib/opennebula-client-2.0.1.jar:lib/ws-commons-util-1.0.2.jar:lib/xmlrpc-client-3.1.2.
    jar:lib/xmlrpc-common-3.1.2.jar:." \
src/diversity_agent/*.java src/drivers/*.java src/diversities/*.java
```

Create a manifest file, which will contain information regarding the utility JAR files existent in lib directory.

```
$ cat > MANIFEST.MF << EOF
Manifest-Version: 1.0
Created-By: 1.6.0_20 (Sun Microsystems Inc.)
Class-Path: lib/opennebula-client-2.0.1.jar lib/ws-commons-util-1.0.2.jar lib/xmlrpc-
    client-3.1.2.jar lib/xmlrpc-common-3.1.2.jar


EOF
```

Create the DiversityAgent JAR to be used by other applications.

```
$ jar cvfm DiversityAgent-v0.1b.jar MANIFEST.MF -C src . lib
```

B.1.3 After obtaining the package

Move the DiversityAgent package to your application folder. The $PATH_TO_APP variable is the path location of your application folder root.

```
$ mv ./DiversityAgent-v0.1b.jar <$PATH_TO_APP>/lib
```

B.2 Basic usage

The basic usage consists in importing the DiversityAgent, coding some basic interactions with it, compiling and running the client system. Import the DiversityAgent class in your application resource manager component.

```
...
import diversity_agent.DiversityAgent;
...
```

Create a DiversityAgent instance for your service.

```
...
DiversityAgent myAgent = new DiversityAgent();
...
```

Register available cloud providers, VM images and diversities.

```
...
// Registring a cloud provider
myAgent.createCloud("my_cloud_01_name", "OPEN_NEBULA", "Private", "192.168.2.140",
        "username", "password");

// Registering an available VM image
myAgent.createImage("Ubuntu Oneiric 11.10#GNU/Linux#Ubuntu 11.10");

// Registering diversities to be considered on resource selection
myAgent.createDiversity(new String[] {"OPERATING_SYSTEM","CLOUD","PHYSICAL_HOST"});
...
```

Create and delete service instances, considering that using the createVm interface without any argument means that the registered clouds are configured accordingly with the default DiversityAgent properties.

```
...
int id;
id = myAgent.createVm();
myAgent.deleteVm(id);
...
```

B.3 Advanced usage

Providing a predefined property set is the current way to configure some VM aspects without modifying the DiversityAgent source code. Such provisioning is also important when you want to ignore some diversity algorithm during virtual machine creation. For both use cases, create an empty property set on your resource manager, include an existent property with valid values and request the creation of a new server considering such properties.

```
...
int id;
Properties myProperties = new Properties();
//inserting VM properties
myProperties.setProperty("vm.cpu", "4");
myProperties.setProperty("vm.vcpu", "4");
myProperties.setProperty("vm.memory", "8192");
myProperties.setProperty("vm.network.name", "Private");
// inserting diversity properties
myProperties.setProperty("cloud.name", "my_cloud_01_name");
id = myAgent.createVm(myProperties);
...
```

Once a virtual machine is created through createVm, the interface will return an internal identifier regarding the created VM, which is meaningful only in DiversityAgent context. Use the readVm interface to obtain further information regarding the created virtual machine.

```
...
VirtualMachine myVm = myAgent.readVm(id);
System.out.println("Identifier on agent: " + id);
System.out.println("Identifier on cloud: " + myVm.getIdOnCloud());
System.out.println("Cloud name: " + myVm.getCloudName());
System.out.println("Host name: " + myVm.getHostName());
System.out.println("VM IP address: " + myVm.getIpAddress());
System.out.println("VM IP address: " + myAgent.getIpAddress(id);
System.out.println("VM image: " + myVm.getImageName());
System.out.println("VM memory (in MB): " + myVm.getMemory());
...
```

The last advanced usage topic is the recovery feature. You can save or load all information maintained by DiversityAgent on its DataContainer about the service session in question, through the following interfaces.

```
...
myAgent.saveState("/tmp/DiversityAgent.state");
myAgent.loadState("/tmp/DiversityAgent.state");
...
```

B.4 Finalising

All resources created with or registered in DiversityAgent can be deleted or removed from current state at any time. Clean the entire service session with the following steps.

```
...
for(int i=0 ; i<myVmIdGroup.size() ; i++){
        mDiversityAgent.deleteVm(myVmIdGroup.get(i));
}
myAgent.deleteImage("Ubuntu Oneiric 11.10");
myAgent.deleteCloud("my_cloud_01_name");
myAgent.deleteDiversity(new String[] {"OPERATING_SYSTEM","CLOUD","PHYSICAL_HOST"});
...
```

Appendix C

Customizing DiversityAgent

C.1 Creating new diversity algorithms and properties

We provide a modular structure where it is possible to insert new diversities through the following steps. To exemplify, we will explain how to create a new diversity for software support to obtain hypervisor diversity (e.g. Xen, KVM, Hyper-V, VMWare, etc).

Create a *Hypervisor* class that extends *Diversity* class in a *Hypervisor.java* file and save this in diversities folder (*DiversityAgent/src/diversities*).

```
package diversities;
public class Hypervisor extends diversity_agent.Diversity {
        ...
}
```

Create the class constructor and define the diversity group.

```
...
        public Hypervisor() {
                mDiversityAxis = diversity_agent.DIVERSITY_AXIS.SUPPORT_SOFTWARE;
        }
...
```

Implement the *getContribution* abstract method, which receives a property set and a *DataContainer* instance copy with DiversityAgent current state. Imagining that your property is a list of all hypervisors that already were used by your service and that you want to avoid in next VM creations, then you can define, for example, *hypervisor.name.differ* as your property key.

```
...
        public Properties getContribution(Properties theProperties, DataContainer
            theDataContainer) {
                //Verify if your property does not exist
                if(theProperties.getProperty("hypervisor.name.differ") != null){
                        return theProperties;
                }
                //Define an algorithm to get all hypervisors already used
                ...
                //Add your property to the property set
                theProperties.setProperty("hypervisor.name.differ", myHypDifferList.
                    toString());
                //Return the new set of properties
                return theProperties;
        }
```

```
...
```

Create a tag and define it to instantiate your *Hypervisor* class on *DiversityFactory* source code. Normally, this tag is different of property key, because it will be used for other purposes. It is not a standard, but it can be defined in uppercase, for example, “HYPERVISOR”.

```
...
package diversity_agent;
...
public class DiversityFactory {
        public Diversity getDiversity(String theDiversity) {
                Diversity aDiversity = null;
                if(theDiversity.equals("CLOUD")) {
                                aDiversity = new diversities.CloudProvider();
                } else if (theDiversity.equals("PHYSICAL_HOST")) {
                        aDiversity = new diversities.Hostname();
                } else if (theDiversity.equals("OPERATING_SYSTEM")) {
                        aDiversity = new diversities.OperatingSystem();
                } ...
                //Here your code starts
                } else if (theDiversity.equals("HYPERVISOR")){
                        aDiversity = new diversities.Hypervisor();
                }
                //Here it ends
                return aDiversity;
        }

}
...
```

Modify cloud drivers to make them aware of your properties, which in this case is done on *OpenNebula.java* file available on drivers folder (*DiversityAgent/src/drivers*).

```
...
        public VirtualMachine createVm(Properties theProperties, DataContainer
            theDataContainer) {
                ...

                if(theProperties.getProperty("hypervisor.name.differ") != null) {
                        //Define an algorithm to insert all hypervisors already used in
                            your request or template
                        //In OpenNebula, we could use the following expression in
                            template:
                        //REQUIREMENTS = "HYPERVISOR != \"vmware\" & HYPERVISOR != \"xen
                            \""
                        ...
                }

                //The request to cloud provider is exactly the same
                ...
        }
...
```

The new diversity is available to be used by DiversityAgent clients. Register the created diversity on your DiversityAgent instance and request creation of new VMs.

```
...
        myAgent.createDiversity(new String[] {"HYPERVISOR"});
...
```

C.2 Creating new cloud drivers

We provide a modular structure to create new cloud drivers that is similar to the previously presented for diversities. The new driver must implement the methods existent in *Cloud* abstract class and at *createVm* method it should take care of all description properties already existent or registered by all diversities. To exemplify, we will explain how to create a new cloud driver for Amazon EC2 interface.

Create a *AmazonEC2* class that extends *Cloud* class in a *AmazonEC2.java* file and save this in drivers folder (*DiversityAgent/src/drivers*).

```
package diversities;
public class AmazonEC2 extends diversity_agent.Cloud {
        ...
}
```

Create the class constructor, where you will receive all information required to create a connection with the cloud provider. The first four (cloud name, tool, type and IP address) must be passed to the *Cloud* class constructor. There is an official API called *AWS SDK for Java*[1], provided by the Amazon Web Services, which could be used to help the creation of this cloud driver, similarly as the Java OCA was used on OpenNebula driver.

```
...
        public AmazonEC2(String theCloudName, String theTool, String theType, String
            theAddress, String theUsername, String thePassword)
        {
                super(theCloudName, theTool, theType, theAddress);
                //Create a client connection with the cloud provider fron end
                //If the connection succeed
                        //Then you can maintain the connection open in a global property
                        //or just save the authentication credetials to future
                            connections
                //Else
                        //Inform the client that was not possible to connect with cloud
                            provider and why
        }
...
```

Implement the *createVm* abstract method, which receives a property set and a *DataContainer* instance copy with DiversityAgent current state.

```
...
        public VirtualMachine createVm(Properties theProperties, DataContainer
            theDataContainer)
        {
                //Create a new diversity_agent.VirtualMachine instance
                //Set the cloud provider name on VirtualMachine instance
                //Set the VM image name on VirtualMachine instance
                //Prepare the template or request
                //Request the VM creation to the cloud provider
                //Get the VM identifier on cloud and set it on VirtualMachine instance
                //Wait the VM be running
                //Get the VM IP address and set it on VirtualMachine instance
                //Get the host name on cloud and set it on VirtualMachine instance
                //Increment the number of VMs running and created on this cloud instance
                //Return the VirtualMachine instance created
        }
...
```

[1] Available at http://aws.amazon.com/sdkforjava/. Accessed on March 19, 2012.

Such VM creation has to consider all properties supported in DiversityAgent version which you are using, even if there are properties not defined on property set received by argument. The next step is implement the *deleteVm* abstract method, which receives the VM identifier to be deleted on cloud provider.

```
...
        public boolean deleteVm(int theVmIdOnCloud)
        {
                //Request to the cloud provider delete the VM identified by the Id
                    passed as argument
                //Return the successfulness on deleting the VM
        }
...
```

You have to implement also the *readVm* method, which returns some VM metadata (passed as argument) from the cloud provider and the *getVmStatus* method, which returns the current VM status on cloud provider. After providing all methods required, create a tag and define it to instantiate your *AmazonEc2* class on *CloudFactory* source code. It is not a standard, but it can be defined in uppercase, for example, "AMAZON_EC2".

```
...
package diversity_agent;
...
public class CloudFactory {
        public Diversity getCloud(String theCloudName, String theTool, String theType,
            String theAddress, String theUsername, String thePassword) {
                Cloud aCloud = null;
                if(theTool.equals("OPEN_NEBULA")) {
                        ...
                } else if (theDiversity.equals("AMAZON_EC2")){
                        aCloud = new drivers.AmazonEC2(theCloudName, theTool, theType,
                            theAddress, theUsername, thePassword);
                }
                return aCloud;

        }
...
}
```

C.3 Publishing contributions

DiversityAgent is a free and open source software under GNU Lesser General Public License (LGPL v3.0), which means that it can be used even in proprietary software. We decided by this license to allow users freely choose which license they will provide their software and systems that uses DiversityAgent.

Modularising software since development beginning can contribute to provide easier extension and contribution to open source community. Feel free to produce your own diversity algorithms, properties and cloud drivers for DiversityAgent, and if you feel comfortable to spread your contribution, publishing it in official DiversityAgent page will be a pleasure.

Abbreviations

API	Application Programming Interface
AVI	Attack, Vulnerability and Intrusion
AWS	Amazon Web Services
BFT	Byzantine Fault Tolerance
COTS	Commercial Off-The-Shelf
COW	Copy-On-Write
CPU	Central Processing Unit
CRUD	Create, Read, Update and Delete
DBMS	Database Management Systems
DSA	Digital Signature Algorithm
FCT	Fundação para Ciência e Tecnologia
FIT	Fault and Intrusion Tolerance
FITCH	Fault and Intrusion Tolerant Cloud Computing Hardpan
FTP	File Transfer Protocol
GPS	Global Positioning System
IaaS	Infrastructure as a Service
IP	Internet Protocol
ISO	International Organization for Standardization
ISP	Internet Service Provider
JDK	Java Development Kit
LAN	Local Area Network
LaSIGE	Laboratório de Sistemas Informáticos de Larga Escala
LGPL	GNU Lesser General Public License
LRU	Least Recently Used
LTS	Long Term Support
MTBF	Mean Time Between Failures

MTTF Mean Time To Failure
MTTR Mean Time to Recover

NVD National Vulnerability Database

OCA OpenNebula Cloud API
OS Operating System
OTS Off-The-Shelf

PaaS Platform as a Service

RPC Remote Procedure Call
RSA Rivest/Shamir/Adleman algorythm
RTT Round Trip Time

SaaS Software as a Service
SDK Software Development Kit
SMR State Machine Replication
SSH SSH Secure Shell

TPM Trusted Platform Module

VM Virtual Machine
VMM Virtual Machine Monitor

WAN Wide Area Network

XML Extensible Markup Language

Bibliography

[1] Algirdas Avižienis. Design of fault-tolerant computers. In *Proceedings of the November 14-16, 1967, fall joint computer conference*, AFIPS '67 (Fall), pages 733–743, New York, NY, USA, 1967. ACM.

[2] Algirdas A. Avižienis. Fault-tolerance and fault-intolerance: Complementary approaches to reliable computing. *SIGPLAN Not.*, 10:458–464, Apr. 1975.

[3] Algirdas A. Avižienis and Liming Chen. On the implementation of N-version programming for software fault tolerance during execution. In *Proc. COMPSAC 77, 1st IEEE-CS Int. Comput. Software Appl. Conf.*, pages 149–155. IEEE, Nov. 1977.

[4] Alysson N. Bessani. From byzantine fault tolerance to intrusion tolerance (a position paper). In *Proceedings of the 5th Workshop on Recent Advances on Intrusion-Tolerant Systems (WRAITS)*, Hong Kong, China, Jun. 2011.

[5] Alysson N. Bessani, Miguel P. Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. DepSky: dependable and secure storage in a cloud-of-clouds. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 31–46, New York, NY, USA, Apr. 2011. ACM.

[6] Håvard K. Bjerke, Dimitar Shiyachki, Andreas Unterkircher, and Irfan Habib. Euro-par 2008 workshops - parallel processing. chapter Tools and Techniques for Managing Virtual Machine Images, pages 3–12. Springer-Verlag, Las Palmas de Gran Canaria, Spain, Aug. 2009.

[7] Miguel Castro and Barbara Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, USA, Oct. 2000.

[8] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20:398–461, Nov. 2002.

[9] Jimmy Clidaras, David W. Stiver, and William Hamburgen. Water-based data center, Aug. 2008.

[10] Vinicius V. Cogo and Marcelo Pasin. DiversityAgent: Automatic diversity in cloud computing, 2012. Available at http://code.google.com/p/diversity-agent/. Accessed on February 15, 2012.

[11] Robert R. Collins. The Pentium F00F bug. *Dr. Dobb's Journal of Software Tools*, 23(5):62, 64–66, May 1998.

[12] Miguel P. Correia, Nuno F. Neves, and Paulo E. Verissimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, SRDS '04, pages 174–183, Washington, DC, USA, 2004. IEEE Computer Society.

[13] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: a secretless framework for security through diversity. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, Berkeley, CA, USA, Jul. 2006. USENIX Association.

[14] Yves Deswarte, Karama Kanoun, and Jean-Claude Laprie. Diversity against accidental and deliberate faults. In *Computer Security, Dependability, and Assurance: From Needs to Solutions*, pages 171–181. IEEE Press, Nov. 1998.

[15] Tobias Distler and Rüdiger Kapitza. Increasing performance in byzantine fault-tolerant systems with on-demand replica consistency. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 91–106, New York, NY, USA, 2011. ACM.

[16] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004.

[17] Ilir Gashi, Peter Popov, and Lorenzo Strigini. Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers. *IEEE Trans. Dependable Secur. Comput.*, 4:280–294, Oct. 2007.

[18] Miguel G. T. Henriques. Diversity management in intrusion tolerant systems. Master's thesis, University of Lisbon, Lisbon, Portugal, 2011.

[19] Miguel G. T. Henriques, Alysson N. Bessani, Ilir Gashi, Nuno F. Neves, and Rafael R. Obelheiro. OS diversity for intrusion tolerance: Myth or reality? In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, Hong Kong, China, Jun. 2011.

[20] David Hilley. Cloud computing: A taxonomy of platform and infrastructure-level offerings. Technical report, College of Computing, Georgia Institute of Technology., Apr. 2009.

[21] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Softw. Eng.*, 12:96–109, January 1986.

[22] John C. Knight and Nancy G. Leveson. A reply to the criticisms of the knight & leveson experiment. *SIGSOFT Softw. Eng. Notes*, 15:24–35, January 1990.

[23] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, 41:45–58, Oct. 2007.

[24] Dionysius Lardner. Babbage's calculating engine. *Edinburgh Review*, 59(120):263–327, Jul. 1834.

[25] Soyini D. Liburd. An N-version electronic voting system. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004.

[26] James Martin. *Managing the Data Base Environment*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1983.

[27] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. In *Reports on Computer Systems Technology*. NIST, Sep. 2011. Special Publication 800-145.

[28] Rafael R. Obelheiro, Alysson N. Bessani, Lau C. Lung, and Miguel P. Correia. How practical are intrusion-tolerant distributed systems? Technical report, Department of Informatics, DI/FCUL, University of Lisbon, Sep. 2006.

[29] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22:299–319, Dec. 1990.

[30] Paulo Sousa, Alysson N. Bessani, and Rafael R. Obelheiro. The FOREVER service for fault/intrusion removal. In *Proceedings of the 2nd workshop on Recent advances on intrusiton-tolerant systems*, WRAITS '08, pages 5:1–5:6, New York, NY, USA, 2008. ACM.

[31] Paulo E. Veríssimo, Nuno F. Neves, and Miguel P. Correia. Architecting dependable systems. chapter Intrusion-tolerant architectures: concepts and design, pages 3–36. Springer-Verlag, Berlin, Heidelberg, 2003.

[32] Giuliana S. Veronese, Miguel P. correia, Alysson N. Bessani, and Lau C. Lung. EBAWA: Efficient Byzantine Agreement for Wide-Area Networks. In *Proceedings of the 2010 IEEE 12th International Symposium on High-Assurance Systems Engineering*, HASE '10, pages 10–19, San Jose, CA, USA, Nov. 2010.

[33] Giuliana S. Veronese, Miguel P. Correia, Alysson N. Bessani, Lau C. Lung, and Paulo E. Verissimo. Minimal byzantine fault tolerance : Algorithms and evaluation. Technical report, Department of Informatics, DI/FCUL, University of Lisbon, Jun. 2009.

[34] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. Zz and the art of practical bft execution. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 123–138, New York, NY, USA, 2011. ACM.