UNIVERSITÀ DEGLI STUDI DI ROMA "TOR VERGATA"

DOTTORATO DI RICERCA IN
Computer Science, Control and GeoInformation

CICLO DEL CORSO DI DOTTORATO
XXV

MOSES: a QoS-driven Autonomic Framework for Service Oriented Systems

Stefano Iannucci

A.A. 2013/14

Docente Guida/Tutor: Prof. Valeria Cardellini

Coordinatore: Prof. Giovanni Schiavon

I am convinced that I would never have been able to finish my Ph.D. studies without the support of my advisor Prof. Valeria Cardellini. Her exceptional culture, her character and her availability encouraged me to pursue this important goal. She is an absolute certainty.

This thesis is dedicated to my parents
and to my beautiful beans.
Thank you!

Abstract

Service Oriented Systems (SOSs) based on the SOA paradigm are becoming popular thanks to a widely deployed internetworking infrastructure. They are composed by a possibly large number of heterogeneous third-party subsystems and usually operate in a highly varying execution environment, that make challenging to provide applications with Quality of Service (QoS) guarantees. A well-established approach to face the heterogeneous and varying operating environment is to design a SOS as a runtime self-adaptable software system, so that a prospective enterprise willing to realize a SOA application can dynamically choose the component services that best fit its requirements and the environment in which the application operates. These SOSs are commonly architected as self-adaptive systems following the MAPE-K (Monitor, Analyze, Plan, Execute, and Knowledge) reference model for autonomic computing.

In this thesis we will first present a taxonomy of self-adaptive SOSs and then we will present the methodology, the design and architecture, and the performance evaluation of MOSES: a QoS-driven autonomic framework for service oriented systems. This framework, which is freely available with an opensource license at `http://uniroma2-moses.sourceforge.net`, fully implements the MAPE-K reference model by providing a platform for developing, testing and running different adaptation mechanisms exploited by autonomic SOSs. Specifically, we will focus on the methodologies at the core of the Plan phase that support QoS-driven adaptation. To this end, we will propose two policies that follow a different perspective in the request management, by optimizing either each single request or a flow of requests. The design and architecture of MOSES will be discussed according to the layered view of the Cloud Computing stack: we will start from the design and realization of an Infrastructure as a Service (IaaS) on which we deployed the Platform as a Service (PaaS) which acts as a container for MOSES, that in its turn resides at the Software as a Ser-

vice (SaaS) layer. Finally, we will present the results of an extensive performance evaluation, which takes into account both the effectiveness of the proposed QoS-driven adaptation methodologies and the scalability of the framework. Thanks to the modular architecture of MOSES, we are confident that its public release will allow the experimentation of alternative approaches to QoS-driven adaptation of Service Oriented Systems.

Publications

Parts of the work presented in this thesis have been published in the following book chapters, conference, journal and workshop papers.

Journals

J1 V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. Lo Presti, R. Mirandola, "MOSES: a framework for QoS driven runtime adaptation of service-oriented systems", *IEEE Transactions on Software Engineering*, Vol. 38, No. 5, pp. 1138-1159, Sept./Oct. 2012. doi: 10.1109/TSE.2011.68

J2. V. Cardellini, V. Di Valerio, V. Grassi, S. Iannucci, F. Lo Presti, "QoS driven per-request load-aware service selection in service oriented architectures", *International Journal of Software and Informatics*, Special Issue on Service Oriented Systems Engineering, Vol. 7, No. 2, pp. 195-220, 2013.

Book Chapters

BC1. V. Cardellini, V. Di Valerio, S. Iannucci, F. Lo Presti, "Service-oriented systems for adaptive management of service composition", *Adaptive Web Services for Modular and Reusable Software Development: Tactics and Solutions*, G. Ortiz and X. Cubo (eds.), IGI Global, pp. 161–195, 2013. doi: 10.4018/978-1-4666-2089-6.ch006

Proceedings of International Conferences and Workshops

IC1. V. Cardellini, S. Iannucci, "Designing a broker for QoS driven runtime adaptation of SOA applications", *Proceedings of the IEEE International Conference on Web Services (ICWS 2010)*, Applications and Industry Track, Miami, FL, pp. 504–511, July 2010. doi: 10.1109/ICWS.2010.77 (acceptance rate: 17.5%)

IC2. A. Bellucci, V. Cardellini, V. Di Valerio, S. Iannucci, "A scalable and highly available brokering service for SLA-based composite services", *Proceedings of the 8th International Conference on Service Oriented Computing (ICSOC 2010)*, San Francisco, CA, Lecture Notes in Computer Science Vol. 6470, Springer, pp. 527–541, Dec. 2010. doi: 10.1007/978-3-642-17358-5_36 (acceptance rate: $36/234 = 15.4\%$)

IC3. V. Cardellini, V. Di Valerio, V. Grassi, S. Iannucci, F. Lo Presti, "A performance comparison of QoS-driven service selection approaches", *Proceedings of the 4th European ServiceWave Conference (ServiceWave 2011)*, Poznam, Poland, Lecture Notes in Computer Science Vol. 6994, Springer, pp. 167–178, Oct. 2011. doi: 10.1007/978-3-642-24755-2_16

IC4. V. Cardellini, V. Di Valerio, V. Grassi, S. Iannucci, F. Lo Presti, "A new approach to QoS driven service selection in service oriented architectures", *Proceedings of the IEEE 6th International Symposium on Service-Oriented System Engineering (IEEE SOSE 2011)*, Irvine, CA, pp. 102-113, Dec. 2011. doi: 10.1109/SOSE.2011.6139098 (acceptance rate: 33%) **Best paper award**

IC5. V. Cardellini, S. Iannucci, "Designing a flexible and modular architecture for a private cloud: a case study", *Proceedings of the 6th International Workshop on Virtualization Technologies in Distributed Computing (VTDC 2012)* (in conjunction with the 21st International ACM Symposium on High-Performance Parallel and Distributed Computing), Delft, The Netherlands, pp. 37-44, June 2012. doi: 10.1145/2287056.2287067

IC6. E. Casalicchio, S. Iannucci, L. Silvestri, "Cloud Desktop Workload: a Character-

ization Study", *Proceedings of the IEEE 3rd International Conference on Cloud Engineering*, Tempe, AZ, March 2015.

Posters and National Conferences

P1. V. Cardellini, S. Iannucci, "Improving SOA applications response time with service overload detection", *21st International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC 2012)*, poster presentation, Delft, The Netherlands, June 2012.

NC1. A. Bellucci, V. Cardellini, V. Di Valerio, S. Iannucci, "A scalable and dependable system for QoS-aware SOA applications", *Workshop Informatica Quantitativa 2010 (InfQ 2010)*, Pisa, July 2010.

NC2. S. Iannucci, V. Cardellini, "Designing a system architecture for a private cloud provider", *V Conferenza Italiana sul Software Libero (ConfSL 2011)*, Milano, June 2011.

NC3. E. Casalicchio, S. Iannucci, L. Silvestri, "Characterization of CPU and disk load for Cloud Desktop Providers" *Workshop Informatica Quantitativa 2014 (InfQ 2014)*, Torino, November 2014.

Under preparation

UP1 V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. Lo Presti, R. Mirandola, "MOSES: a platform for experimenting QoS-driven self-adaptation policies for service oriented systems", book chapter in *Software Engineering for Self-Adaptive Systems: Assurances*, LNCS, Springer, 2015.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Definition 6

1.3 Key Research Issues 8

1.4 Contributions 9

1.5 Thesis Outline 11

2 Autonomic Service Oriented Systems **13**

2.1 SOA Fundamentals 14

2.2 Fundamentals of Autonomic Systems 16

2.3 Dimensions of Self-Adaptation for Service Oriented Systems 18

2.4 Dimensions of Self-Adaptation for MAPE-K based Service Oriented Systems 24

2.4.1 Monitor Taxonomy 24

2.4.2 Analyze Taxonomy 27

2.4.3 Plan Taxonomy 30

2.4.4 Execute Taxonomy 38

3 MOSES: a Framework for QoS Driven Runtime Adaptation of Service-Oriented Systems **43**

3.1 Overview of MOSES 44

3.1.1 A Comparison of Frameworks for Self-Adaptation of SOSs . 45

3.2 Plan Phase 51

3.2.1 Composite Service Model 52

3.2.2 Adaptation Actions 55

3.2.2.1 Adaptation Actions for Stateless and Stateful Services 57
3.2.3 SLA Model 58
3.2.4 Adaptation Policy Model 59
3.2.5 QoS Model 60
3.2.5.1 Task QoS Attributes 62
3.2.6 Optimization Policies 64
3.2.6.1 Per-Request Optimization 64
3.2.6.2 Per-Flow Optimization 68
3.2.6.3 Load-Aware Per-Request Optimization 75
3.3 Monitor Phase 82
3.3.1 Monitoring the Concrete Services 83
3.4 Analyze Phase 83

4 Case Study: Design and Implementation of MOSES 89
4.1 IaaS Layer 90
4.1.1 Back-End Subnet 91
4.1.1.1 Redundant Network Topology 91
4.1.1.2 Redundant Storage 94
4.1.1.3 Volumes Management 97
4.1.1.4 Complete Storage Architecture 98
4.1.2 Choosing the IaaS Management Platform 101
4.1.3 Front-End Subnet 104
4.2 PaaS Layer 105
4.3 SaaS Layer 107
4.3.1 Overview of the MOSES Architecture 107

4.3.2 MOSES Design within OpenESB 111
4.3.3 MOSES Components 113
4.3.4 MOSES Clustered Architecture 114
4.3.5 MOSES Overheads 116

5 Performance Evaluation 119

5.1 Testing Environment 121
5.2 Workload Generator 121
5.3 Performance Evaluation 123
5.3.1 Runtime Binding Overhead Analysis 123
5.3.2 Performance of MOSES ESB Clustered 125
5.4 Effectiveness Evaluation 126
5.4.1 Effectiveness of Per-Flow Adaptation Policy 126
5.4.1.1 Maximization of the Average Reliability 129
5.4.1.2 Minimization of the Average Cost 131
5.4.1.3 Adaptation Policy Computational Cost 133
5.4.2 Improving Reliability through Web Service Monitoring 136
5.4.2.1 Baseline Scenario 138
5.4.2.2 Graceful Failure and Join of Concrete Services . . . 140
5.4.2.3 Undetected Failure/Join of Web Services 141
5.4.2.4 Detection of Web Service Failures to Improve Reliability 142
5.4.3 Comparison of Per-Request and Per-Flow Approaches 142
5.4.3.1 Per-Request and Per-Flow Optimization Time Comparison 145

5.4.3.2 Per-Request and Per-Flow Execution Time Comparison 146
5.4.4 Comparison of Per-Request and Load-Aware Per-Request Approaches 151
5.4.4.1 Per-Request Approach 154
5.4.4.2 Comparison between Per-Request and Load-Aware Per-Request Approaches 155
5.4.4.3 Scalability of Load-Aware Per-Request Approach . 157
5.4.4.4 Effectiveness of Load-Aware Per-Request Approach 157
5.4.4.5 Effectiveness of the Adaptive Cusum Algorithm . . 159

6 Conclusions **167**
6.1 Summary 167
6.2 Future Work 169

List of Tables

3.1 Main notation adopted in the thesis. 53
3.2 Workflow composition rules. 54
3.3 Coordination patterns. 56
3.4 Recursive rules to calculate the average value of the QoS attributes of a composite service according to the per-flow workflow model. 71

5.1 Operation SLA parameters. 128
5.2 Class SLA parameters. 129
5.3 Measured values for SLA parameters (mean and 95% confidence interval). 132
5.4 Operation SLA parameters. 139
5.5 Average reliability and 95% confidence interval for the baseline experiment 139
5.6 Average reliability and 95% confidence interval for experiment with graceful failures 140
5.7 Comparison of the average reliability and 95% confidence interval for the experiments with and without the WS Monitor 143
5.8 SLA parameters for candidate operations (top) and service classes (bottom) 148
5.9 Performance comparison with the per-flow approach of [11] and per-request approaches of [6, 12] (time measured in seconds). 149
5.10 SLA parameters for candidate operations 152
5.11 SLA parameters for service classes 153
5.12 Average response times of the load-aware per-request policy for all service classes under light and heavy loads 158

List of Figures

2.1 Workflow of the composite service managed by MOSES 15
2.2 MAPE-K control loop. 17
2.3 Taxonomy of self-adaptation for SOA. 19
2.4 Conceptual model of the SOA domain. 21
2.5 Monitor taxonomy. 25
2.6 Analyze taxonomy. 28
2.7 Plan taxonomy. 31
2.8 Execute taxonomy. 38

3.1 The MOSES approach. 45
3.2 MOSES within the self-adaptable SOS taxonomy. 46
3.3 MOSES within the Plan taxonomy 51
3.4 A MOSES-compliant workflow. 54
3.5 Implementation of the MOSES adaptation policy for a single task. . . 60
3.6 Composite service labeled tree. 70
3.7 MOSES within the monitor taxonomy 86
3.8 MOSES within the analyze taxonomy 87

4.1 Network topology. 93
4.2 Data flow on the storage architecture. 100
4.3 Network stack of the front-end servers. 105
4.4 MOSES high-level architecture. 108
4.5 Typical execution flow in the ESB-based MOSES prototype. 111
4.6 MOSES clustered architecture. 115

5.1 Workflow of the composite service managed by MOSES 122

5.2 MOSES response time. 124
5.3 Throughput in the closed model. 126
5.4 Response time in the open model. 127
5.5 ($w_d = 1$): reliability for all classes over time. 130
5.6 ($w_d = 1$): response time for all classes over time. 131
5.7 ($w_c = 1$): reliability for all classes over time. 133
5.8 Optimization problem execution time for different values of maximal redundancy: (a) no redundancy ($p = 0$); (b) at most two concrete services using the *par_or* pattern ($p = 2$) and (c) at most three concrete services using the *par_or* pattern ($p = 3$). 134
5.9 Optimization problem execution time as function of the number of service classes. 136
5.10 Baseline reliability over time under low request rate 140
5.11 Baseline reliability over time under high request rate 141
5.12 Reliability over time when services are subject to graceful failures under low request rate 142
5.13 Reliability over time when services are subject to graceful failures under high request rate 143
5.14 Reliability over time when services are subject to failures, without WS Monitor under low request rate 144
5.15 Reliability over time when services are subject to failures, without WS Monitor under high request rate 145
5.16 Reliability over time when services are subject to failures, with WS Monitor under low request rate 146

5.17 Reliability over time when services are subject to failures, with WS Monitor under high request rate 147

5.18 Scenario 1: response time of the composite service for class 1 160

5.19 Scenario 2: response time of the composite service over time for class 1 161

5.20 Response time of the traditional per-request service selection policy . 162

5.21 Response time of the traditional versus load-aware per-request service selection policies 162

5.22 CPU usage of the concrete service selected for S_1 by the traditional per-request policy 163

5.23 CPU usage of the concrete services selected for S_1 by the load-aware per-request policy 163

5.24 Response time of the load-aware per-request policy under the two sets of concrete services 164

5.25 Response time of the load-aware per-request policy for all service classes over time 165

5.26 Response time of the load-aware per-request policy under external load without QoS Monitor 166

5.27 Response time of the load-aware per-request policy under external load with QoS Monitor 166

1 Introduction

Contents

1.1 Motivation **1**
1.2 Problem Definition **6**
1.3 Key Research Issues **8**
1.4 Contributions **9**
1.5 Thesis Outline **11**

1.1 Motivation

In computer science, Service-Oriented Architecture (SOA) is now a mature reference paradigm for developing network accessible, service-based applications. The main goal of designing applications following the SOA paradigm is to achieve a better degree of interoperability with respect to legacy distributed applications, which are tied up by constraints, such as programming languages and specific protocols and technologies. SOA applications are built up by composing black-box services that can be discovered and invoked using standard protocols, therefore hiding possibly different technologies. The service composition is usually described by a workflow representing the actual business logic of the application, defining both the execution and data flow. SOA applications have the clear advantage over legacy applications to be easily reused because they can be published as services in a standard registry, where other

applications can discover them for further invocation. As a consequence, the focus in developing a SOA application is shifted to activities concerning the identification, selection, and composition of services offered by third parties rather than the classic in-house development. Systems realized using the SOA paradigm take the name of Service Oriented Systems (SOSs). They benefit from the SOA flexibility as well as from the presence of a widely deployed internetworking infrastructure.

The diffusion of systems deployed using the SOA paradigm is leading to the proliferation of service marketplaces (such as SAP Service Marketplace and Windows Azure Marketplace), where an enterprise can find every component needed to build its SOA applications. With an ever increasing number of service providers on the global market scene, it is becoming easy to find multiple providers implementing the same functionality with different quality levels, e.g., different providers can exhibit different response times or costs for services that present the same logic. Therefore, depending on the needs of the SOA application, it is possible to dynamically select the services that best fit its (possibly changing) requirements. However, several problems arise when a SOA application, which is offered using third party services, needs to fulfill non-functional requirements, because existing services may disappear or their performance may quickly fluctuate over time, due to the highly varying execution environment.

The SOA paradigm easily allows to replace services with equivalent ones, but this task could be very challenging for a human being, especially when several services must be replaced at the same time. Similarly, when the service composition logic needs to be partially or even entirely modified in order to account for changes in the functional requirements, it is hard to manually choose among several alternative workflows, considering also the non-functional requirements. In addition, the management

complexity of SOSs rapidly grows as the number of services involved in the compositions increases. To tackle such complexity, to reduce management costs, and to provide better operativeness, a common and well-established approach is to design SOSs as runtime self-adaptable software systems [91], that is, software systems able to detect changes in the environment and to properly reconfigure themselves. In the field of self-adaptable software systems, the main research branches that have been pursued regard the functional and non-functional requirements of SOA applications. The former concern the overall application logic to be implemented, while the latter concern the Quality of Service (QoS) levels that should be guaranteed. In this thesis, we focus on non-functional requirements expressed as QoS attributes of SOSs.

The adaptation of non-functional requirements can follow either the best effort or QoS-driven strategy. The former aims to generally improve non-functional attributes (e.g., response time or reliability) of the overall SOA application, but without ensuring any kind of guarantee. On the other hand, the latter aims to provide a SOA application with predictable QoS attributes. In the last years both approaches have been largely investigated, e.g., [42, 59, 70] for the best effort strategy and [12, 23, 65] for the QoS-driven strategy. Each solution has its own characteristics and peculiarities in the way it faces the self-adaptation. In particular, since in the context of SOA applications, the management, the control, and performance prediction of the QoS characteristics of the offered service have been identified as the most critical tasks as they ultimately determine how the system guarantees QoS levels, most of the efforts have focused on and mostly differ for the different strategies adopted for the aforementioned tasks. Nevertheless, despite their differences, all these approaches follow a more general framework, called MAPE-K.

MAPE-K [52] is a conceptual guideline for realizing self-adaptable systems [91] and is composed of four essential phases: Monitor, Analyze, Plan, and Execute. There is also a Knowledge layer that support all the phases. The model is based on a feedback-control loop, that detects changes in the execution environment, analyzes them, plans the necessary actions to optimize some utility function, and executes these actions. In literature, the same approach is also referred to as CADA [39], which stands for Collect, Analyze, Design, and Act.

It is generally possible to distinguish between two families of adaptation strategies, according to the granularity level they provide to manage the requests to the SOS. With the *per-request* grain, the adaptation concerns a single request addressed to a composite service, and aims at making the system able to fulfill the QoS requirements of that request, independently of the concurrent requests that may be addressed to the system. With the *per-flow* grain, the adaptation concerns an overall flow of requests, and aims at fulfilling QoS requirements concerning the global properties of that flow. Most of the proposed methodologies focus on the per-request case (e.g. [6, 12, 22, 104, 105]), while only a few works have focused on the per-flow granularity (e.g. [11, 23, 24]). However, both the approaches have been formalized as optimal service selection problems. An optimal service selection problem aims at finding the optimal combination of services needed to implement the required abstract functionalities while satisfying the QoS constraints.

To be able to scale and to optimize resource utilization, SOSs usually exploit infrastructures realized with the Cloud Computing [13, 20], which has recently emerged as a paradigm for delivering computational services over the Internet. Small and medium organizations are attracted by this computing paradigm because it let them not own

an in-house datacenter with the associated risks and costs, but rather just rent what is effectively needed time after time.

There are different types of cloud deployment models, each with its own benefits and drawbacks. With *public clouds*, providers offer their resources as services to the general public. Key benefits of using public clouds include no initial capital investment on the infrastructure and risk shifting to cloud providers. *Private clouds* are instead designed for exclusive use by a single organization. They may be built by the organization itself or by an external service provider. A private cloud offers the highest degree of control over performance, reliability, and security. However, it is often criticized for being similar to traditional proprietary server farms and does not provide benefits such as no up-front capital costs. Finally, *hybrid clouds* are in between public and private: they are primarily based on private clouds, but they can extend their capacity with public clouds should the need arise.

Cloud providers offer services at three different layers, respectively named: *Infrastructure as a Service (IaaS)*, *Platform as a Service (PaaS)*, *Software as a Service (SaaS)* [106]. These three levels offer a layered view of the cloud computing stack. The user of the lowest layer rents from a IaaS provider virtualized resources like computational power, storage, and network and has the task to manage the rented resources. The middle layer is the PaaS, in which the cloud user acquires the control over a software platform, ideally an application server and an application development environment, where he has to deploy and manage his own applications. At the uppermost layer, i.e., the SaaS, the cloud provider offers software applications to its users which might be, of course, SOSs.

1.2 Problem Definition

In this thesis we address the challenging problem of designing and developing a QoS-aware *autonomic* broker of SOS.

We suppose to operate in an environment where multiple service providers offer possible functionally equivalent services with different QoS attributes. Such QoS attributes are formalized by providers into contracts named Service Level Agreements (SLAs). Each SLA contains both the quality parameters (such as response time and reliability) and the cost that will be charged for each service invocation. The broker, which is in charge of executing a QoS-aware service workflow, accepts the SLAs with service providers and, in turn, proposes its own SLAs to perspective users. The main task of the broker is therefore to satisfy the SLA established with the users for the entire workflow, given the ones bargained with the service providers of the single services and the operating environment. Furthermore, since the broker will operate over the Internet, it might face several issues such as service churn or performance fluctuation with or without consequent SLA violation. It must therefore be able to adapt the workflow it serves according to such environmental changes. In particular, it must be able:

- to *self-optimize* service selection according to the required SLA;
- to *self-configure* its components according to a possibly changing environment (for example, it must be able to scale up and down according to the submitted load in order to avoid itself to be a bottleneck);
- to *self-heal* its components and the workflow it serves in case of failures.

These three aspects (along with the *self-security* aspect which will not be addressed by this thesis) constitute the foundations of an autonomic system.

Since there are possibly tenths or hundreds of concrete services that might be used to implement each task of the workflow, the ability to self-optimize is crucial for a broker of SOSs. Self-optimizing means in this case selecting the *best* implementing services, given the current environment. This is a particularly tricky task due to the definition of *best*: there could be different optimization goals, possibly conflicting. For instance, one goal of the broker could be to maximize its revenue; however, to fulfill the SLAs agreed with the users, it could be forced not to choose the cheapest services.

Self-configuration is a core requirement for the broker. Since there are many components working together to realize the final application, it is not arguable to let a human being manage all this complexity. Furthermore, the system is exposed to Internet and may be subject to sudden load burst and therefore it must be able to easily scale according to the current load. This is a challenging task because both the broker and the infrastructure where it resides must be designed and implemented in order to support automatic component deployment and configuration.

Due to the high number of component involved, it is also fundamental to consider that some of them could fail over time. Self-healing feature provides the broker with the ability to detect such a failure and to consequently react. This task is really difficult because there are different kinds of failures to address. For instance, a component might completely fail or it could violate its SLA. In the first case we have no choice and the auto-healing behavior of the broker must detect the problem and replace the failed component with a functionally equivalent one, but in the second case this rarely is an easy decision: the alternative services might be more expensive or might not provide the required QoS attributes. It is therefore needed to use sophisticated analyze methods in order to actually state that the component should be replaced.

1.3 Key Research Issues

The problems identified in Section 1.2 produce the key research issues summarized in the following.

Service selection The service selection problem has been widely investigated in recent years. We can distinguish among three generations of service selection solutions. First generation of service selection solutions focused on a *local* approach [61,105] that time by time associates each running task of a SOS with the best available service that implements that task. However, this local approach can guarantee only local QoS constraints, for example the response time of a given task lower than a given threshold and did not solve the problem of having a full workflow compliant with QoS constraints.

To overcome this limitation, second generation service selection solutions shifted the focus on a *global* approach, introducing the concept of granularity level of the adaptation: at the *per-request* grain [6, 12, 22, 60, 65, 104], the adaptation focuses on each single request submitted to the system and aims at fulfilling the QoS constraints of that specific request. On the contrary, the *per-flow* grain [11, 23, 24, 54] considers the flow of requests of a user rather than the single request, and the adaptation goal is to fulfill the QoS constraints that concern the global properties of that flow, *e.g.*, the average SOS response time or its reliability.

However, the solutions proposed so far for both per-request and per-flow granularities are not satisfactory, either in terms of QoS guarantees or scalability to user requests. The per-request grain exhibits scalability problem under a sustained traffic of requests, because each request is managed independently of all the other concurrent ones. As a consequence, multiple service requests could be assigned to the same concrete service,

that could be overloaded. On the other hand, the per-flow grain is not able to ensure QoS guarantees to a single request, and the user perceived QoS could be very different from that stipulated in the SLA.

Design, implementation and performance testing of autonomic SOSs broker Currently there exist some implementations of frameworks for QoS brokering of Web services (e.g., [10, 22, 69]). Menascé et al. have proposed a SOA-based broker for negotiating QoS goals [69] but their broker does not offer a composite service and its components are not organized as a self-adaptive system. PAWS [10] is a framework for flexible and adaptive execution of business processes but some of its modules work only at design time. Proxy-based approaches for the runtime binding to concrete services, have been previously proposed, either for re-binding purposes [22] or for handling runtime failures in composite services as in the TRAP/BPEL framework [43]. The SASSY framework for self-adaptive SOSs has been proposed in [66]. It self-architects at runtime a SOS to optimize a system utility function. Nonetheless, none of the works in the SOA field has evaluated the proposed prototype in terms of performance and scalability. We believe that such kind of evaluation is needed for any prototype to be adopted and developed in an industrial environment.

1.4 Contributions

In the following we summarize the main contributions of this thesis.

Contribution 1. Performance-oriented design of an autonomic SOSs broker This contribution regards the design and implementation of MOSES: an autonomic SOSs broker. Many effort have been conducted in order to realize an efficient broker with

very small overheads with respect to the execution of a SOS with static bindings. We designed and implemented every aspect of the system: from the underlying private cloud infrastructure to its the modular architecture based on Enterprise Service Bus (ESB). Furthermore, we demonstrated how it is possible to realize such a complex system by using exclusively free software.

Contribution 2. A third generation service selection solution We introduce with this thesis a third generation service selection solution which overcomes the limits of existing approaches described in Section 1.3. Specifically, we propose a per-request service selection solution which is able to balance the load across the available concrete services: we borrow from the second generation per-request approach the ability to guarantee QoS levels for each single request, but we also borrow from the per-flow approach the ability to scale over multiple concrete services. We also contributed to extend the existing second generation per-flow approach presented in [24, 25].

Contribution 3. Extensive effectiveness and performance testing This contribution regards an extensive test of the performance and effectiveness of MOSES. Performance tests were conducted in order to show the actual overhead introduced by the runtime binding as well as the scalability features of the prototype. Thanks to the modular architecture of MOSES we were able to implement and compare several existing service selection solutions: for each implementation, effectiveness tests have been conducted to prove its benefits and limits. Finally, we also implemented and evaluated the new proposed per-request third generation service selection solution, in comparison with the second generation one. At the state of the art, none of the existing brokers [66, 70, 89] with the exception of MOSES have been evaluated both from the

point of view of the effectiveness and of the performance, therefore it is unknown if they can be adopted in a production environment.

Contribution 4. Release to the community of a framework for self-adaptive SOSs As a last contribution, we release to the community the source code of MOSES. We believe that this will allow the experimentation of alternative approaches to QoS-driven adaptation of Service Oriented Systems. To the best of our knowledge, besides MOSES, the only other broker whose source code is available is VRESCo [70], but it lacks many features available in MOSES, as described in Section 3.1.1.

1.5 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 provides background information about the SOA and autonomic computing domains and introduces two taxonomies for autonomic Service Oriented Systems. The taxonomies are then used to classify several related works; Chapter 3, which covers Contribution 2, describes the MOSES methodology according to the phases of the MAPE-K reference model for autonomic systems; Chapter 4, which covers Contribution 1, describes the design of MOSES, as well as the design of a private cloud infrastructure. In Chapter 5, which covers Contribution 3, we show the results of the performance and effectiveness tests. Finally, Chapter 6 concludes the thesis with a reflection on the contributions and outlook for future work.

2

Autonomic Service Oriented Systems

Contents

2.1 SOA Fundamentals 14
2.2 Fundamentals of Autonomic Systems 16
2.3 Dimensions of Self-Adaptation for Service Oriented Systems . . 18
2.4 Dimensions of Self-Adaptation for MAPE-K based Service Oriented Systems 24
2.4.1 Monitor Taxonomy 24
2.4.2 Analyze Taxonomy 27
2.4.3 Plan Taxonomy 30
2.4.4 Execute Taxonomy 38

In this chapter we will discuss about autonomic systems that use a service orientation approach. First of all, we will introduce the fundamental concepts behind Service Oriented Architecture (SOA) and then the MAPE-K reference framework [52] for building autonomic systems. As major contribution for this chapter, we will provide a characterization of the problem space for self-adaptive software systems by organizing it along several *dimensions*, where each dimension captures one or more related facets of the problem. Papers addressing this issue have provided somewhat different characterizations [8, 19, 34, 48, 64, 87, 91], mainly because of some difference in the adopted perspective. Overall, they can be considered as possible answers to some basic questions [91]:

- *why* the adaptation should be performed (which are its goals);
- *when* should adaptation actions be applied;
- *where* the adaptation should occur (in which part of the system) and *what* elements should be changed;
- *how* should adaptation be implemented (by means of which actions);
- *who* should be involved in the adaptation process.

The answers provided by the papers cited above aim at addressing the whole software systems domain. In this chapter we adopt a narrower viewpoint, and propose two taxonomies that provide possible answers to these questions according to:

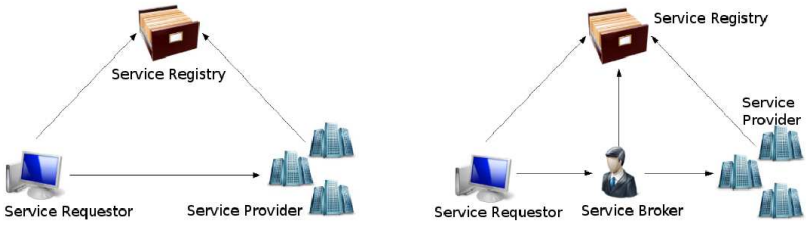
- the specific features of the SOA domain with special emphasis on QoS aspects and
- the specific adaptation methodology provided by the MAPE-K framework.

Our main goal is to analyze the key issues to be tackled in the design of an autonomic SOS while providing specific references to related works for all the considered aspects of the taxonomies. We do not aim at presenting an exhaustive analysis of the literature for the SOA domain, for which we refer to [50, 95, 97].

2.1 SOA Fundamentals

The SOA reference model defines the interacting actors and their interaction modes. Looking over the SOA domain, the main actors are: the *service provider*, that offers

a service, and the *service requestor*, that requests the service. To issue a service invocation, the service requestor has to know a service provider offering the needed functionality. To this end, the service registry holds information about existing services, which are published by service providers themselves. Figure 2.1a illustrates the SOA reference model.



(a) SOA reference model (b) SOA reference model with broker

Figure 2.1: Workflow of the composite service managed by MOSES

One of the peculiarities of SOA is that is possible to assemble services offered by several service providers in order to build a *service composition*. The composition itself can be in its turn exposed as a service and used by other service compositions.

As an example of service composition, consider a prospective travel planner application: it could be composed by several steps such as one or more hotel reservations, flights reservations, credit card checks and so on. This application follows a flow of activities described by a business process and every step could involve an invocation to a different service provider.

The aforementioned business process can be described either using a *service orchestration*, or a *service choreography* paradigm. The former is shown in Figure 2.1b, where a centralized entity named *service broker* is responsible for the execution of the business logic and for the invocation of the component services. That is, the totality of

the business logic is held by the broker and the component services are completely unaware of each other. With the latter approach, instead, the business logic is distributed across the component services and there is no central coordination. Our focus in this thesis is on service orchestration.

A common implementation of the SOA reference model is realized by Web services. In this thesis we will therefore use the terms service and Web service interchangeably.

2.2 Fundamentals of Autonomic Systems

MAPE-K is an architectural framework for realizing self-adaptable applications and, in a more general way, it can be used to build autonomic applications: the MAPE-K control loop uses an intelligent agent to perceive the surrounding environment through sensors and uses the collected information to determine the actions that have to be performed on the environment itself. In the context of SOA applications, the managed environment is constituted by (i) a workflow of activities concerning the invocation of external services, (ii) the external services, and (iii) the network interconnecting these activities, the clients and the service providers. The managers are software components which belong to the different MAPE-K phases, namely (i) Monitor, (ii) Analyze, (iii) Plan, and (iv) Execute.

Figure 2.2 illustrates the MAPE-K control loop: the four steps of the autonomic manager, the managed element, the sensors, and the actuators. In the context of SOA the managed element is the SOA application itself, while the autonomic manager is a (possible complex) software layer supervising the actual SOA application. While the application runs, the manager goes through the different MAPE-K steps:

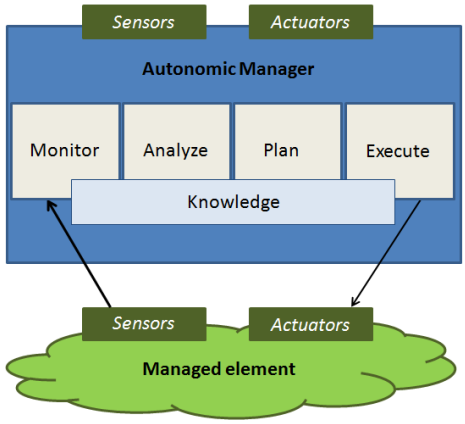


Figure 2.2: MAPE-K control loop.

1. Monitor: this phase monitors the application execution through sensors. In the SOA context, the sensors are implemented by means of probes over external services, with the objective of detecting new services as well as the actual quality attributes like response time, reliability, availability of already known services.

2. Analyze: the Monitor phase output is taken as input by the Analyze phase, which usually performs statistical computation on the raw data collected by the preceding phase. The data analysis has the objective to decide whether some quality attribute has violated (or is going to violate) a previously specified internal policy, usually stated into the Knowledge layer. In the SOA domain, an internal policy could be the violation of a certain threshold on a quality attribute, e.g., the average detected response time for a given service is greater than what established in the internal policy, or its reliability is less than what expected.

3. Plan: when the Analyze phase detects some kind of violation of the internal

policies, the Plan phase is activated and an adaptation plan is computed, possibly using the data elaborated by the Analyze phase together with the Knowledge layer. In the SOA context, the elaboration of a new execution plan could be represented by a different service selection, that is, the selection of different service providers implementing the needed functionalities. Otherwise, it could be represented by an internal workflow re-arrangement so that internal policies specifying the application requirements could be satisfied.

4. Execute: the new computed plan has to be executed by the SOA application controlled by the MAPE-K control loop. Such corrective actions are applied by means of actuators on the underlying SOA application. In the SOA domain, the corrective actions could be represented by a different binding of functionalities to service providers, as well as by application re-deployments.

2.3 Dimensions of Self-Adaptation for Service Oriented Systems

Figure 2.3 summarizes the main concepts of this characterization. For the sake of clarity, the class diagram in Figure 2.4 illustrates some elements of the SOA domain we use in this characterization. A more detailed taxonomy of these elements can be found, for example, in [18, 36].

Why. The basic goal of adaptation is to make the system able to fulfill its *functional* and/or *non functional* requirements, despite variations in its operating environment, which are very likely to occur in the SOA domain. Our focus in this thesis is on non functional requirements concerning the delivered QoS and cost. In the SOA domain, these requirements are usually the result of a negotiation process engaged between

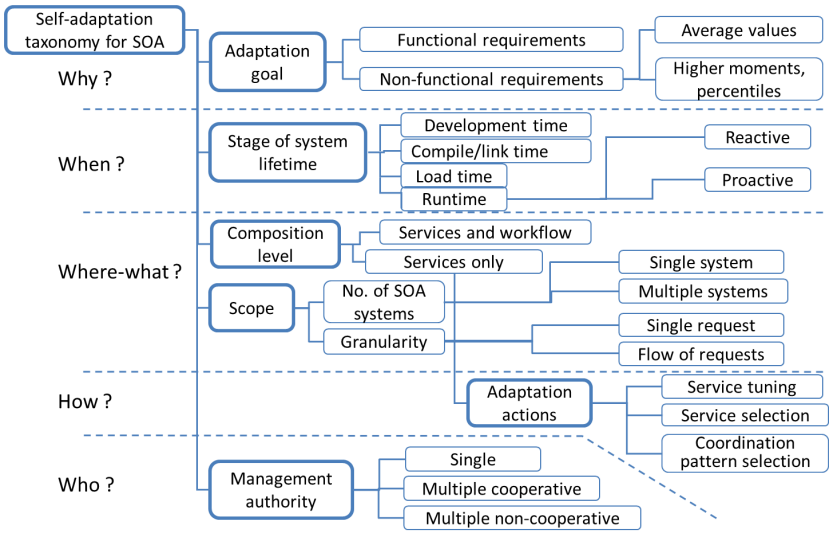


Figure 2.3: Taxonomy of self-adaptation for SOA.

the service provider and user, which culminates in the definition of a *Service Level Agreement* (SLA) concerning their respective obligations and expectations [67]. In a stochastic setting, a SLA specifies guarantees about the *average value* of quality attributes, or more tough guarantees about the *higher moments* or *percentiles* of these attributes [26, 92].

A completely different approach is described in [58], where the goal of the adaptation is not to optimize the QoS, rather to minimize the cost resulting from SLA violations, which can occur if financially desirable.

With regard to functional requirements, we just mention that, in the SOA domain, adaptation may play a relevant role in tackling runtime interoperability issues among dynamically discovered and selected services (e.g., [35, 73]).

When. Broadly speaking, adaptation can be performed at different stages of the system lifetime [64]: development time, compile/link time, load time, runtime. In the SOA domain, the emphasis is on building systems by late composition of running services. Hence, the focus of adaptation in this domain is on the *runtime stage*. This narrower viewpoint of the "when" dimension is also adopted in [91] for the broader field of self-adaptive software. Within this stage, we may further distinguish *reactive* and *proactive* adaptation. In the reactive mode, the system adapts itself after a change has been detected. In the proactive mode, the system anticipates the adaptation based on a prediction of possible future changes.

Where-What. The SOA paradigm emphasizes a compositional approach to software systems development, where the units of composition are services. A service can be considered as a black-box component deployed on some platform, operated by an independent authority and made accessible through some networking infrastructure using standard protocols. Hence, the composition of services can be considered as the basic *locus* for adaptation in the SOA domain. Looking at service composition, we may distinguish an *abstract composition*, where only the required functionalities (*tasks*) and their composition logic are specified, and a *concrete composition*, where the tasks of an abstract composition are bound to actual implementations, based on the use of *operations* offered by network accessible *concrete services*. Based on this distinction, adaptation in the SOA domain may take place at two different levels:

- *services only*: the adaptation only involves the concrete composition, acting on the implementation each task is bound to, leaving unchanged the composition logic (*i.e.*, the overall abstract composition) [12, 23, 24, 54, 77];
- *services and workflow*: the adaptation involves both the concrete and abstract

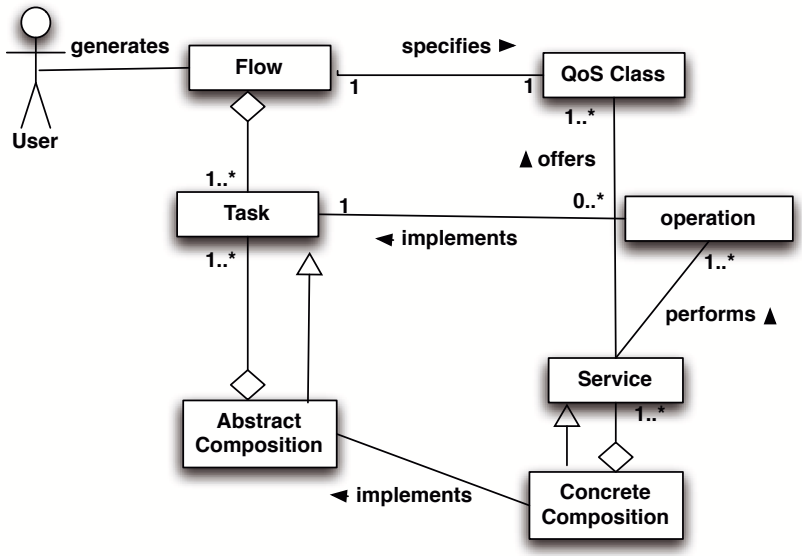


Figure 2.4: Conceptual model of the SOA domain.

composition; in particular, the composition logic can be altered. Several approaches have been proposed that use aspect oriented programming to inject fragments of code (e.g. [5, 59]) or that provide extensions to orchestration languages [32, 56].

We may also look at the *where* question from the perspective of the adaptation *scope*. In this perspective, we may take two different viewpoints: the *number* of SOSs operating in the same environment that are directly involved in the adaptation process, and the *granularity* level at which adaptation is performed, considering the flow of requests addressed to a SOS by the same or different users.

We first discuss this issue from the "granularity level" viewpoint in the "scope" dimension:

- *single request*: the adaptation concerns a single service request, and aims at making the system able to fulfill the requirements of that request, irrespective of whether it belongs to some flow generated by one or more users [6, 12, 22, 104, 105]);

- *flow of requests*: the adaptation concerns an overall flow of requests, and aims at fulfilling requirements concerning the global properties of that flow [11, 23, 24].

Let us consider now the "number of SOA systems" viewpoint:

- *single system*: a single system is explicitly considered as the system to be adapted, while everything else, including other competing SOA systems, is considered part of its environment;

- *multiple systems*: several SOA systems, competing for overlapping sets of services in the same environment, are explicitly considered in the adaptation process.

How. Possible answers to this question depend on the level of the composition where adaptation takes place, as discussed above. For adaptations involving only the services of the composition, adaptation actions could be based on:

- *service tuning*: the behavior and/or properties of the operations of concrete services are changed, depending on the current operating conditions, exploiting some management interface exposed by the concrete services themselves (*e.g.*, based on WSDM MOWS [63]). This kind of action does not change the current binding between tasks and operations of concrete services;

- *service selection*: the goal of this action is to identify and to bind to each task a corresponding *single* operation offered by a concrete service, selecting it from a set of candidates. This kind of action could change the binding between tasks and operations, if the previous selection is no longer suitable for the new operating conditions; most of the approaches in literature focus on service selection [50]. Related works about service selection will be presented in Section 2.4.3.

- *coordination pattern selection*: rather than binding each task to a single operation, this action binds it to a *set* of functionally equivalent operations offered by different concrete services, coordinating them according to some spatial or temporal redundancy pattern. The coordination pattern is selected within a set of implementable patterns (e.g., 1-out-of-n parallel redundancy, alternate service [25]), that could in general guarantee different QoS and cost levels, for the same set of coordinated operations. Binding a task to a set of equivalent operations allows to obtain QoS levels (concerning reliability and, in some cases, performance) that could not be achievable binding it to a single operation. Of course this advantage should be weighted against the higher cost caused by the use of multiple concrete services.

Who. This dimension concerns the "authorities" that manage the adaptation process and it is related to the "number of SOA systems" dimension discussed above. In the case of a single system, we may assume that its adaptation is under the control of a *single authority* (that must take into account the fact that the constituent services of the managed system could be operated by third parties). In the case of multiple systems, their adaptation could be still under the control of a single authority. Alternatively, it could be under the control of *multiple cooperating authorities*, that, for example, agree

on some common utility objective. Some works have been recently proposed in this direction: in [76] the authors propose a decentralized composition mechanism based on the notion of stigmergy, taking inspiration from the interactions exhibited by social insects to coordinate their activities. In [4] the authors propose instead an approach based on the friction concept with the goal of minimizing the waiting time of service requests.

Finally, the adaptation process could be under the control of *multiple non cooperating authorities*, that compete in a selfish way for some set of services (e.g. [45]).

2.4 Dimensions of Self-Adaptation for MAPE-K based Service Oriented Systems

The taxonomy presented in Section 2.3 has been used to provide a high-level classification of self-adaptable SOSs. Anyway, most of them are architected according to the MAPE-K reference framework for autonomic systems. Therefore, in this section we provide a classification of self-adaptable SOSs according to how they implement each MAPE-K phase.

2.4.1 Monitor Taxonomy

The taxonomy of the Monitor phase is shown in Figure 2.5.

What We consider the monitoring of QoS parameters, i.e., the set of attributes that describe the performance of the SOA application, or the hardware/software resources that support its execution. For example, the attributes concerning the hardware resources could be the CPU utilization or the amount of available memory, while those

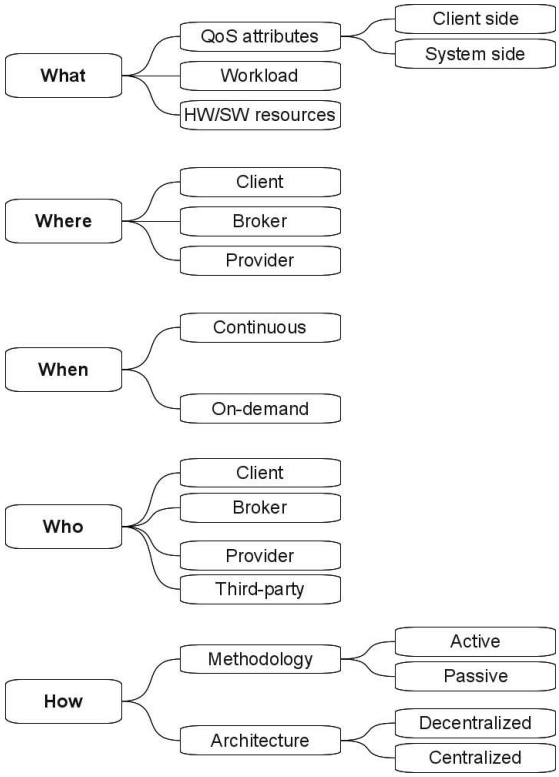


Figure 2.5: Monitor taxonomy.

regarding the software resources could be the length of the backlog queues or the number of threads used by the application server. We can identify two different types of QoS parameters: (i) client-side parameters, like response time, availability, reliability, cost and so on which capture how the clients perceive the application QoS; (ii) system-side parameters, like throughput, cost and reputation which are relevant to the system managers. Given the large set of QoS parameters, the monitoring typically focuses only on those that are involved in the adaptation loop. For example, in frameworks

that dynamically adapt the amount of hardware resources used by the SOA application [21, 71], the monitoring focuses on the hardware resources utilization in order to decide whether and when resize the CPU, memory, or disk. In other frameworks, that do not consider the hardware resource adaptation, other attributes are monitored, such as response time and reliability [3, 9, 15, 33, 66, 75, 89]. Furthermore, the workload submitted to the SOA application can also be monitored, for example because the gathered information can be used to derive some useful metric, like response time. Example of frameworks that monitor the workload can be found in [11, 15, 21].

Where The monitored data can be collected at various different locations. One possible approach is to collect the data at the client side of the SOA application, like in [89], where the client is responsible for detecting the SLA violations. Another approach is to collect the data at the provider side, for example the Amazon CloudWatch service: the service provider collects data for itself and makes them available to its clients. However, the most common solution adopted in the SOA context is to collect data on the service broker that manages the adaptation of the SOA application, as done in [9, 15, 21, 33, 66, 71, 75].

When The monitoring activity can be accomplished continuously or on-demand. Although the latter seems to be reasonable, for example the planning phase might choose to start another monitor activity on a different perspective of the system, to the best of our knowledge the monitoring activity is performed always on a time-continuous base. The frequency is often determined by the cost of collecting the data itself.

Who Several entities could be interested in the monitoring activity: the client for example might want to monitor to detect SLA violation, the broker more generally to detect change in the operational environment and the provider to control the resource utilization. Furthermore, a third party entity not involved in the SOA application might collect data in order to offer to some client the monitoring service.

How The monitoring activities differ in the methodology used to collect the monitored data and in the architecture of the monitoring infrastructure. The methodology can be either active, if the data are collected sending proper inputs to the monitored entities, or passive, if the data are collected without injecting additional load but rather observing the system behavior. The latter solution is usually preferred, especially in the context of the SOA applications, where each service invocation has a cost. For example, [9, 15, 21, 33, 71, 75, 89] all use the passive approach. The active monitoring can be used to proactively determine the service availability. For example, in [15] the framework periodically checks if the used services are available, in order to anticipate a fault diagnosis, without waiting the failure of a service invocation issued by a client.

2.4.2 Analyze Taxonomy

Figure 2.6 depicts the taxonomy for the Analyze phase of the MAPE-K loop.

What The Analyze phase takes input data from the Monitor phase; therefore, it deals with the monitored data.

Where The data analysis can be carried out at different places: the client itself, the broker, and a third-party entity. Client-side analysis is typically carried out in SOA

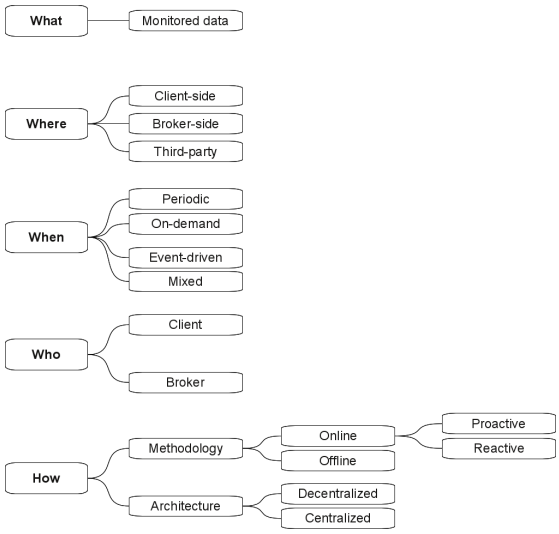


Figure 2.6: Analyze taxonomy.

architectures that do not include an intermediary broker; in this case, the analysis of the monitored data is demanded to either: (i) a monitor service under the control of the client (see [88]), (ii) a third-party (collaborative) monitoring service as in [107]. The analysis algorithms could be as simple as identifying the violation of a threshold or as complicated as creating an empirical distribution function fitting the actual QoS parameters distribution as proposed in [33,88]. Broker-side analysis is usually performed by broker-based frameworks with the help of self-collected monitored data or using a third-party monitoring system. Finally, third-party collaborative services usually offer data analysis as a counterpart for receiving monitored data from SOA executors, as in [107].

When The frequency at which data analysis is carried out is often determined as a trade-off between the need to react to critical events and the cost to process the data. The simplest approach is to periodically analyze the data at fixed interval [15, 33]. We then have event-driven analysis, which is usually based on the concept of Continuous Query Processing (CQP), where each monitored data, besides being stored, might activate triggers based on usually simple policies like threshold violations [21]. On the other hand, event-driven analysis can also occur after the execution of a given service, or a set of services or even the whole workflow [9]. The two approaches, i.e., periodic and event-driven analysis, can be combined to have a periodic analysis coupled with an event-driven analysis for critical events detection [21]. Finally, we can also have on-demand analysis, which is directly requested by a client, depending on its own analysis policies.

Who The entities interested in the analysis phase coincide with those that will plan the adaptation actions, that is, the client and the broker. A client is interested in analyzing data when it does not rely on an external service broker; a service broker is instead always interested in analyzing the monitored data.

How We distinguish between two different aspects of how the analysis can be accomplished: methodological aspects and architectural aspects. Analysis policies can be roughly divided in two macro-categories: online and offline analysis. Since the SOS operations require the adaptation loop to quickly react to a changing environment, a fast analysis is often needed to allow for an early detection and reaction to significant events. As a consequence, we might need to resort to heuristics whenever exact algorithms are too computationally intensive (see [88, 96]), hence not suited to online

operations. Offline analysis still plays a significant role since the collected data is used to identify suitable models of the complex SOA environment. Online algorithms can be further divided into reactive and proactive analysis. In reactive approaches, the system evaluates the collected data and reacts to event as they are detected, e.g., [21, 33, 75]. This implies that the system can only react to events after they occur. Proactive approaches take advantages of predictive models to actually anticipate the occurrence of events, thus possibly invoking the adaptation planner before the violation could actually happen, e.g., [9]. From an architectural point of view, we distinguish between centralized and decentralized approaches. The former have the well-known quality of being easily manageable, while the latter have the ability to be more scalable.

2.4.3 Plan Taxonomy

The taxonomy of the Plan phase is shown in Figure 2.7.

What The Planning phase is the pivotal phase around which the entire autonomic cycle revolves around. The role of planning is to determine and identify the plans and its constituent adaptations actions to be set forth for the system to attain its goals and/or maintain its objectives in face of a changing internal and/or external environments. Planning take different forms depending on whether the adaptation cycle consider the functional or the non-functional behavior of the SOA application. When the adaptation concerns the functional behavior, since the core of a SOA application is a workflow that defines the business logic of the application itself, planning the adaptation of the functional behavior means evaluating what changes have to be made to the workflow itself. For example, in [33, 71] the interactions between services already involved in the workflow could be removed, or new ones could be introduced; furthermore, it is also

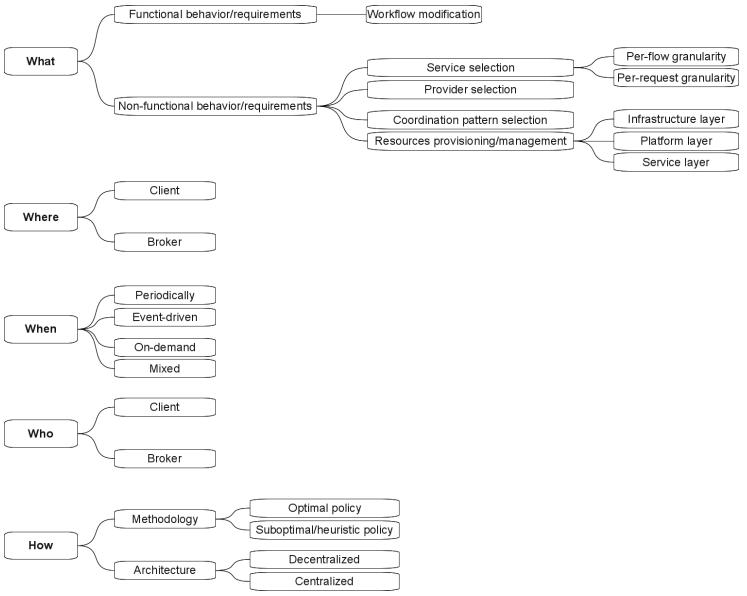


Figure 2.7: Plan taxonomy.

possible to introduce new services with subsequent interactions. Actually, in [71] the framework does not plan by itself the necessary changes to the workflow, but rather it is a client task to submit a set of possible solutions to the new functional requirements; the framework task is to evaluate the QoS of each solution choosing the most suitable one with respect to a given utility function. In [33], the behavior of the abstract tasks can be relaxed or complicated according to the required non-functional attributes: for instance, an order processing workflow could be re-arranged by excluding the credit check before conclusion if response time and customer satisfaction are preferred to risk and cost.

Adaptation of the non-functional requirements has received increasing attention in the last few years. In most of the existing frameworks, adaptation is typically achieved

by selecting at runtime the service(s) that implements the application itself. In this context, the role of planning entails the discovery, identification, and determination of the actual services implementing the SOA application as to satisfy non-functional requirements while optimizing, at the same time, a suitable utility function. In literature, this service selection has been considered at two separate granularity levels: per-request [9] and per-flow [11, 24, 54], introduced in Section 2.3.

In some of these approaches, to satisfy the non-functional requirements the subject of the plan activity also entails the selection of the service providers with which bargaining a SLA. This provider selection could be done, for example, to build the set of semantically equivalent services that serve as basis to plan the service selection.

Some frameworks [15, 66], consider also the coordination pattern service selection. These frameworks, rather than just selecting a single service for each functionality required in the workflow of the SOA application, select a coordination pattern, i.e., a set of services implementing the same functionality, for example to improve the reliability of the whole SOA application. An example of coordination pattern is the invocation in parallel of multiple services in order to improve the reliability, or their sequential invocation to obtain the same goal but at a lower cost and worse response time.

Other approaches plan the provisioning of the manageable resources, e.g., [21, 71], to adjust the system resources allocated to individual services, for example with the aim to sustain the submitted workload. However, such strategy is feasible only for the resources internally administered by the provider of the SOA application, and not for those services offered by external providers.

Where The planning phase is usually executed on the broker, and this is the solution adopted in almost all of the frameworks (e.g. [9, 21, 23, 103]). However, it is also

possible to execute the planning on the client, like in [89], in case of a broker-less architecture.

When Similarly to the Analyze phase, the Planning execution is determined by the trade-off between the need to react to critical events, as the arrival or departure of clients or the SLA violations by a service, and the execution time of the service selection. Planning can be either carried out at fixed time interval or executed whenever the changes in the environments as detected by the Analyze phase might cause the current plan to be no longer adequate to guarantee the system requirements. As noted before, we can combine the two approaches, i.e., a periodic planning coupled with an event-driven planning activated by the analysis component. Finally, we can have on-demand planning, which is directly requested by a client depending on its own planning policies and current perception of the quality attributes of the SOA application.

Who The entities interested in the planning phase are the same that perform the analyze task, that is, client and broker. A client is interested in planning the adaptations actions when it does not rely on an external service broker; a service broker is instead always interested in the planning phase to keep the adaptation decisions under its control.

How The execution of the planning phase can be accomplished using two different methodologies aimed at computing an optimal or a suboptimal/heuristic policy.

The former type of methodologies determines an optimal solution given a utility function and some constraints. The optimization problem can be formulated using linear programming [24, 54], integer programming [6], or even mixed integer linear

programming [12, 33]. The high computational complexity of the optimal service selection policies may limit their use for an online implementation. Various factors affect the time complexity of the service selection policies, among which the most important are the number of abstract tasks, the number of concrete services implementing each abstract task, and the number of QoS constraints that have to be considered. The service selection can be modeled as a Multi-choice Multidimensional Knapsack problem (MMKP), which is known to be NP-hard and therefore the time complexity in finding an exact solution is expected to be exponential [62].

However, in a real-world scenario, the Plan component of the SOS must be able to determine in near real-time the optimal service selection under possibly heavy load. To address this issue, many research efforts have proposed computationally efficient, albeit suboptimal, solutions to the service selection problem. Since a MMKP problem can be formally expressed with an IP formulation, a common approach [16, 54] is to relax the integer restriction on the variables of the IP problem, thus obtaining a LP problem that can be efficiently solved in polynomial time. The caveat is however that a solution to the relaxed problem does not necessarily solve the original problem. Therefore, solutions based on a LP formulation are more suited to address the selection problem at per-flow granularity level, where the QoS constraints are evaluated in the long-term and for a flow of requests, rather than the per-request granularity, where individual executions could violate the constraints. The work in [16] proposes an algorithm for finding a sub-optimal solution to the original IP problem by enumerating the solutions of the LP problem in a clever way, until the IP problem constraints are not violated. The authors show that the proposed heuristic is able to compute close to optimal solutions in a fraction of the time with respect to the exact MIP formulation, e.g., in case of

a SOA application composed by 21 tasks, the heuristic reaches 98.83% of the objective function value of the optimal solution, but only needs 0.19% of the computation time to compute it. On the other hand, the proposal in [54] does not try to fit the original IP problem, but rather to refine the LP solution so that it can be used to guarantee some QoS constraints for every execution of the SOA application, or at least for a large percentage (e.g., 99.9%) of the executions. The authors show that the proposed heuristic is able to provide less than 3% of deviation from the original IP solution. Another approach to face the complexity of the IP formulation is to reduce the number of decision variables of the problem itself, as in [66]. The authors first decompose each global QoS constraint into a set of local constraints, so that each local constraint serves as a conservative upper bound such that the satisfaction of every local constraint guarantees the satisfaction of global constraints. Then, they divide the quality range of each QoS attribute into a set of discrete quality levels and map each known concrete service to the appropriate quality level. This approach has two major benefits: first, it allows to distribute the computational effort among different nodes, because only independent local optimization problems have to be solved; secondly, since concrete services are replaced by quality levels, the size of the problem space is reduced. The authors show that their heuristic can achieve above 96% of optimality when compared to the results obtained by the global optimization approach. However, since QoS levels are discretized without considering potential correlations among different quality attributes, in scenarios with relatively strict constraints it is possible to incur in very restrictive decompositions of the global constraints, which therefore could not be satisfied by any concrete service even though a solution to the problem exists. A solution to the latter problem is presented in [7], where the authors propose a different method for QoS level discretiza-

tion: for each abstract task, skyline (dominant) concrete services are first determined. Subsequently, skyline concrete services are clustered using the k-means algorithm and, for each cluster, a virtual concrete service is created whose quality level is given by the worst quality attributes of the concrete services belonging to that cluster. Those virtual concrete services are then used to discretize QoS levels in a multidimensional fashion. A completely different approach is proposed by [22], where a Genetic Algorithm (GA) is used to realize an enumeration of the optimization problem solutions. The search for the optimal solution starts with an initial population of individuals that are going to evolve over time: at each algorithm step individuals are evaluated using a fitness function and then selected through a selection operator. The higher is the fitness value of an individual, the more is likely that such an individual will be chosen for reproduction. The reproduction is obtained by applying crossover and mutation operators. The former produces an offspring recombining parent's genes, while the latter modifies one or more genes. The application of a GA in service selection maps a solution of the optimization problem to an individual, where each individual is composed by genes and every gene represents a particular instance of concrete services. A different objective is pursued by [100], which uses a GA for the service provisioning problem: in their work the individual is composed by several genes which do not represent a particular instance of concrete service, but the number of concrete services needed by a given abstract task to fulfill certain QoS constraints. Finally, in [104] the authors compare the MMKP problem solved through the branch-and-bound technique with several heuristics, based on either a combinatorial or a graph model. The proposed heuristics differ in the type of considered workflow structure, which can be either only sequential or more general (a sequential workflow contains neither conditional branches nor forks).

Combinatorial heuristics for both sequential and general workflows are realized as a walk in the solution space: first, a concrete service is selected for each abstract task such that a quality attribute (possibly different for each abstract task) is locally maximized. If the obtained solution is feasible, then the second step tries to improve such a solution by both feasible and unfeasible upgrades, so that both local and global optima can be reached. The authors claim that in most cases (more than 98%), the heuristic finds a feasible solution at the first try, while the time complexity is a polynomial function. As regards general workflows, an additional heuristic is proposed, which tries to optimize only the execution route with the highest probability, while finding only feasible solutions for other routes.

Graph-based heuristics are based on the algorithm of single-source shortest paths in Directed Acyclic Graphs (DAG) [38]: a DAG is built up from the workflow by replacing every node representing a single abstract task with a set of nodes representing the concrete services implementing it and by adding edges between two concrete services if the abstract tasks they implement are connected. Loops, if any, are unfolded. The proposed heuristic limits the information held by each node: instead of maintaining the complete list of paths that meet the QoS constraints from the source to the node itself, only K paths are kept. The authors show that limiting the information to the K best paths leads to an optimality approximation greater than 90% even for small values of K, with a gain in terms of time and memory consumption of approximately 500%.

With regards to the architecture of the planner, it is centralized in most of the frameworks, although some decentralized approach exists, as in [6] where part of the computation is distributed across the network.

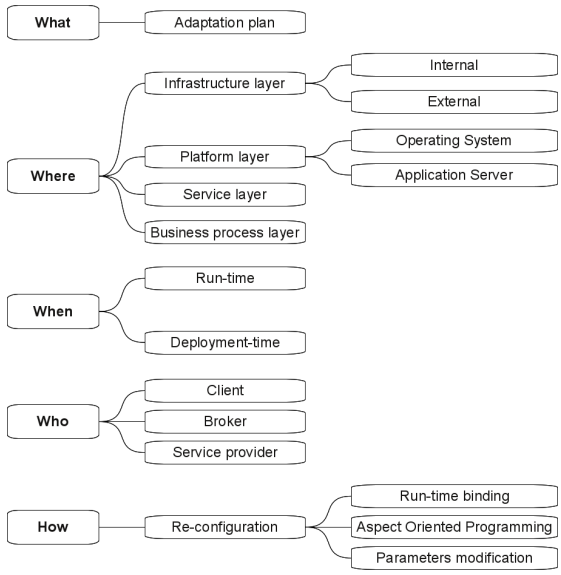


Figure 2.8: Execute taxonomy.

2.4.4 Execute Taxonomy

Figure 2.8 shows the taxonomy of the Execute phase of the MAPE-K loop.

What In this taxonomy, the question assumes a trivial meaning: what we are going to execute is exactly what we have planned in the previous phase.

Where We can apply the adaptation plan computed by the Plan phase at different layers, ranging from the business process layer to the infrastructure layer. Starting from the lowest layer, namely the Infrastructure layer, we can choose to run the adaptation actions on the internal infrastructure [21, 71] or on an external infrastructure. By internal infrastructure we mean all those physical and virtual resources that are directly

manageable by the SOA application provider, while with external infrastructure we intend every external physical or virtual resource used to improve or to replace any internal infrastructure; at this abstraction level we can operate by adding or removing physical or virtual machines, by improving network connections or storage systems and so on.

Going up through the abstraction layers, we find that adaptation can take place at the Platform layer. The latter identifies every software needed to run the service we intend to adapt, thus ranging from the Operating System to any Application Server [21]. Changes on this layer involve everything that goes from kernel reconfiguration to application server tuning, but it does not involve any modification on services that take part in the business process. Such modifications belong to the Service layer, where we can operate both service re-configuration and service tuning. Finally, at the Business process layer, the adaptation actions involve the high-level logic of the business process [15, 21, 66].

When Most of times the adaptation actions have to be carried out introducing the lowest possible delay into the business process execution. Depending on the adaptation actions, the adaptation may happen (i) at run-time or (ii) at deployment-time. Although it is possible to execute adaptation actions also at development-time or design-time, we do not consider them because we only focus on those solution that do not require human intervention, being the latter a requirement for a truly autonomic system. We include in the deployment-time phase all those approaches that require a (even small) service interruption in order to apply the adaptation plan. All other approaches can be classified in the run-time case.

Who The three entities involved in the actuation of an adaptation plan are the client, the broker, and the service provider. A client managing the entire service orchestration can apply by its own the adaptation actions previously computed in the planning phase. A broker could either apply its own computed adaptation plan or it could rely on some adaptation plan directly provided by the client as in [71]. Finally, the service provider could modify its behavior according to directives provided by the client or the broker. For example, it could receive an adaptation request issued by a broker that has detected a slowdown in the provider performance.

How The adaptation actions that can be taken are all part of a meta-branch called re-configuration. In particular, we have identified three possible mechanisms to apply the adaptation plan: run-time binding, Aspect Oriented Programming (AOP), and parameters modification. The run-time binding is the most leveraged approach, as it provides the SOA application with the ability to bind at run-time the invocation with the actual service according to the Plan decision. It is the most suited mechanism to implement service selection, coordination pattern selection or even a simple load balancing policy among functionally equivalent services. AOP can be used to inject code fragments (also known as sub-processes) into the SOA application itself, in order to have process segments changing at run-time [59] or at deployment-time. This methodology is suited for both non-functional and functional adaptation as it can modify the functional as well as non-functional application behavior. The AOP methodology is based on the concepts of aspect (cross-cutting concerns, which are turned off and on at design or run-time), advises (the actual implementation in terms of business logic of the aspects), joinpoints (points on the business process where advices can potentially be inserted), and weaving (the process of dynamically inserting advises in joinpoints).

Finally, the parameters modification encompasses all those mechanisms that can be used to change some operative feature of the SOS.

3

MOSES: a Framework for QoS Driven Runtime Adaptation of Service-Oriented Systems

Contents

3.1 Overview of MOSES 44
3.1.1 A Comparison of Frameworks for Self-Adaptation of SOSs 45
3.2 Plan Phase 51
3.2.1 Composite Service Model 52
3.2.2 Adaptation Actions 55
3.2.3 SLA Model 58
3.2.4 Adaptation Policy Model 59
3.2.5 QoS Model 60
3.2.6 Optimization Policies 64
3.3 Monitor Phase 82
3.3.1 Monitoring the Concrete Services 83
3.4 Analyze Phase 83

In this chapter we will present the main concepts behind the realization of an autonomic QoS-aware *service broker* for SOSs. We will start by introducing MOSES: MOdel-driven SElf-adaptation of SOA Systems, a service broker designed according to the MAPE-K loop (Figure 3.1). We will give an overview on MOSES according to the taxonomy proposed in Section 2.3 and then we will introduce a comparison between MOSES and other existing frameworks in Section 3.1.1.

The core of the chapter will be the analysis on how MOSES follows each phase of the MAPE-K reference: Section 3.2 will describe the MOSES Plan phase, which is the core of the MAPE loop; then we will focus on the MOSES Monitor and Analyze phases respectively in Sections 3.3 and 3.4. Since the MOSES Execution phase is purely implementative, we will not describe it in this chapter which focuses on the design, but rather we will provide an in-depth description in Section 4.3.

3.1 Overview of MOSES

MOSES is a service broker designed according to the MAPE-K loop. Its main task is to offer to prospective users a SOA application architected as a composite service with a range of different service classes. To this end, it exploits a set of existing concrete services, driving the adaptation of the composite service to fulfill the QoS goals of the different service classes it offers, when changes occur in its operating environment.

The MOSES input consists of the description of the composite service in some suitable workflow orchestration language (e.g., BPEL [82]), and the set of candidate concrete services that can be used to implement the required tasks (including the parameters of their SLAs). MOSES uses this input to build a model which is then used (and kept up to date) at runtime to determine possible adaptation actions to be performed. Each macro-component in Fig. 3.1 is actually architected as a set of interacting components. We give some details about these components and their functions in Section 4.3.

Figure 3.2 shows how MOSES fits into the self-adaptive SOSs taxonomy presented in Section 2.3. Specifically, MOSES does not address adaptation according to functional requirements: it instead supports adaptation based on non-functional require-

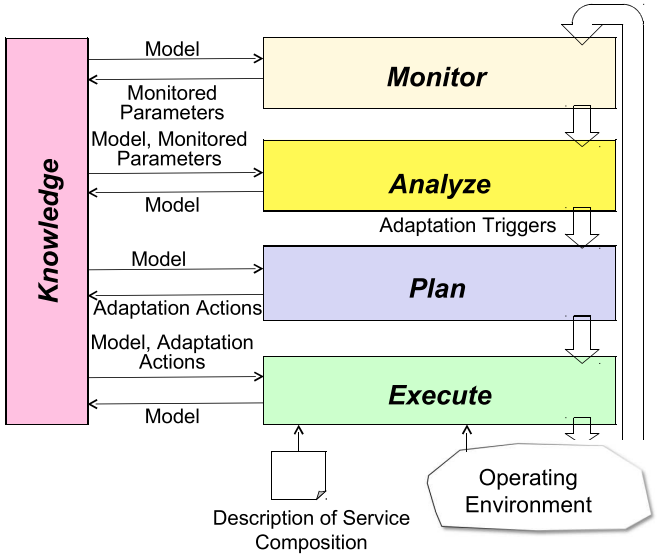


Figure 3.1: The MOSES approach.

ments and can act both on average values and on percentiles. Its reactive behavior produces adaptation actions which are taken at *runtime* and consist in *service selection* as well as *coordination pattern selection*. Therefore, the adaptation is carried out at *services only* level, thus never modifying the original workflow. From the *scope* point of view, MOSES currently supports the management of a *single system*, that is, the adaptation policy is centrally computed and enforced. Such a policy supports both adaptations based on *per-request* and *per-flow* strategies.

3.1.1 A Comparison of Frameworks for Self-Adaptation of SOSs

Several solutions have been proposed for the self-adaptation of SOSs (e.g. [4, 6, 9, 11, 12, 16, 47, 54, 58], but very few frameworks and platforms have been realized and

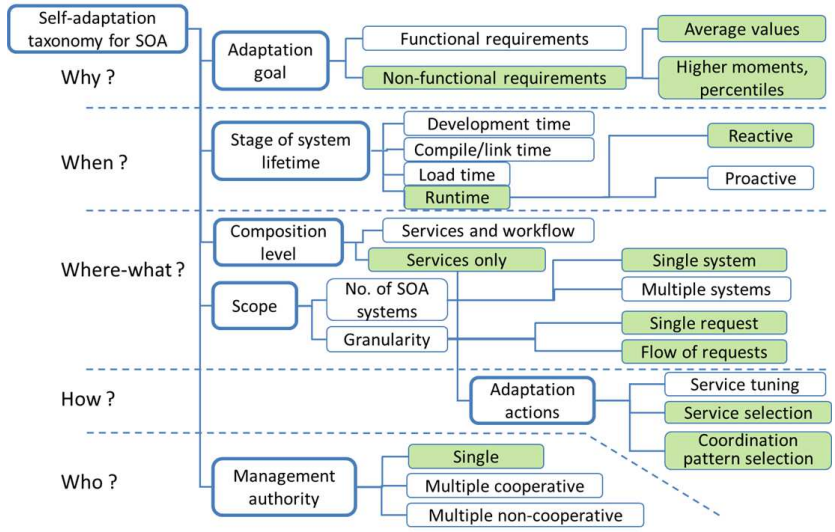


Figure 3.2: MOSES within the self-adaptable SOS taxonomy.

evaluated in the scope of self-adaptation of SOSs.

To the best of our knowledge, besides MOSES, realized and evaluated platforms are: MUSIC [89], SASSY [66, 68], VieDAME [74] and VRESCo [70]. In this section we review the characteristics and features of the aforementioned brokers according to the taxonomy presented in Section 2.3 and compare them to MOSES.

Adaptation Goal The adaptation goal of all the considered frameworks is the optimization of the quality attributes of the served SOSs. However, with respect to MOSES, MUSIC, SASSY, and VRESCo only support the strict fulfillment of every single request. Even if this behavior can be also activated in MOSES, we found that it leads to an unstable system due to the possible frequent changes of the execution plan.

Stage of System Lifetime All the considered systems work at run-time in a reactive fashion: once the SOS is ran for the first time, the first step is to compute an optimal or sub-optimal service selection policy, which is then updated according to the current state of the environment. However, thanks to its modular architecture, MOSES can be easily extended to implement a proactive monitoring data analyzer in order to forecast possible future threats and anticipate them by requesting an early optimization. In addition to any other framework, VieDAME also supports the possibility to change both the QoS parameters and QoS model at run-time without service interruption. To this end, it defines and implements an high-level configuration language named VieDASSL (Vienna Domain-Specific Service Selection Language), that can be used by domain expert to define the non-functional behavior of the business application.

Composition Level None of the considered frameworks, neither MOSES, support adaptation at workflow level: all the approaches are focused on the adaptation of the services used in the composition.

Scope - Number of SOA Systems MUSIC is the only framework supporting adaptation for multiple systems. It explicitly supports two kind of resources: internal and external. An internal resource is a service that can be locally consumed, while an external resource is a service that has to be remotely invoked eventually by paying a fee. In the first case, even if there are possibly no fees to pay, MUSIC must reserve enough system resources in order to be able to execute the local service respecting the SLA, therefore acting as a customer with respect to itself; furthermore, MUSIC also considers the possibility of selling the service to an external MUSIC requestor, therefore acting as a service provider. Summarizing, since MUSIC acts both as a service

requestor and as a service provider, it should also consider the opportunity cost.

Scope - Granularity MUSIC, SASSY, VieDAME and VRESCo only support the *single request* granularity. MOSES is the only platform supporting both *single request* and *flow of requests* granularities.

Adaptation Actions All the considered frameworks support adaptation actions in terms of *service selection*. That is, they implement a run-time binding of abstract services to concrete services which can be exploited to maximize/minimize a given utility function. However, none of the considered frameworks, with the exception of MOSES, implement or provide the tool for an optimal service selection strategy.

In detail, the MUSIC optimization strategy, described in [90], is based on an heuristic approach which enumerates existing service plans trying to maximize a given utility function. Even if this approach could lead to a possible optimal solution in case of a complete enumeration of all the possible service plans, no detail is provided about their generation.

The SASSY approach is based on a sub-optimal heuristic presented in [66]. However, even if in the example that the authors present they manage to increase the system availability by almost 8.5% at the expense of a service time increased by almost 4.7% and a cost increased by 134%, they do not provide the comparative results of the optimal service selection strategy, therefore not giving a baseline to compare the effectiveness of the proposed heuristic. Furthermore, in the discussed example, which consisted of 5 abstract services implemented by 17 concrete service in total, the sub-optimal solution is computed in 2.25 s. The authors claim that this result demonstrates that the heuristic can be used to solve the optimization problem in near real-time. However,

we believe that a single example is not enough to generalize the result. In comparison with MOSES, in [23] we prove that the optimal solution for a LP problem for a process composed by 10 abstract services, to each of which correspond 10 concrete services, and considering also coordination patterns can be computed in 0.1 s.

VieDAME does not support strict fulfillment of QoS attributes: it exploits VieDASSL to define rules that are subsequently used to drive the computation of scores for the available services. Such scores are then used to build a classification of the services and the best one is chosen to implement the required functionality. In order not to overload the best service, a selection post-processor is also implemented: instead of always using the same best concrete service, a subset of the available services with a score greater than a threshold is considered.

VRESCo does not propose either optimal or sub-optimal solutions to the service selection problem: the authors just provide a framework for dynamic binding and a query language that can be used to retrieve execution plans that must be afterwards processed by an optimization algorithm. From this point of view, VRESCo provides an additional feature with respect to MOSES: *Service Mediation*. In MOSES the input of the concrete services is the exact input provided by the service requestor because we suppose that, given an abstract service, all the concrete services implement the same abstract interface. VRESCo goes one step further, assuming that the there could be interface differences. To this end, VRESCo accepts from the service requestor a high-level representation of the data that will be used as input for the concrete services. These data are then lowered (i.e., transformed from high-level representation into low-level format) in order to be compatible with the given concrete service. The response is instead lifted (i.e., transformed from low-level format to high-level representation)

in order to be compatible with service requestor expectations. From a performance perspective, the VRESCo run-time binding adds about 400 ms to the service execution time, without considering the service mediation. This additional time is due to the addition of a proxy which, for each service invocation, queries the database in order to know which service to invoke. In MOSES we managed to reduce this overhead to about 25 ms (see Section 5.3.1 for more details).

MOSES is the only existing framework which is able to provide out of the box three different optimization strategies [12, 23, 27] based on LP and MILP. However, thanks to its modular architecture, MOSES also provides interfaces that can be implemented with different and new optimal or sub-optimal (heuristic) optimization strategies.

Coordination pattern selection is supported by SASSY and MOSES. In detail, SASSY supports three coordination patterns named: *basic (B), load balancing (LB), fast fault tolerant (fFT)*. The B coordination pattern is equivalent to the *single* pattern offered by MOSES, while fFT corresponds to the *parallel or*. Both coordination patterns are described in Section 3.2.2. SASSY provides one more coordination pattern, that is LB, which dispatches service requests to concrete services using a weighted round robin rule. However, even if MOSES does not implement the LB coordination pattern, load balancing can be achieved using the already implemented coordination patterns with a load-aware service selection policy. Currently, MOSES implements two load-aware optimization algorithms described in [23, 27].

Management Authority All the considered frameworks with the exception of MUSIC are under the control of a *single* management authority. The latter instead considers an environment where *multiple non-cooperative* MUSIC instances can work together.

After having analyzed the frameworks closely related to MOSES, we now focus

our attention on the MOSES design.

3.2 Plan Phase

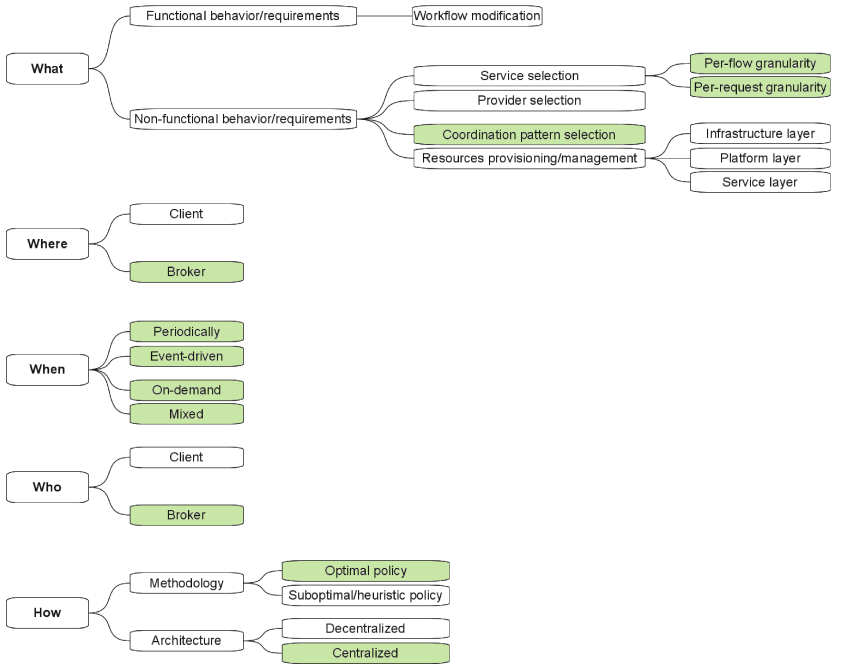


Figure 3.3: MOSES within the Plan taxonomy

Figure 3.3 shows how MOSES fits into the taxonomy proposed in Section 2.4.3.

In Section 3.2.1 we present a grammar used to identify all the possible instances of composite services manageable by MOSES; in Section 3.2.2 we describe how MOSES can exploit multiple implementations of the same abstract service to increase QoS values; since MOSES acts both as a client towards concrete services and as a server towards composite service clients, it must establish contracts with both ends. In Section

3.2.3 we describe how we model such contracts. The core of the Plan phase is represented by the optimization problem that determines which concrete implementation must be bound to each abstract task in order to fulfill the SLAs: in Section 3.2.4 we describe how we represent such information (i.e. this is the output of the optimization problem), while in Section 3.2.5 we show how to compute the expected QoS attributes of a single abstract task. Finally, in Section 3.2.6 we present three service selection policies: two second generation per-request and per-flow policies and finally a third generation load-aware per-request policy.

For each presented policy we will illustrate: (i) the workflow model, (ii) the computation of QoS attributes and (iii) the optimization problem.

3.2.1 Composite Service Model

The class of services managed by MOSES consists of all those composite services whose orchestration logic (*i.e.*, their abstract composition, according to the terminology of Section 2.3) can be abstractly defined as an instance generated by the following grammar:

$$\mathrm{C} ::= \mathrm{S}|seq(\mathrm{C}^+)|loop(\mathrm{C})|sel(\mathrm{C}^+)|par\_and(\mathrm{C}^+)$$

$$\mathrm{S} ::= S_1|S_2|...|S_m$$

In this definition, C denotes a composite service, S_1, S_2, ..., S_m denote tasks (i.e., functionalities needed to compose a new added value service), and C^+ denotes a list of one or more services. Hence, MOSES is currently able to manage composite services consisting either of a single task, or of the orchestration of other services according to the composition rules: *seq*, *loop*, *sel*, *par_and*. Table 3.2 summarizes the intended

meaning of these rules and the corresponding BPEL constructs. For the sake of clarity, in Table 3.1 we summarize the notation used throughout this thesis.

Symbol	Description
K	Set of classes
k	Class index
$R^k_{\max}$	Class k upper bound on the expected response time
C^k	Class k cost
D^k_{min}	Class k lower bound on service reliability
λ^k_u	Class k flow of request rate generated by user u
L^k	Class k flow of request rate
S_i	Task
i	Task index
m	Number of tasks
op_{ij}	Operation/Concrete service
ij	Concrete service index
z_{ij}, $z = r\|c\|d$	Operation op_{ij} response time, cost and reliability
L_{ij}	Maximum operation op_{ij} load
$\Im_i$	Set of task i implementations
J	Implementation index
$Z(S_i; J)$, $Z = R\|\log D\|C$	Task S_i response time, cost and (log of the) reliability under implementation J
x^k_{iJ}	Fraction of class k requests for task S_i that are bound to implementation J
V^k_i	Expected number of times task S_i is invoked by a class k user
$Z^k(\boldsymbol{x})$, $Z = R\|\log D\|C$	Class k response time, cost and (log of the) reliability under adaptation policy $\boldsymbol{x}$
w_z, $z = r\|c\|d$	Normalized QoS attribute weight

Table 3.1: Main notation adopted in the thesis.

We point out that the above grammar is purposely abstract, as it intends to succintly specify only the structure of the considered composite services. Hence, we omit details

such as how to express the terminating condition for a loop. A thorough approach to the modeling of service orchestration is presented in [53], based on the *Orc* language; [37] shows how *Orc* can model the workflow patterns listed in [1]. In this respect, we point out that the grammar we define does not capture all the possible structured orchestration patterns, but includes a significant subset[1].

Rule	Meaning	BPEL
$seq(\mathrm{C}^+)$	sequential execution of services in C^+	`sequence`
$loop(\mathrm{C})$	repeated execution of service C	`while`
$sel(\mathrm{C}^+)$	conditional selection of one service in C^+	`switch`
$par\_and(\mathrm{C}^+)$	concurrent execution of services in C^+ (with complete synchronization)	`flow`

Table 3.2: Workflow composition rules.

Figure 3.4 shows an example of an orchestration pattern described as a UML2 activity diagram, and the corresponding instance generated by the grammar.

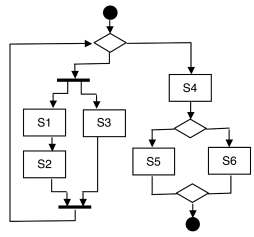


Figure 3.4: A MOSES-compliant workflow.

MOSES uses this grammar to check whether the orchestration pattern of an actual SOS matches the kind of patterns it is able to manage. In the positive case, it uses the grammar to support the construction of a suitable runtime model to be used for

[1] In particular, it can be easily realized that our grammar captures the structure of workflow patterns 1, 2+3, 4, 10 (for structured cycles only), 13 and 16 reported in [1].

adaptation purposes.

3.2.2 Adaptation Actions

MOSES performs adaptation actions that take place at the *services only* composition level, as classified in Section 2.3. Their goal is to determine at runtime the most suitable implementation to be bound to each abstract task S_i, selecting it from a set $\Im_i$ of available implementations, built as follows.

We assume that a set $CS = \{cs_l\}$ of candidate concrete services have been identified to build an overall implementation of the composite service. Different cs_l can be offered by different providers with different QoS and cost attributes, or even by the same provider offering differentiated services.

Each cs_l implements a set $OP(cs_l)$ of operations. We denote by $OP = \cup OP(cs_l)$ the set of all the available operations, and by $\mathcal{OP}_i \subseteq OP$ the subset of functionally equivalent operations that implement the task S_i.

MOSES exploits the availability of multiple equivalent operations to build implementations of each S_i based on the use of *redundancy* schemes, to get QoS levels possibly higher than those guaranteed by each single operation, at the expense of a higher cost. According to these schemes, a possible implementation of a task S_i may consist of a set of two or more equivalent operations belonging to $\mathcal{OP}_i$, coordinated according to some coordination pattern.

At present, the MOSES framework includes two such coordination patterns, denoted as *alt* and *par_or*, besides the simple *single* pattern. Table 3.3 summarizes their intended meaning. We have selected these two coordination patterns as they have complementary characteristics with respect to their QoS and cost, as will be explicitly dis-

cussed in Section 3.2.5.1.

Table 3.3: Coordination patterns.

Rule	Meaning
$single$	execution of a single operation
alt	sequential (alternate) execution of operations in a list, until either one of them successfully completes, or the list is exhausted
par_or	concurrent execution of the operations in a set (with 1 out of n synchronization)

Hence, the set $\Im_i$ of available implementations for each task S_i is given by the union of the following sets:

$$\Im_i = \mathcal{OP}_i \cup \mathcal{OP}_i^{alt} \cup \mathcal{OP}_i^{par}$$

where:

- $\mathcal{OP}_i$ has been already defined above; selecting an element in this set models the selection of an implementation of S_i based on a single operation;
- $\mathcal{OP}_i^{alt}$ is the set of all the ordered lists of at least two elements belonging to $\mathcal{OP}_i$, with no repetitions; selecting an element in this set models the selection of an implementation of S_i based on the *alt* pattern applied to that list;
- $\mathcal{OP}_i^{par}$ is the set of all the subsets of at least two elements belonging to $\mathcal{OP}_i$; selecting an element in this set models the selection of an implementation of S_i based on the *par_or* pattern applied to that subset.

For a given abstract composition that models the business logic of a SOS, the selection for each S_i of different elements in the set $\Im_i$ corresponds to different concrete

configurations of the overall composite service, each characterized by different values of their overall QoS attributes. We call *adaptation policy* the runtime selection and implementation of one of these configurations, to best match the QoS constraints and objectives in a given operating environment.

3.2.2.1 Adaptation Actions for Stateless and Stateful Services

In the discussion above about the MOSES adaptation actions, we implicitly assume that tasks can be bound to any concrete service implementing them. Actually, this holds only for *stateless* tasks, *i.e.*, tasks that do not require sharing any state information with other tasks. In the general case, composite services may include *stateful* tasks, *i.e.*, tasks that do need state information to be shared among them; as a consequence, these tasks need to be implemented by operations of the same concrete service. This very requirement limits the possibility of exploiting redundancy patterns to implement stateful tasks. Indeed, the functionally equivalent operations used within these patterns generally belong to different concrete services. This makes unlikely, or even impossible, the sharing of state information among them, unless we put constraints on the implementations. To overcome this problem, MOSES currently uses the *alt* or *par_or* patterns for the implementation of stateless tasks only, while the implementation of stateful tasks is restricted to only the *single* pattern.

We model the presence of *stateful* tasks by considering a partition $\mathcal{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_f\}$ of the set of tasks $\{S_1, \ldots, S_m\}$. Tasks that need to share some state information belong to the same subset $\mathcal{S}_h \in \mathcal{S}$ and need to be implemented by operations of the same concrete service cs_l. A *stateless* task S_i is simply modeled by associating it with a singleton $\mathcal{S}_h \in \mathcal{S}$.

3.2.3 SLA Model

In general, a SLA may include a large set of parameters, referring to different kinds of functional and non-functional attributes of the service, and different ways of measuring them. MOSES presently considers the following attributes:

- *response time*: the interval of time elapsed from the service invocation to its completion;
- *reliability*: the probability that the service completes its task when invoked[2];
- *cost*: the price charged for the service invocation.

Other attributes, like reputation or availability, could be easily added.

Our general model for the SLA between the provider and the user of a service thus consists of a tuple $\langle R, C, D, L \rangle$, where: R is the upper bound on the service response time, C is the service cost per invocation, D is the lower bound on the service reliability. The provider guarantees that thresholds R and D will hold on average provided that the request rate generated by the user does not exceed the load threshold L.

In our framework MOSES performs a two-fold role of service provider towards its users, and of service user with respect to the providers of the concrete services it uses to implement the composite service it is managing. Hence, it is involved in two types of SLAs, corresponding to these two roles, that are both defined using the SLA template. In the case of the SLAs between the composite service users and MOSES (acting the provider role), we assume that MOSES offers a set K of service classes. Hence, the SLA for user u of service class $k \in K$ is defined as a tuple $\langle R^k_{\max}, C^k, D^k_{\min}, \lambda^k_u \rangle$. All

[2]This measure is called *successful execution rate* in [105].

these coexisting SLAs (for each u and k) define the QoS objectives that MOSES must meet.

To meet these objectives, we assume that MOSES (acting the user role) has already identified for each task S_i a pool of concrete services implementing it. The SLA contracted between MOSES and the provider of the operation $op_{ij} \in \mathcal{OP}_i$ is defined as a tuple $\langle r_{ij}, c_{ij}, d_{ij}, L_{ij} \rangle$. These SLAs define the constraints within which MOSES should try to meet its QoS objectives.

3.2.4 Adaptation Policy Model

The MOSES adaptation policy is based on a set of directives used to select at runtime the "best" implementation of the composite service in a given scenario. The MOSES adaptation policy consists of determining, for each service class k and each task S_i:

- the coordination pattern(s) and the corresponding list of operations to be used to build concrete implementation(s) for S_i (selected among the *single*, *alt* and *par_or* patterns).
- the fraction of requests generated by class k requests for S_i that must be switched and bound to a specific implementation of S_i.

We model the MOSES adaptation policy by associating with each class k a vector $\boldsymbol{x}^k = [\boldsymbol{x}_1^k, \ldots, \boldsymbol{x}_m^k]$, where each entry $\boldsymbol{x}_i^k = [x_{iJ}^k]$, $0 \leq x_{iJ}^k \leq 1$, $J \in \Im_i$, $\sum_{J \in \Im_i} x_{iJ}^k = 1$, $i = 1, \ldots, m$, denotes the adaptation policy for task S_i. Here, x_{iJ}^k denotes the fraction of class k requests for S_i to be bound to the implementation denoted by J. We denote by $\boldsymbol{x} = [\boldsymbol{x}^k]_{k \in K}$ the MOSES adaptation policy vector which encompasses the adaptation policy of all the service classes.

The adaptation policy vector $\boldsymbol{x}$ is used by the Adaptation Manager, a software module described in Section 4.3.1, to determine for each and every invocation of a task S_i the coordination pattern to be used and the actual service(es) to implement it. Given a class k request for the task S_i, the Adaptation Manager chooses the implementation denoted by J with probability x_{iJ}^k, thus possibly giving rise to a randomized partitioning among the implementations in $\Im_i$ of the overall class k flow directed to S_i. As an example, consider the case $\mathcal{OP}_i = \{op_{i1}, op_{i2}, op_{i3}, op_{i4}\}$ for task S_i and assume that the adaptation policy $\boldsymbol{x}_i^k$ for a given class k specifies the following values: $x_{i\{op_{i1}\}}^k = x_{i\{op_{i3}\}}^k = 0.3$, $x_{i\{op_{i2},op_{i3}\}}^k = 0.4$ and $x_{iJ}^k = 0$ otherwise. According to this policy, given a class k request for task S_i, the Adaptation Manager binds the request: with probability 0.3 to operation op_{i1}, with probability 0.3 to operation op_{i3}, and with probability 0.4 to the pair op_{i2}, op_{i3} coordinated by the *par_or* pattern (see Fig. 3.5).

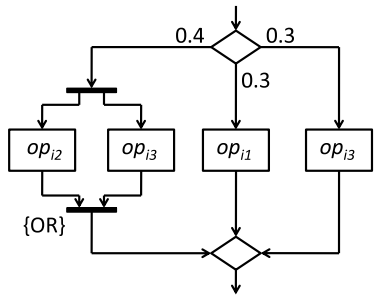


Figure 3.5: Implementation of the MOSES adaptation policy for a single task.

3.2.5 QoS Model

MOSES presently considers the following attributes for each service class $k \in K$:

- the expected response time R^k, which is the time needed to fulfill a class k request for the composite service;
- the expected execution cost C^k, which is the price to be paid for a class k invocation of the composite service;
- the expected reliability D^k, which is the probability that the composite service completes its task for a class k request. As in [105], when writing expressions, we will work with the logarithm of the reliability rather than the reliability itself, to obtain linear expressions, when composing the reliability of different services.

For each service class, the overall QoS of a composite service implementation depends on: the usage profile and the composition logic of the composite service tasks; the adopted adaptation policy; the QoS of the task implementation selected within that adaptation policy.

In Section 3.2.5.1 we derive the QoS attributes of a task as a function of the selected implementation, while in Sections 3.2.6.1, 3.2.6.2, 3.2.6.3 we show how MOSES takes into account task orchestration and usage profile to compute the composite service QoS.

QoS attributes are calculated based on the following assumptions:

- service invocation is synchronous;
- services fail according to the fail-stop model;
- service cost is charged on a per-invocation basis.

3.2.5.1 Task QoS Attributes

Let us first consider a task in isolation. For each class of service, the QoS of a task depends on: 1) the QoS associated with the different set of operations and the associated coordination pattern that can be bound to the task to build its concrete implementation; and 2) the probability that a particular coordination pattern and set of operations is bound to a given request.

Let $Z^k(S_i;\boldsymbol{x})$, $Z = C|D|R$, denote class k QoS attribute of task S_i under the adaptation policy $\boldsymbol{x}$. Since implementation J is chosen with probability x_{iJ}^k, we readily have:

$$C^k(S_i;\boldsymbol{x}) = \sum_{J\in\Im_i} x_{iJ}^k C(S_i;J) \tag{3.1}$$

$$\log D^k(S_i;\boldsymbol{x}) = \sum_{J\in\Im_i} x_{iJ}^k \log D(S_i;J) \tag{3.2}$$

$$R^k(S_i;\boldsymbol{x}) = \sum_{J\in\Im_i} x_{iJ}^k R(S_i;J) \tag{3.3}$$

where $R(S_i;J)$, $C(S_i;J)$ and $D(S_i;J)$ denote the average response time, cost and reliability of S_i, when the implementation of S_i corresponds to a given $J \in \Im_i$.

We now determine the value of these QoS attributes when S_i is implemented according to the three different coordination patterns currently considered within MOSES.

We distinguish among the three cases:

- $J \in \mathcal{OP}_i$: assuming $J = \{op_{ij}\}$, the QoS attributes coincide with those of the selected concrete operation op_{ij}:

$$C(S_i;J) = c_{ij}, \quad D(S_i;J) = d_{ij}, \quad R(S_i;J) = r_{ij} \tag{3.4}$$

- $J \in \mathcal{OP}_i^{alt}$: the concrete operations listed in $J = [op_{ij_1},\ldots,op_{ij_l}]$ are tried

in sequence, starting from the first in the list, until one of them successfully completes. Hence, the reliability of this pattern is derived from the probability that at least one operation completes, while the cost and time to completion of all the elements of the list must be summed, each weighted by the probability that the invocations of all the preceding elements in the list have failed:

$$\begin{aligned} C(S_i;J) &= \sum_{h=1}^{l} c_{ij_h} \prod_{s=1}^{h-1}(1-d_{ij_s}) \\ D(S_i;J) &= 1 - \prod_{h=1}^{l}(1-d_{ij_h}) \\ R(S_i;J) &= D(S_i;J)^{-1} \sum_{h=1}^{l} r_{ij_h} d_{ij_h} \prod_{s=1}^{h-1}(1-d_{ij_s}) \end{aligned} \tag{3.5}$$

- $J \in \mathcal{OP}_i^{par}$: in this case, the costs of all the operations in $J = \{op_{ij_1}, \ldots, op_{ij_l}\}$ must be summed as they are invoked in parallel, while the completion time is the minimum of the completion times of those operations that successfully complete; thus $R(S_i;J)$ is the sum of the minimum completion time of all non-empty subsets $H \subseteq J$ weighted with the probability that only the operations in H do complete successfully:

$$\begin{aligned} C(S_i;J) &= \sum_{h=1}^{l} c_{ij_h} \\ D(S_i;J) &= 1 - \prod_{s=1}^{l}(1-d_{ij_s}) \\ R(S_i;J) &= D(S_i;J)^{-1} \sum_{H \in 2^J \setminus \{\emptyset\}} \Big(\prod_{j_s \in H} d_{ij_s} \prod_{j_s \in J \setminus H} (1-d_{ij_s}) \Big) \cdot \\ &\quad \min_{j_s \in H} \{r_{ij_s}\} \end{aligned} \tag{3.6}$$

We make the following remarks concerning the evaluation of $R(S_i; J)$:

- In both Equations (3.5)-(3.6) $R(S_i; J)$ is calculated conditioned on the event that at least one service in the considered list terminates. The probability of this event is equal to the service reliability $D(S_i; J)$.
- The expression for $R(S_i; J)$ in (3.6) is actually an approximation: the Jensen's inequality [79] ensures that the expectation of the minimum of random variables is lower than or equal to the minimum of the expectations, with the equality holding only in the deterministic case. Nevertheless, the approximation is accurate in case of small variances. In other cases a more suitable expression should be used, which would require the knowledge of the response time distribution, but this is out of the scope of this thesis.

From Equations (3.5)-(3.6), we see that the implementations of S_i according to the *alt* or *par_or* patterns have the same reliability when they use the same set of services. On the other hand, it is not difficult to verify (with some algebra) that *alt* has a lower cost than *par_or*, but a higher response time, since the sequential invocation used by *alt* means that on the average not all the selected services are invoked, but the response time of those invoked must be summed.

3.2.6 Optimization Policies

3.2.6.1 Per-Request Optimization

The main feature of the per-request optimization approach is that the QoS attributes are guaranteed for every invocation to the composite service, therefore in the per-request approach we need to identify the concrete service to be bound to each abstract service for all execution paths [12]. This approach currently does not support coordination

patterns, therefore only a single concrete service is chosen at a time to be bound to an abstract service. Hence, QoS attributes coincide with those of the selected concrete operation op_{ij} (Equation 3.4).

Workflow Model We assume that all the workflows of the composite services managed by the service broker have a single initial task, or that they start with a fork-join parallel sequence. Furthermore, we assume that for each conditional branch we know the probability of executing it; similarly, we assume to know the probability of reiterating loops.

The composite service graph is obtained by transforming the workflow of the composite service as in [12]. In particular, loops are peeled, *i.e.*, they are transformed in a sequence of branch conditions, each of which evaluates if the loop has to continue with the next iteration or it has to exit, according to the branch probability introduced above. A pre-requisite for loop peeling is the knowledge of the maximum number of supposed iterations. We calculate this value as the *p-percentile* of the distribution of reiterating the loop. After loop peeling, the composite service can be modeled as a Directed Acyclic Graph (DAG). As in [12], we define:

- *Execution path*. An execution path ep_n is a multiset of tasks $ep_n = \{S_1, S_2, \ldots, S_I\} \subseteq \mathcal{S}$, such that S_1 and S_I are respectively the initial and final tasks of the path and no pair $S_i, S_j \in ep_n$ belongs to alternative branches. We need a multiset rather than a simple set because a single task may appear several times in the execution path. An execution path may also contain parallel sequences, but it does not contain loops, which are peeled. A probability of execution p_n is associated with every execution path and can be calculated as the product of the probabili-

ties of executing the branch conditions included in the path. Similarly, the branch conditions that arise from loop peeling produce other execution paths.

- *Subpath*. A subpath of an execution path ep_n is a sequence of tasks $[S_1, \ldots, S_I]$, from the initial to the end task, that does not contain any parallel sequence. In other words, each branch b of a parallel sequence identifies a subpath inside the execution path ep_n. We denote a subpath by sp_b^n.

Therefore, the set of all the execution paths identifies all the possible execution scenarios of the composite service. The QoS constraints must hold for every execution path to guarantee the SLAs the service broker stipulated with its users.

QoS Attributes Computation Given the service selection policy $\boldsymbol{x}$ and the execution paths that arise from the composition logic, we can calculate the QoS attributes of each abstract task and then the overall QoS attributes of the composite service.

Let r_{ij} be the response time of the concrete service cs_{ij}, d_{ij} its reliability, and c_{ij} its cost. The worst case QoS values of the abstract task S_i, namely, the worst case response time R_i, the reliability D_i, and the worst case cost C_i, are given by the following expressions:

$$R_i^w = \max_{j \in \Im_i} r_{ij} x_{ij} \tag{3.7}$$

$$D_i^w = \min_{j \in \Im_i} d_{ij} x_{ij} \tag{3.8}$$

$$C_i^w = \max_{j \in \Im_i} c_{ij} x_{ij} \tag{3.9}$$

where x_{ij} is a binary variable indicating whether the concrete service cs_{ij} is bound to the abstract task S_i.

Using these formulas and the notion of execution paths and subpaths, we can calculate the QoS attributes along each execution path itself, using the aggregation formulas presented in [12]. We denote by R_n the maximum response time of the execution path ep_n, with D_n its minimum reliability, and with C_n its maximum cost; these are, in other words, the QoS attributes values calculated in the worst case scenario. They are:

$$R_n = \max_{sp_b^n \in ep_n} \sum_{S_i \in sp_b^n} R_i^w \tag{3.10}$$

$$D_n = \prod_{S_i \in ep_n} D_i^w \tag{3.11}$$

$$C_n = \sum_{S_i \in ep_n} C_i^w \tag{3.12}$$

While the cost and the reliability are simply obtained, respectively, as a sum and as a multiplication of the QoS attributes of each abstract task in the execution path, the matter is slightly different for the response time. Indeed, the response time of an execution path is equal to the response time of the longest subpath inside the execution path itself.

Optimization Problem The per-request optimization problem is formulated as a Mixed Integer Linear Programming (MILP) problem. We denote with the vector $\boldsymbol{x} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m]$ the optimal policy for a request to the composite service, where each entry $\boldsymbol{x}_i = [x_{ij}]$, $x_{ij} \in \{0, 1\}$, $i \in \mathcal{S}$, $j \in \Im_i$, denotes the adaptation policy for task S_i and the constraint $\sum_{j \in \Im_i} x_{ij} = 1$ holds. That is, x_{ij} is the decision variable equal to 1 if task S_i is implemented by concrete service cs_{ij}, 0 otherwise. Assume that the per-request policy $\boldsymbol{x}$ determines that for a given request $\boldsymbol{x}_i = [0, 0, 1, 0]$. According to this policy, for S_i the broker binds the request to cs_{i3}.

Following the per-request strategy in [12], we need to consider all the possible *execution paths* derived from the workflow.

The general goal of the optimization problem is to maximize the aggregated QoS value, considering all of the possible execution scenarios, i.e., all the execution paths arising from the business process. For simplicity's sake, in the formulation below we consider that the service broker's goal is to minimize for each request the response time of the composite service it offers.

$$\text{Problem } \textbf{per-request: min } \sum_{ep_n} p_n * R_n(\boldsymbol{x})$$

$$\textbf{subject to: } R_n(\boldsymbol{x}) \leq R_{max} \quad \forall ep_n \tag{3.13}$$

$$\log D_n(\boldsymbol{x}) \geq \log D_{min} \quad \forall ep_n \tag{3.14}$$

$$C_n(\boldsymbol{x}) \leq C_{max} \quad \forall ep_n \tag{3.15}$$

$$x_{ij} \in \{0,1\} \ \forall j \in \Im_i, \sum_{j \in \Im_i} x_{ij} = 1 \quad \forall i \in \mathcal{S} \tag{3.16}$$

We note that the minimization of the response time is only one of the possible objective functions that can be used, depending on the utility goal of the broker. An alternative expression can be found in [12], where the objective function is formulated using the weighted z-scores of QoS attributes.

3.2.6.2 Per-Flow Optimization

Differently from the per-request optimization approach, the per-flow optimization does not aim to satisfy QoS attributes for each invocation to the concrete service, rather it aims at satisfying them on the average, over a flow of requests. This main characteristic gives to the approach much more flexibility than the per-request one, allowing it to realize a probabilistic binding towards concrete services, therefore realizing a much

better load balancing over all the possible implementations. On the other and, this approach is unable to satisfy strict QoS constraints on every single request to the composite service. Preliminary steps for the definition of the per-flow optimization strategy presented in this section can be found in [24, 25].

Workflow Model For each service class $k \in K$ MOSES builds and maintains a labeled tree $T = (V, E, \mathcal{L})$, where V, E and $\mathcal{L}$ are the tree nodes, edges and labels, respectively. T is derived from the syntax tree that describes the production rules used to generate the composite service, by simply collapsing the S and C nodes. The leaf nodes of T are thus associated with tasks, while its internal nodes are associated with composition rules. Hence, for each non root node $v \in V$, its parent node $f(v)$ denotes the composition rule within which v occurs.

The set $\mathcal{L}$ of edges is defined as follows. Each edge $(f(v), v) \in E$ is labeled with $\ell^k(f(v), v)$, the expected number of times v is invoked within $f(v)$ for a class k request:

- if $f(v)$ is the seq or par_and composition rule then $\ell^k(f(v), v) = 1$;
- if $f(v)$ is the $loop$ rule, $\ell^k(f(v), v))$ is the average number of times the loop body is executed;
- if $f(v)$ is the sel rule, $\ell^k(f(v), v)$ corresponds to the probability that v is executed.

MOSES performs a monitoring activity to keep these values up to date. Figure 3.6 shows the tree T maintained by MOSES for the composite service depicted in Fig. 3.4 (labels equal to 1 are omitted).

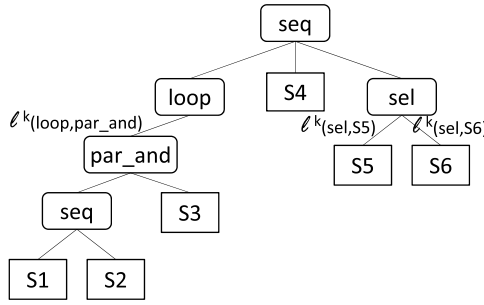


Figure 3.6: Composite service labeled tree.

QoS Attributes Computation Based on the workflow model, following well known QoS composition rules [30], we can derive the overall composite service QoS attributes $R^k(\boldsymbol{x})$, $C^k(\boldsymbol{x})$ and $D^k(\boldsymbol{x})$ (defined at the beginning of Section 3.2.5), given $R^k(S_i;\boldsymbol{x})$, $C^k(S_i;\boldsymbol{x})$ and $D^k(S_i;\boldsymbol{x})$, $1 \leq i \leq m$. Table 3.4 shows these rules, where for each node $v \in V$ we denote by $d(v)$ the (possibly empty) set of its children. These rules define a visit algorithm of the labeled tree T, from which we obtain:

$$Z^k(\boldsymbol{x}) = \mathrm{Z}^k(root;\boldsymbol{x})$$

$Z = C|\log D|R$, where *root* denotes the root node of T.

From the rules of Table 3.4 we now derive closed form expressions for the QoS attributes of the composite service, that will provide the basis for the optimization problem formulation of the next Section. In these expressions, for each node $v \in V$, we write $v \prec u$ if node v is a descendant of node u.

Cost and Reliability. For these attributes, from the recursive rules of Table 3.4, it is easy to realize that

Table 3.4: Recursive rules to calculate the average value of the QoS attributes of a composite service according to the per-flow workflow model.

node $v \in V$	**QoS rules** (where $\mathtt{Z}^k = \mathtt{C}^k \mid \log \mathtt{D}^k \mid \mathtt{R}^k$)
seq	$\mathtt{Z}^k(v;\boldsymbol{x}) = \sum_{u \in d(v)} \mathtt{Z}^k(u;\boldsymbol{x})$
$loop$	$\mathtt{Z}^k(v;\boldsymbol{x}) = \ell^k(v,d(v))\mathtt{Z}^k(d(v;\boldsymbol{x}))$
sel	$\mathtt{Z}^k(v;\boldsymbol{x}) = \sum_{u \in d(v)} \ell^k(v,u)\mathtt{Z}^k(u;\boldsymbol{x})$
par_and	$\mathtt{C}^k(v;\boldsymbol{x}) = \sum_{u \in d(v)} \mathtt{C}^k(u;\boldsymbol{x})$ $\log \mathtt{D}^k(v;\boldsymbol{x}) = \sum_{u \in d(v)} \log \mathtt{D}^k(u;\boldsymbol{x})$ $\mathtt{R}^k(v;\boldsymbol{x}) = \max_{u \in d(v)} \mathtt{R}^k(u;\boldsymbol{x})$
S_i	$\mathtt{Z}^k(u;\boldsymbol{x}) = Z^k(S_i;\boldsymbol{x})$

$$C^k(\boldsymbol{x}) = \sum_{i=1}^{m} \big(\prod_{j \succeq S_i} \ell^k(f(j),j) \big) C^k(S_i;\boldsymbol{x}) = \sum_{i=1}^{m} V_i^k C^k(S_i;\boldsymbol{x}) \tag{3.17}$$

and

$$\begin{aligned} \log D^k(\boldsymbol{x}) &= \sum_{i=1}^{m} \big(\prod_{j \succeq S_i} \ell^k(f(j),j) \big) \log D^k(S_i;\boldsymbol{x}) \\ &= \sum_{i=1}^{m} V_i^k \log D^k(S_i;\boldsymbol{x}) \end{aligned} \tag{3.18}$$

where $V_i^k = \prod_{l \succeq S_i} \ell^k(f(l),l)$, $S_i \in V$, is the expected number of times task S_i is invoked by the composite service for a service class k user.

Response Time. For $R^k(\boldsymbol{x})$, we need to account for the fact that the overall response time of the par_and pattern is the largest response time among its component tasks. As a consequence, the response time is no longer additive and we cannot derive an expression analogous to (3.17). In this case, we obtain a recursive set of expressions for the response time, whose number is linear in the number of par_and composition

patterns in the process. To this end, we first introduce the notion of *direct descendant* among nodes in V. We say that a node $v \in V$ is a direct descendant of $u \in V$, denoted by $v \prec_{dd} u$, if $v \prec u$ and for any other node $w \in V$, $v \prec w \prec u$ implies $w \neq par\_and$, *i.e.*, if there is no node labelled par_and in the path from v to u. In other words, a node $v \in V$ is said to be a direct descendant of u if task/pattern v is nested within the composition pattern u, but, within u, it is not nested within a par_and pattern.

Let $\Pi \subset V$ denote the set of nodes corresponding to par_and activities. We have the following result for the response time R^k (the proof, which is a simple application on the recursive formulas of Table 3.4, can be found in [29]).

Theorem 1 *For QoS class $k \in K$, the response time R^k can be computed recursively as follows:*

$$R^k(\boldsymbol{x}) = \mathtt{R}^k(root; \boldsymbol{x}) \tag{3.19}$$

$$\mathtt{R}^k(v; \boldsymbol{x}) = \begin{cases} \max_{u \in d(v)} \mathtt{R}^k(u; \boldsymbol{x}) & v \in \Pi \\ \sum_{S_i \in V, S_i \prec_{dd} v} \frac{V_i^k}{V_v^k} R^k(S_i; \boldsymbol{x}) + & \\ \sum_{u \in \Pi, u \prec_{dd} v} \frac{V_u^k}{V_v^k} \mathtt{R}^k(u; \boldsymbol{x}) & v \notin \Pi \end{cases} \tag{3.20}$$

Theorem 1 provides the response time $R^k(v)$ of each composition pattern $v \in V$ and the composite service response time R^k, $k \in K$. Observe that if the par_and pattern is not present in the workflow, $\Pi = \emptyset$ and Equation (3.20) reduces to $R^k(\boldsymbol{x}) = \sum_{i=1}^{m} V_i^k R^k(S_i; \boldsymbol{x})$.

Optimization Problem The basic goal of the per-flow optimization strategy is to determine an adaptation policy $\boldsymbol{x}$ that allows it to meet its QoS objectives stated by the $\langle R^k_{\max}, C^k, D^k_{\min}, \lambda^k_u \rangle$ SLAs, given the constraints determined by the $\langle r_{ij}, c_{ij}, d_{ij}, L_{ij} \rangle$ SLAs. Within the possibly empty set of feasible $\boldsymbol{x}$'s that satisfy these constraints, MOSES wants to select the $\boldsymbol{x}$ that optimizes a given utility function. Depending on

the utilization scenario of MOSES, the utility function could be aimed at optimizing specific QoS attributes for the different service classes (*e.g.*, minimizing their average response time) and/or it could be aimed at optimizing the MOSES own utility, *e.g.*, minimizing the overall cost to offer the composite service (that would maximize the MOSES owner incomes). These different optimization goals could be possibly conflicting, thus leading to a multi-objective optimization problem. To deal with it we transform it into a single objective problem using for this purpose the Simple Additive Weighting (SAW) technique [49], which is the most widely used scalarization method. According to SAW we define the MOSES utility function $F(\boldsymbol{x})$ as the weighted sum of the (normalized) QoS attributes of all users. More precisely, let

$$Z(\boldsymbol{x}) = \frac{\sum_{k \in K} L^k Z^k(\boldsymbol{x})}{\sum_{k \in K} L^k} \tag{3.21}$$

where $Z = R|\log D|C$ is the expected overall response time, reliability and cost, respectively, and $L^k = \sum_u \lambda_u^k$ is the aggregated flow of class k requests. We define the utility function as follows:

$$F(\boldsymbol{x}) = w_r \frac{R_{\max} - R(\boldsymbol{x})}{R_{\max} - R_{\min}} + w_d \frac{\log D(\boldsymbol{x}) - \log D_{\min}}{\log D_{\max} - \log D_{\min}} + w_c \frac{C_{\max} - C(\boldsymbol{x})}{C_{\max} - C_{\min}} \tag{3.22}$$

where $w_r,\ w_d,\ w_c \geq 0$, $w_r + w_d + w_c = 1$, are weights for the different QoS attributes. $R_{\max}$ ($R_{\min}$), $D_{\max}$ ($D_{\min}$), and $C_{\max}$ ($C_{\min}$) denote, respectively, the maximum (minimum) value for the overall expected response time, cost and reliability. We will describe how to determine these values shortly. With these definitions, the optimization

problem can be formulated as follows:

$$\begin{aligned}
&\textbf{max } F(\boldsymbol{x}) \\
&\textbf{subject to: } C^k(\boldsymbol{x}) \leq C^k, \quad k \in K && (3.23)\\
&\log D^k(\boldsymbol{x}) \geq \log D^k_{\min}, \quad k \in K && (3.24)\\
&\mathtt{R}^k(root;\boldsymbol{x}) + T_{ovd} \leq R^k_{\max}, \quad k \in K && (3.25)\\
&\mathtt{R}^k(u;\boldsymbol{x}) \leq \mathtt{R}^k(v;\boldsymbol{x}), \quad u \in d(v), v \in \Pi, k \in K && (3.26)\\
&\mathtt{R}^k(v;\boldsymbol{x}) = \sum_{S_i \prec_{dd} v} \frac{V^k_i}{V^k_v} \sum_{J \in \Im i} x^k_{iJ} R(S_i;J) + \\
&\qquad + \sum_{u \in \Pi, u \prec_{dd} v} \frac{V^k_u}{V^k_v} \mathtt{R}^k(u;\boldsymbol{x}), v \notin \Pi, k \in K && (3.27)\\
&\sum_{k \in K} \sum_{J \in \Im i, j \in J} x^k_{iJ} V^k_i L^k \leq L_{ij}, \quad op_{ij} \in OP && (3.28)\\
&x^k_{iJ} \geq 0, J \in \Im i, \sum_{J \in \Im i} x^k_{iJ} = 1, \quad 1 \leq i \leq m, k \in K && (3.29)\\
&x^k_{i_1 j_1} = x^k_{i_2 j_2} \quad op_{i_1 j_1}, op_{i_2 j_2} \in OP(cs_l) && (3.30)\\
&\qquad\qquad S_{i_1}, S_{i_2} \in \mathcal{S}_l, |\mathcal{S}_l| > 1, k \in K
\end{aligned}$$

Equations (3.23)-(3.27) are the QoS constraints for each class on the cost, reliability and response time. The constraints (3.25)-(3.27) for the response time are directly derived from (3.20). The additional term T_{ovd} accounts for the overhead introduced by the broker itself in managing the system. Equations (3.28) are constraints on the operations load and ensure that the system managed by MOSES does not exceed the volume of invocations agreed with the providers of those operations. The LHS of (3.28) is the volume of invocations of operation op_{ij} under adaptation policy $\boldsymbol{x}$. It is the sum over all service classes of the per class number of invocations per unit time of a given operation op_{ij} (the second summation is over all the implementations J in which j occurs). The RHS of (3.28) is the maximum load L_{ij} negotiated with the provider of the operation. Equations (3.29) are the functional constraints. Finally, Equations (3.30)

are the stateful constraints which basically require that, for stateful tasks, the fraction of requests that are bound to different operations of the same concrete service must be the same. Remember that if S_i is stateful, we only use the service selection adaptation technique; in this case J takes values only in $\mathcal{OP}_i$.

The maximum and minimum values of the QoS attributes in the objective function (3.22), used to get a normalized value, are determined by replacing $Z^k(x)$, $Z = R|\log D|C$ in Equation (3.21) with the maximum and minimum value that the QoS attributes can attain. $R_{\max}$, $C_{\max}$, and $D_{\min}$ are simply expressed respectively in terms of $R^k_{\max}$, C^k, and $D^k_{\max}$. For example, the maximum cost is given by $C_{\max} = \frac{\sum_{k\in K} L^k C^k_{\max}}{\sum_{k\in K} L^k}$. Similar expressions hold for $R_{\max}$ and $D_{\min}$. $R_{\min}$, $C_{\min}$, and $D_{\max}$ are similarly expressed in terms of the $R^k_{\min}$, $C^k_{\min}$, and $D^k_{\max}$, the minimum response time, minimum cost and maximal reliability that can be experienced by a class k request. For instance, $C^k_{\min} = \sum_{i=1}^{m} V^k_i C^*(S_i)$ where $C^*(S_i) = min_{J\in\Im_i} C(S_i; J)$ is the minimum cost implementation of task S_i. Similar expressions hold for $R^k_{\min}$ and D^k_{max}.

We conclude by observing that the per-flow optimization problem is a Linear Programming (LP) problem which can be efficiently solved via standard techniques.

3.2.6.3 Load-Aware Per-Request Optimization

The load-aware per-request policy exploits the multiple available implementations of each abstract task, and realizes a runtime probabilistic binding. In this way, different concurrent requests to the same abstract task are bound to different concrete services, realizing a randomized load balancing, in a way similar to the per-flow solutions presented in Section 3.2.6.2. At the same time, however, the QoS constraints are ensured *for each request* that the user submits like the per-request approach presented in Section

3.2.6.1. This approach currently does not support coordination patterns, therefore only a single concrete service is chosen at a time to be bound to an abstract service. Hence, QoS attributes coincide with those of the selected concrete operation op_{ij} (Equation 3.4).

Randomized Load Balancing The core of the load-aware per-request selection policy is the randomized load balancing of the requests directed to each abstract task S_i, so that they are switched to multiple concrete services implementing it. The load balancing is tuned on the basis of the capacity of each concrete service cs_{ij}, defined by the parameter L_{ij}, and of the rate of requests submitted by the users to S_i. As we already mentioned, an abstract task is bound by the broker to a set of concrete services, each one having an associated probability. So, at abstract task binding time, only one of these services is probabilistically chosen. As a consequence, only a fraction of the incoming requests is switched to a given concrete service cs_{ij}, and this fraction depends on the probability x_{ij} determined by the broker. We define a service selection policy as the set of all these probabilities, that we represent with the vector $\boldsymbol{x} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m]$, where for each entry $\boldsymbol{x}_i = [x_{ij}], i \in \mathcal{S}$, $j \in \Im_i$, the constraints $x_{ij} \in [0, 1]$ and $\sum_{j \in \Im_i} x_{ij} = 1$ hold. Our idea is to drive the value of the x_{ij} probabilities, forcing on each x_{ij} an upper bound P_{ij}, so that the fraction of requests switched by the broker to the concrete service cs_{ij} does not overload it. The upper bound P_{ij} is calculated through the ratio $P_{ij} = \dfrac{L_{ij}}{\lambda_i}$, where λ_i is the actual request rate to the abstract task S_i and L_{ij} is the load threshold for cs_{ij}. If P_{ij} is greater than 1, it means that there is no upper bound because cs_{ij} is able to satisfy all the incoming requests to S_i on its own. Vice versa, if P_{ij} is less than 1, cs_{ij} alone cannot satisfy all the requests directed to S_i but it must be backed by other concrete services, so that their overall capacity can

sustain the submitted load.

Workflow Model The workflow model used by the load-aware per-request optimization is the same used by the per-request optimization policy, already described in Section 3.2.6.1. Such a workflow model correctly captures the need of having a worst-case scenario, indispensable when QoS attributes have to be guaranteed for every request to the composite service.

QoS Attributes Computation Given the service selection policy $\boldsymbol{x}$ and the execution paths that arise from the composition logic, we can calculate the QoS attributes of each abstract task and then the overall QoS attributes of the composite service. We are interested in the average QoS perceived by the users as well as in its worst case value. As discussed below, we need both these values to maximize the broker utility function and satisfy the QoS constraints.

Let r_{ij} be the response time of the concrete service cs_{ij}, d_{ij} its reliability, and c_{ij} its cost. The average QoS values of the abstract task S_i, namely, the average response time R_i, the reliability D_i, and the average cost C_i, are given by the following expressions:

$$R_i = \sum_{j \in \Im_i} r_{ij} x_{ij} \tag{3.31}$$

$$D_i = \sum_{j \in \Im_i} d_{ij} x_{ij} \tag{3.32}$$

$$C_i = \sum_{j \in \Im_i} c_{ij} x_{ij} \tag{3.33}$$

The worst case QoS values, denoted by R_i^w, D_i^w, and C_i^w, are given by:

$$R_i^w = \max_{j \in \Im_i} r_{ij} y_{ij} \tag{3.34}$$

$$D_i^w = \min_{j \in \Im_i} d_{ij} y_{ij} \tag{3.35}$$

$$C_i^w = \max_{j \in \Im_i} c_{ij} y_{ij} \tag{3.36}$$

where y_{ij} is a binary variable indicating whether the concrete service cs_{ij} can be ever bound to the abstract task S_i, *i.e.*, $y_{ij} = 1$ if $x_{ij} > 0$ and 0 otherwise.

Using these formulas and the notion of execution paths and subpaths, we can calculate the QoS attributes along each execution path itself, using the aggregation formulas presented in [12]. Note that the same formulas apply both for the average case and for the worst case; therefore, for the sake of simplicity, we show only the latter case. We denote by R_n the maximum response time of the execution path ep_n, with D_n its minimum reliability, and with C_n its maximum cost; these are, in other words, the QoS attributes values calculated in the worst case scenario. They are:

$$R_n = \max_{sp_b^n \in ep_n} \sum_{S_i \in sp_b^n} R_i^w \tag{3.37}$$

$$D_n = \prod_{S_i \in ep_n} D_i^w \tag{3.38}$$

$$C_n = \sum_{S_i \in ep_n} C_i^w \tag{3.39}$$

While the cost and the reliability are simply obtained, respectively, as sum and multiplication of the QoS attributes of each abstract task in the execution path, the matter is slightly different for the response time. Indeed, the response time of an execution path is equal to the response time of the longest subpath inside the execution path itself.

Optimization Problem Given a composite service P, the goal of the service broker is to find a selection policy $\boldsymbol{x}$ that ensures the QoS constraints for every execution

scenario, *i.e.*, for each execution path ep_n that arises from P, while realizing the randomized load balancing. The selection policy $\boldsymbol{x}$ is calculated by solving a suitable optimization problem. We formulate this optimization problem as a Mixed Integer Linear Problem (MILP), with the following decision variables:

- x_{ij}: it takes value in the range $[0, 1]$ and represents the probability that the concrete service $cs_{ij} \in \Im_i$ is bound to the abstract task S_i; it is used to drive the randomized load balancing.

- y_{ij}: it is equal to 1 if cs_{ij} is bound to S_i with a given probability defined by x_{ij}, 0 otherwise. We use it to ensure that the QoS constraints are met.

While the QoS constraints are evaluated using the worst case values of the QoS attributes for each abstract task, the objective function is maximized using the average values, because it is the value that is expected along multiple executions of the composite service. In particular, the optimization problem maximizes the aggregated QoS values, which are calculated over all the possible execution paths that arise from the composite service workflow, taking into account the relative probability p_n. We obtain the aggregated values by applying the Simple Additive Weighting (SAW) technique as scalarization method.

In the first phase, each quality dimension along an execution path is normalized according to the following formulas, depending on whether the QoS attribute is a positive (3.40) or a negative (3.41) one. A QoS attribute is defined positive (negative) if the greater the value is, the greater (lower) the quality of that attribute. Reliability is an example of positive attribute (the higher the reliability, the better the quality is), while response time is an example of negative attribute (the lower the response time,

the better the quality is).

$$z_n^h(\boldsymbol{x}) = \begin{cases} \dfrac{q_n^h(\boldsymbol{x}) - \min q_n^h}{\max q_n^h - \min q_n^h}, & \text{if } \max q_n^h \neq \min q_n^h \\ 1, & \text{if } \max q_n^h = \min q_n^h \end{cases} \tag{3.40}$$

$$z_n^h(\boldsymbol{x}) = \begin{cases} \dfrac{\max q_n^h - q_n^h(\boldsymbol{x})}{\max q_n^h - \min q_n^h}, & \text{if } \max q_n^h \neq \min q_n^h \\ 1, & \text{if } \max q_n^h = \min q_n^h \end{cases} \tag{3.41}$$

In the above formulas, $q_n^h(\boldsymbol{x})$ is the h-th quality dimension value calculated over the execution path ep_n using the selection policy $\boldsymbol{x}$. $\max q_n^h$ and $\min q_n^h$ are its maximum and minimum values and can be estimated across several composite service executions.

In the second phase a score is obtained using a weighted sum of the normalized quality attributes, as follows:

$$score_n = \sum_h w_h z_n^h(\boldsymbol{x}) \tag{3.42}$$

where the weight w_h specifies the relative importance that the broker assigns to a QoS attribute with respect to the others.

Finally, the objective function is obtained using the following weighted formula:

$$F(\boldsymbol{x}) = \sum_n p_n score_n(\boldsymbol{x}) \tag{3.43}$$

The optimal service selection policy $\boldsymbol{x}$ can be obtained solving the following optimization problem (for sake of simplicity, we use n instead of ep_n):

$$\begin{aligned}
& \textbf{max } F(\boldsymbol{x}) \\
\textbf{subject to: } & \sum_{j \in \Im_i} x_{ij} = 1 && \forall i && (3.44)\\
& x_{ij} \leq P_{ij} && \forall i, \forall cs_{ij} \in \Im_i && (3.45)\\
& x_{ij} \leq y_{ij} && \forall i, \forall cs_{ij} \in \Im_i && (3.46)\\
& r_{ij} y_{ij} \leq R_i^w && \forall i, \forall cs_{ij} \in \Im_i && (3.47)\\
& d_{ij} y_{ij} \geq D_i^w && \forall i, \forall cs_{ij} \in \Im_i && (3.48)\\
& c_{ij} y_{ij} \leq C_i^w && \forall i, \forall cs_{ij} \in \Im_i && (3.49)\\
& \sum_{i \in sp_b^n} R_i^w \leq R_n && \forall sp_b^n \in ep_n, \forall n && (3.50)\\
& \sum_{i \in ep_n} log(D_i^w) = D_n && \forall n && (3.51)\\
& \sum_{i \in ep_n} C_i^w = C_n && \forall n && (3.52)\\
& R_n \leq R_{max} && \forall n && (3.53)\\
& D_n \geq log(D_{min}) && \forall n && (3.54)\\
& C_n \leq C_{max} && \forall n && (3.55)\\
& x_{ij} \in \Re^+ && \forall i, \forall cs_{ij} \in \Im_i &&\\
& y_{ij} \in \{0, 1\} && \forall i, \forall cs_{ij} \in \Im_i &&\\
& R_i^w, D_i^w, C_i^w \in \Re^+ && \forall i &&\\
& R_n, C_n \in \Re^+ && \forall n &&\\
& D_n \in \Re^- && \forall n &&
\end{aligned}$$

Constraints (3.44) guarantee that the sum of the probabilities of choosing the concrete services is equal to 1 for each task S_i. Constraints (3.45) define the upper bound to the probability of choosing a concrete service cs_{ij}. These two constraints families implement the randomized load balancing policy. Constraints from (3.47) to (3.49) express the response time, reliability, and cost of every abstract task S_i in terms of

the worst concrete services that are selected to implement that task. Constraints (3.50) evaluate the response time of each execution path ep_n as the response time of its longest subpath sp_b^n, while constraints (3.51) and (3.52) refer respectively to the reliability and cost of each execution path ep_n. Finally, constraints from (3.53) to (3.55) are the QoS constraints to be fulfilled. We use the logarithm of the reliability instead of the reliability in our optimization problem because we need to linearize Equation (3.11) to put it in our MILP problem.

3.3 Monitor Phase

MOSES is a broker prototype for the SOA world. Since SOA makes heavy use of services that are offered by third party providers, even in presence of SLAs, there is no actual guarantee that the services abide to the negotiated QoS parameters as network overload, service overload, and/or power outages may cause a service to not respect the expected QoS level.

As a consequence, to provide QoS guarantees, we have to account for the actual services QoS, rather than the QoS values stated in the SLAs. To this end, we need to monitor the execution of the concrete services and to analyze the collected data, so to be able to determine whether a change in the QoS level has occurred and a new service selection needs to be determined by using the updated QoS parameters.

Figure 3.7 shows how MOSES fits into the taxonomy presented in Section 2.4.1. In the following, we discuss the methodologies we have adopted to monitor the concrete services and to analyze the collected data with particular reference to service response time, but the same applies for other QoS attributes.

3.3.1 Monitoring the Concrete Services

The QoS attributes stated in a SLA are the targets of our monitoring activity. The monitored data can be collected at two different locations: at the service provider side or at the broker side. The former is made possible when the service provider collects data for itself and makes them available to its clients, like the Amazon CloudWatch service. However, the most common solution adopted in the SOA context is to collect data on the service broker that manages the composite service [9, 15, 21, 66], because the monitoring service is hardly provided by the concrete services providers.

Generally speaking, the methodology used to collect the data can be either active, if the data are collected sending proper inputs to the monitored entities, or passive, if the data are collected without injecting additional load but rather observing the system behavior. We preferred the latter solution, because in the context of SOA applications each service invocation may have a cost. Another important question regards the frequency at which data are collected, i.e., after how many invocations of a service we measure the QoS. Clearly, a low frequency approach requires less computational power than a high frequency one, but we adopt this latter in order to react more quickly to a change in the QoS of a service. Therefore, our monitoring activity is quite simple: we measure and store the QoS of each single concrete service invocation on a continuous time basis.

3.4 Analyze Phase

Monitoring is useless if collected data is not properly analyzed to learn the actual behavior of concrete services.

Figure 3.8 shows how MOSES fits into the taxonomy presented in Section 2.4.2.

In this section we describe the online adaptive cumulative sum (Cusum) algorithm [72] for service response time monitoring and abrupt change detection. We did not use the standard Cusum algorithm because it has been designed to detect changes in stationary time series with known statistical characteristics: in a non-stationary context, like the SOA context, whereby the variance can exhibit significant variation over time, its performance is very poor [31]. Indeed, given a time series, its standard deviation is used to properly tune the Cusum algorithm. Furthermore, although this algorithm detects changes in the time series mean, one of main assumptions behind it is that the variance is always constant over time.

The online adaptive Cusum detector we design in MOSES [31] combines an Exponential Weighted Moving Average (EWMA) filter to tracks the slow varying response time series average with a two-sided Cusum test to detect abrupt changes in the series average which cannot be timely accounted by EWMA filter.

We consider the following tracking EMWA filter:

$$\mu_i = \alpha y_i + (1 - \alpha)\mu_{i-1} \tag{3.56}$$

where μ_i represents the *i-th* current estimate of the average response time and y_i represents the *i-th* collected response time sample and α is a small constant. To timely detect the occurrence of significant changes, the Cusum algorithm uses two variables, g^+ and g^-, to detect positive and negative changes, respectively, which measure positive and negative deviation of the time series with respect its average value. They are initialized to 0 and updated at each step as follows:

$$g_i^+ = max\{0, g_{i-1}^+ + y_i - (\mu_i + K^+)\} \tag{3.57}$$

$$g_i^- = max\{0, g_{i-1}^- + (\mu_i - K^-) - y_i\} \tag{3.58}$$

where K^+ (K^-) is the smallest shift we want to detect on the leading (trailing) edge. In our experiments, we set it equal to 25% of the response time stated in the SLOs of the monitored services. A change is detected whenever g_i^+ or g_i^- are greater than a suitable threshold H^*, which represents a trade-off between detection delay and probability of false positive. To compute H^* we followed the approach in [31], which ensures a small probability of false detection (measured in terms of expected number of samples between false positives, we set to 1000). This requires the numerical inversion of the Siegmund approximation [72] which typically yields $H^* \approx 5\sigma_y$, where σ_y is the time series standard deviation. Since σ_y is unknown, we resort to a widely adopted approximation, which basically replaces the standard deviation with the estimate of the mean deviation $E[|y_i - E[y]|]$:

$$\sigma_i = \beta|y_i - \mu_i| + (1 - \beta)\sigma_{i-1} \tag{3.59}$$

where we set β to 0.5.

Whenever an abrupt change is detected, the average response time is updated according to the following equations [72]:

$$\mu_i = \begin{cases} \mu_{i-1} + K + g_i^+/N^+ & \text{if } g_i^+ > H^* \\ \mu_{i-1} - K - g_i^-/N^- & \text{if } g_i^- > H^* \end{cases} \tag{3.60}$$

in place of Equation (3.56), where N^+ (N^-) is the number of samples since the last time g_i^+ (g_i^-) was equal to zero. Upon a change detection, g_i^+ and g_i^- are reset to 0.

The estimates of the services response time, obtained via Equation (3.56) (or (3.60) when an abrupt change is detected), are used by the broker in the actual formulation of the service selection optimization presented in Section 3.2.6.

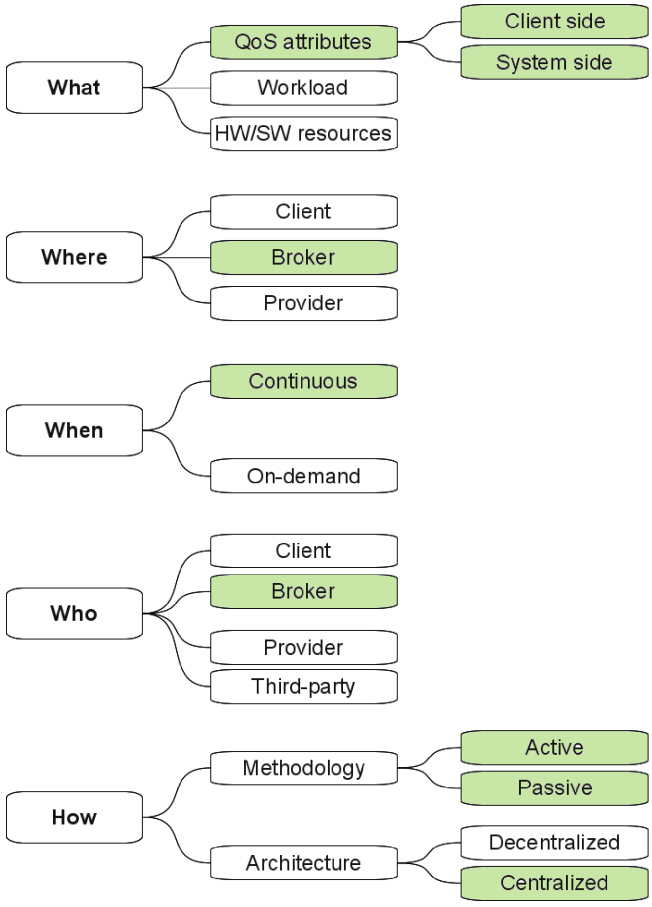


Figure 3.7: MOSES within the monitor taxonomy

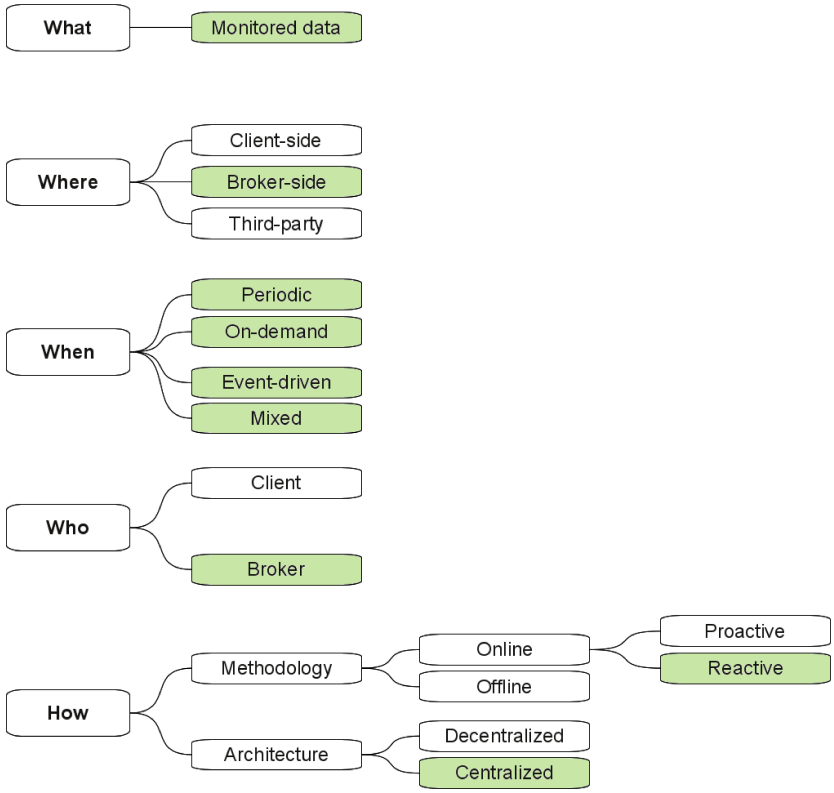


Figure 3.8: MOSES within the analyze taxonomy

4

Case Study: Design and Implementation of MOSES

Contents

4.1 IaaS Layer 90
4.1.1 Back-End Subnet 91
4.1.2 Choosing the IaaS Management Platform 101
4.1.3 Front-End Subnet 104
4.2 PaaS Layer 105
4.3 SaaS Layer 107
4.3.1 Overview of the MOSES Architecture 107
4.3.2 MOSES Design within OpenESB 111
4.3.3 MOSES Components 113
4.3.4 MOSES Clustered Architecture 114
4.3.5 MOSES Overheads 116

MOSES is a Cloud broker service: it belongs to the Software as a Service (SaaS) layer of the Cloud Computing stack [99], therefore it has been designed in order to be easily deployed into any public Cloud provider. However, since one of the target of our work is the evaluation of the performance of MOSES, and since it is very difficult to obtain accurate performance measurements on public clouds [2], we opted for building our own private cloud infrastructure to deploy MOSES in. In this way we had the opportunity to test MOSES without any external noise, thus providing baseline performance measurements that can be taken into account for comparisons with

performances on public clouds.

In the following we will first present a general purpose architecture for building a reliable, scalable, flexible, and modular private cloud that exploits virtualization technologies at different levels. In our specific case, the architecture has been entirely implemented using Free Software and standard protocols, thus realizing a truly open Infrastructure as a Service (IaaS). The obtained IaaS can be used to offer a variety of services that span from web applications and web services to soft real-time applications.

In the MOSES case, the private IaaS has been dedicated to the deployment of several instances of OpenESB (more details in Section 4.2), which enables a Platform as a Service (PaaS) layer. As well as for any other software used to build the IaaS, also OpenESB is Free Software. This software acts as a container for MOSES and offers many facilities to scale the environment. The design and implementation of MOSES will be presented in Section 4.3.

4.1 IaaS Layer

In this section we present a possible architecture that can be adopted by a prospective IaaS provider. It has the main objective to decouple storage resources from computational resources. Such a decoupling makes the entire infrastructure more scalable and more flexible because it allows the administrators to add or remove storage systems without any impact on the computational resources and vice-versa. The system architecture is partitioned in two logical levels, respectively named *back-end subnet* and *front-end subnet*. The back-end subnet is the architectural component aimed at both managing the storage resources and offering the storage as a service to the front-end

subnet, while the latter contains computational resources, which are then attached to storage services to obtain high-level services which are finally exposed to the end users. In addition to being a gateway to storage facilities, the back-end subnet also offers a centralized management of the services provided by the front-end subnet to the end users.

4.1.1 Back-End Subnet

We have designed the back-end subnet in such a way to obtain a clear separation between the services offered to the end users and the hardware/software needed by those services. That is, the back-end subnet provides a decoupling layer that allows the system administrators to manage the entire infrastructure in a modular and scalable way. The basic idea is as follows: a service is just an application that performs a task required by an end user, where each application requires several system resources (CPU, memory, disk, and network) with different needs. These resources have to be assigned by the system administrator in an transparent way with respect to the users, whose expectation is to have a fully functional service without taking care of their administrative issues. Therefore, the back-end subnet goal is to provide all the basic infrastructure facilities, such as a redundant storage management and a highly available management of computational resources, that should be managed as much as possible through a centralized tool to facilitate the system administrator tasks. In the following, we analyze each component of the back-end subnet.

4.1.1.1 Redundant Network Topology

A basic requirement of every dependable infrastructure is a dependable network topology. We introduce a redundant network topology that can (i) scale according to the

number of used physical links and (ii) automatically re-arrange itself by excluding malfunctioning links without any service interruption. The presented topology is generic and does not depend on the physical medium used to implement it: therefore, it is possible to use links based on Fiber Channel over optical fiber or Ethernet links on copper cables.

Left side of Figure 4.1 shows the components belonging to the back-end subnet. Starting from the left side, there are two storage subsystems (primary and replica) and two highly available storage gateways, that are interconnected by a redundant network topology, except for the connection between the storage gateways and the storage subsystems. The latter can also be replicated, whether the employed storage subsystems support multi-pathing; since not all storage subsystems support multi-path connections, we depicted only a single link, even if our implementation supports multi-pathing. We also observe that the network topology supports any number of links among its nodes: although we depicted only two links between the storage gateways and two links between each single storage gateway and the two switches, any number of links can be used, depending on the bandwidth and availability requirements. Specifically, in our implementation we use a single 4 Gbps Fiber Channel link between each storage subsystem and its storage gateway, while we use four 1 Gbps Ethernet links between each storage gateway and the border switches. Finally, the connection between the two storage gateways is a 4 Gbps Fiber Channel.

We designed the network topology with the goal to maximize both dependability and throughput, trying also to minimize the hardware costs, since storage subsystems are often expensive Storage Area Networks (SANs), that require expensive Host Bus Adapters (HBAs) and SAN switches (often switched fabrics) to connect to. The intro-

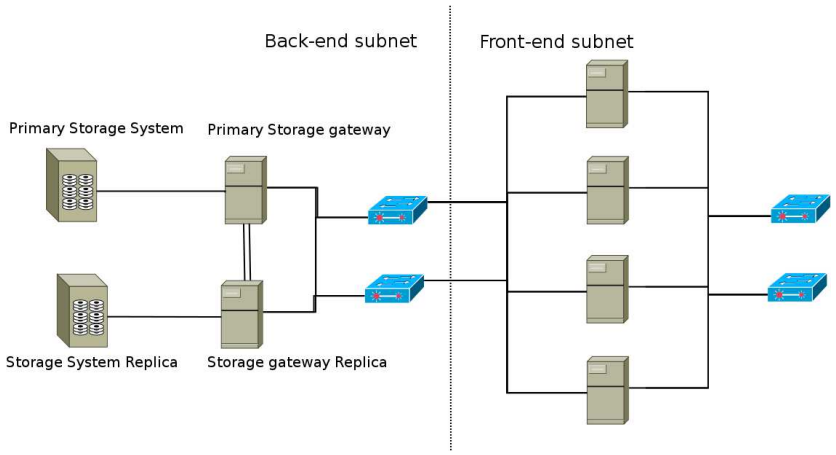


Figure 4.1: Network topology.

duction of two storage gateways decouples the native storage connections from where the storage needs to be attached. In this way, using the storage gateways as front-ends for the storage system, we reach the following objectives:

- to make the front-end servers unaware of the storage type;
- to aggregate different storage systems for reaching maximum expandability: that is, we can attach multiple heterogeneous storage subsystems to each storage gateway, hiding such a complexity to the front-end servers;
- to use arbitrarily complex storage management policies and replication;
- to minimize the hardware costs: we do not need to buy a lot of expensive HBAs and expensive fabrics because the two highly-available storage gateways are directly connected to the storage systems.

The channel aggregation of server NICs is realized using the Linux Bonding Driver.

It provides a method to aggregate multiple network interfaces into a single logical "bonded" interface. The behavior of the bonded interfaces depends on the selected operating mode. It is possible to distinguish between two families of operating modes: the first only provides high availability connections (by using either hot stand-by or broadcast links), while the second also adds load balancing mechanisms on bonded links. The only drawback of using load balancing mechanisms relies in a more difficult network debugging when problems arise.

The overall back-end subnet can be seen as a black-box *storage unit* because it offers a high-level and feature rich storage service to the front-end subnet; we note that two storage gateways may not be sufficient to manage large amounts of I/O when thousands of disks are attached, but storage units can be replicated by using a flat or a hierarchical approach. Therefore, storage units are a scalable way to implement a modular interface to storage subsystems, since each storage unit implicitly provides the same interface to the front-end subnet layer even if the actual implementation could be very different.

4.1.1.2 Redundant Storage

When designing an architecture for an IaaS provider, an important issue is to ensure data protection from system failures: although a short unavailability period of a given service can often be tolerable, data loss is certainly intolerable. To address this issue, we introduced a redundant storage schema in our infrastructure architecture.

Suppose that the primary storage subsystem is a SAN with several RAID arrays: each RAID array in a SAN can host multiple Logical Unit Names (LUNs), each of which is seen as a physical disk by the operating system of the storage gateway connected to the SAN. We manage to introduce the storage replication through a software

layer positioned just over the plain disk seen by the operating system and represented by *Distributed Redundant Block Devices* (DRBD) [40].

DRBD defines two roles in a replication scheme, named *primary* and *secondary*: only a node with primary role can access data if not using a distributed filesystem. We can use different working modes, that range from fully synchronous to asynchronous replication. With synchronous replication the filesystem on the active node is notified that the writing of the block is completed only when the block made it to both disks of the cluster. Using asynchronous replication, the entity that issued the write request is informed about completion as soon as the data is written to the local disk and to the local socket buffer. Finally, with memory-synchronous replication, the filesystem on the active node is notified that the writing of the block is completed when data is written to the local disk and has reached the write buffer of the remote server.

The choice of the working mode to be used strictly depends on the storages type and the interconnecting network. The best scenario is when we have a fast primary storage coupled with a fast[1] secondary storage, connected with a fast and reliable network. The worst scenario is when we have a fast primary storage with a relatively slow secondary storage, coupled with a slow and unreliable network (think to geographic data replication). We do not consider the scenario in which we have two poorly sized storage subsystems. Under the first scenario, synchronous DRBD replication is surely the best choice, because it ensures maximum data protection with a negligible performance loss. On the other hand, in the latter case we must take into account more variables, such as global disk speed, buffers, and network. If geographic replication is not needed and the infrastructure relies on a fast local network, we can choose either

[1] We consider a storage system to be *fast* when it is adequately sized to sustain the submitted workload.

memory-synchronous replication or asynchronous replication, depending on the global disk speeds and the buffers offered by the secondary storage controller: the choice has to be done carefully and really depends on the workload the storage is supposed to face. If we choose a memory-synchronous replication without having enough buffers we can introduce a bottleneck, but if we use asynchronous replication with a fast and adequately buffered secondary storage we are not working in an optimal way.

In our implementation, we have a fast SAN as primary storage and a slow NAS (a server with direct attached storage) as secondary storage, with a fast interconnecting network. From not reported experiments, we found that for this specific configuration the best operating mode is the asynchronous replication with network congestion control: whenever the active storage gateway detects a network congestion[2], it temporarily detaches the secondary storage, thus going in "Ahead/Behind" operational mode. In such a way, the bottleneck is temporarily detached from the infrastructure and data synchronization restarts as soon as the network congestion disappears; the data synchronization process only involves those data blocks that have been changed on the primary storage while it was in "Ahead" mode. This advanced configuration has been pursued to avoid buying an expensive secondary SAN thus lowering the total cost of ownership, but at the same time increasing the overall infrastructure reliability by adding a storage replica.

Similar considerations can be applied to each storage unit in our architecture. Although different storage units have the same network topology and the same external interface, they can have different interconnections and storage speeds. Therefore, we are free to choose different replication policies depending on the components of each

[2]With a fast and reliable network, a congestion is likely to happen only when secondary storage disks utilization is 100% and the buffer is full.

storage unit, offering a better QoS to users that are willing to pay more and a best-effort QoS to thrifty users.

4.1.1.3 Volumes Management

A modern storage system, besides being reliable, needs to be flexible. Volumes management introduces a considerable degree of flexibility, providing features like online volumes resizing, volumes snapshotting, and hot disk addition or removal. Volumes are considered by an operating system just like partitions, since we can use them to hold root filesystems or data directories (that is, they are seen as block devices). We briefly describe below the feature offered by logical volumes.

Online volumes resizing is a key feature: whenever we find out that we allocated less space than needed for a volume, it allows us to expand the space and, if the filesystem has the support, we can also hot-expand the filesystem without rebooting a server and therefore without service interruption.

Volumes snapshotting has a twofold use: we can snapshot a volume either to rapidly have new operating system images ready to use or to make consistent backups. Snapshots are nothing more than simple volumes, with their own allocated space, but they are originated from existing volumes and they use the Copy On Write (COW) [98] optimization strategy. Usually, when using volume snapshotting for operating system image cloning, we have a 1:N ratio of gold images and derived snapshots. Therefore, to avoid disk overload, it is necessary to have a read-only gold image and read-write snapshots, otherwise each single write on the source volume triggers the COW on each snapshot. On the other hand, when using volume snapshotting for consistent backups, we usually have a 1:1 ratio between snapshotted volumes and snapshots, also having the source volume read-write mounted and the snapshot volume read-only mounted. In

this case, we only need to take care of the snapshot size, that has to be enough to hold changes made on the source volume during the snapshot lifetime.

Disk addition and removal features are a must for a flexible architecture. When business grows, we can certainly expect an increase in the data to be stored. A disk addition feature allows to buy only needed disks from time to time, while a disk removal feature allows us to reduce the number of disks if they are no more needed and to replace old disks with newer ones without service interruption.

In our architecture we used Logical Volume Manager (LVM) to implement volumes management. LVM introduces three abstraction layers: physical volumes, volume groups, and logical volumes. LVM Physical Volumes (PVs) are simple partitions or physical disks initialized by LVM. After being initialized, PVs are aggregated to form Volume Groups. LVM Volume Groups (VGs) are an aggregation of PVs. This abstraction level let us overcome the single disk (or the single RAID array) capacity. LVM Logical Volumes (LVs) are flexible partitions built upon VGs and provide every feature described above. Since LVs are managed by the operating system just like partitions, LVM can be positioned either under or over DRBD. Our choice was to position LVM over DRBD on the primary storage and under DRBD on the secondary storage. The motivation is again the asymmetric configuration of the primary and secondary storage: the primary SAN can scale up to 99 disks; the secondary NAS can only scale up to 10 disks. Using larger disk on the NAS we can host DRBD resources over LVM volumes that turns out in consolidating the disk utilization.

4.1.1.4 Complete Storage Architecture

Figure 4.2 unveils the complete storage architecture we have realized: we can roughly divide the figure in two columns. The left column represents the primary storage sub-

system, while the right column represents the secondary storage. Reading the figure bottom-up, we find that disks on the primary storage are organized in a RAID array, over which we created a LUN seen by the operating system with the name `/dev/sdc`. Over the RAID array, we created the physical partition `/dev/sdc1`, backing device for the DRBD resource `imgos`. The latter contains the operating systems images that can be attached to virtual machines (VMs) deployed on the front-end servers. The DRBD device `/dev/drbd0p1` is initialized as a LVM PV and is then inserted into the `imgos_group` VG. `imgos_group` thus contains as many LVs as the number of operating systems images stored in this storage unit. Looking at the DRBD level, it is coupled with a peer DRBD level on the secondary storage, whose backing device is a LVM LV `imgos`, which belongs to a VG named `repdata` and whose backing devices are two partitions on two different RAID arrays[3].

The designed storage architecture ensures high data reliability and throughput, flexibility and easiness of management, low costs, and finally a single asynchronous backup source. Data reliability is ensured by replication: local replication is achieved through RAID arrays, while remote replication is achieved through DRBD. Whenever the secondary storage fails, nobody will notice it and if the primary storage fails, the secondary one instantly replaces it. High data throughput is allowed by an asynchronous replication with congestion control (actually the perceived write speed is the one provided by the primary, fast storage). Flexibility and easiness of management are provided by LVM; to reduce the costs, we built a replicated storage only using free software and with an economic secondary storage. Last but not least, an inter-

[3] Actually, two RAID arrays on the secondary storage are not required, but we added them because the server used as secondary storage could not boot over a logical disk with GPT partitioning schema, which is required by logical disks larger than 2 TB.

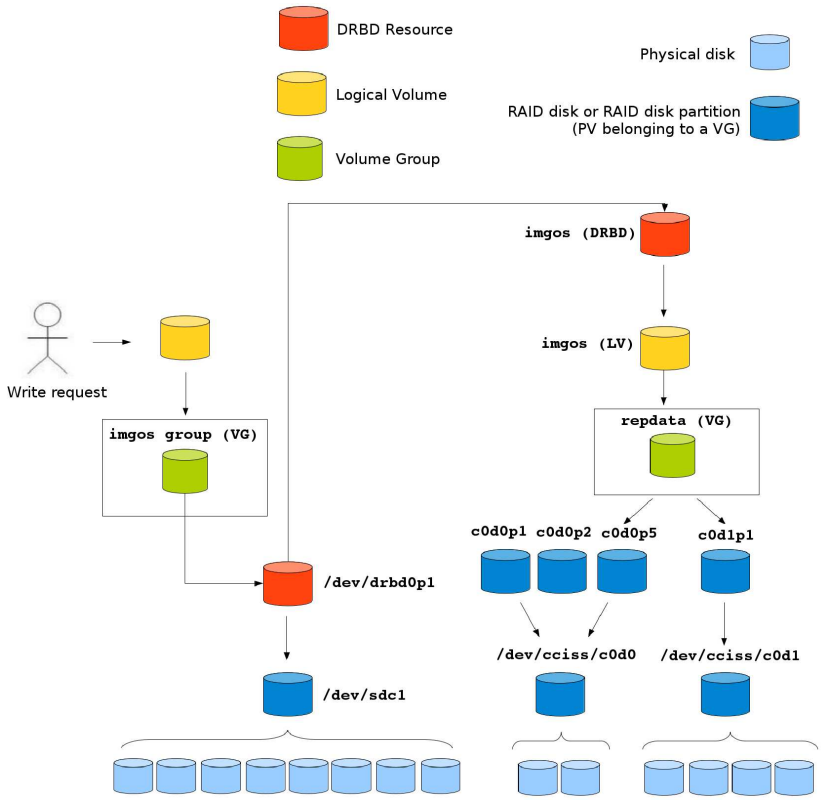


Figure 4.2: Data flow on the storage architecture.

esting feature of the storage architecture is the ability to have a single asynchronous backup source: thanks to the LVM LV backing device for the entire DRBD resource on the secondary storage, we can instantly take a snapshot of the `imgos` resource and, since snapshots never modify original data, we can safely mount volumes found in the `imgos` snapshot, thus having a stable view of the entire storage unit data. This ensures easy, consistent, and economic backups: we do not need to backup every single server,

but we can take them all together and backup them with a single backup application. This feature terribly lowers the effort of system administrators in keeping a working backup.

4.1.2 Choosing the IaaS Management Platform

In this section we briefly review and compare the major open-source platforms for the management of cloud infrastructures: Eucalyptus [81], OpenQRM [85], OpenNebula [84], Nimbus [80], and OpenStack [86] We classify the various platforms according to the abstraction level they provide with respect to the underlying hardware and the customization degree. More detailed analyses and comparisons of open-source cloud computing platforms can be found in [14, 93, 101, 102].

Eucalyptus (Elastic Utility Computing Architecture for Linking Your Programs To Useful Systems) is an open-source cloud management platform developed by the University of California that offers the highest abstraction level among the others, by letting the user choose only from a fixed set of VM templates. It provides a framework similar to Amazon Web Services, by implementing interfaces compatible with Elastic Compute Cloud (EC2) and Amazon Simple Storage Service (S3), also realizing a distributed storage system called *Warlus*, which is designed to imitate Amazon's S3 distributed storage. Whenever a new virtual machine instance is requested to Eucalyptus, its operating system is copied from the storage system to the physical server (from now on *Compute node*) which will execute it.

Nimbus offers a higher customization degree compared to Eucalyptus, but only from the administrators' point of view: it let them configure at a finer level the VM instances that will be offered to end-users. Like Eucalyptus, Nimbus implements a

storage system similar to Amazon S3. Both Nimbus and Eucalyptus are architected in order to decentralize resources: for this reason, each time a VM is requested, its image template is copied to the Compute node that will run the VM. In this way, both the platforms can achieve maximum scalability, as well as a good fault isolation because each Compute node is independent from the others. However, decentralizing storage systems also leads to a more complex management of replication policies as well as backup policies. For this reason, neither Eucalyptus nor Nimbus are compatible with our architecture.

OpenNebula is inspired by different principles compared to both Eucalyptus and Nimbus: it aims to provide a finer level of customizability for front-end users and its architecture tends to centralize resources by offering a central storage. A fine customizability level requires smart users because some details about the underlying infrastructure cannot be hidden anymore. Therefore, OpenNebula is more suited for private clouds where the entire environment is trusted and users have more skills. However, besides offering a central image storage, OpenNebula also provides the ability to run the image locally on the compute node, thus achieving the same flexibility of Eucalyptus and Nimbus.

The OpenStack project started in summer 2010 when RackSpace and NASA jointed their initial projects “CloudFiles” and “Nebula”, respectively. It is based on a distributed architecture composed by several modules, among which the compute, networking, storage, and dashboard. Its approach is similar to Eucalyptus because it lets the administrator define several VM templates and the user can only choose among them. However, differently from what happens in Eucalyptus, with OpenStack it is possible to configure both centralized or local storage for VM disks. At present, it is

the fastest growing free open source software for IaaS management.

The last platform we consider is OpenQRM, which provides the highest level of customizability, because it lets the administrators of the private cloud configure each single aspect of the datacenter. Its core is very small and its architecture is completely plugin-based, e.g., there are plugins to connect to different storage systems (NFS, iSCSI, AoE, etc.), as well as plugins to manage several virtualization technologies (primarily Xen and KVM, but also VMWare, Virtualbox, etc.). OpenQRM can be used just to administer the datacenter. Anyway, installing the *Cloud plugin*, we can exploit every typical cloud feature, like scheduled VM provisioning and de-provisioning and a pay-per-use model. Unlike the other platforms, OpenQRM lets the administrator of the cloud infrastructure configure the products offered to the cloud users at a finer granularity level: instead of configuring a fixed set of instances templates, it is possible to configure parameters like the quantity of CPU and RAM users are allowed to demand (with associated cost), as well as several possibility for data storage. Users can dynamically assemble their VM instances by choosing each single component from a drag-n-drop palette.

We chose OpenQRM for our infrastructure implementation, because it offers the higher customizability level, but our infrastructure is compatible with OpenNebula and OpenStack as well. Such a software can be added to our system architecture in two different positions. In case of a single storage unit, OpenNebula, OpenQRM or OpenStack can be installed on the two storage gateways; otherwise, in case of multiple storage units we need two additional servers at the back-end subnet boundaries to host the datacenter management software.

4.1.3 Front-End Subnet

The front-end subnet, represented on the right side of Figure 4.1, is composed by a bunch of servers with lots of RAM and CPU. The connection schema is the same used for the back-end subnet: each front-end server can be equipped with a number of NIC, in our case each server has 8 NICs (2 quad-port Gigabit Ethernet), where 4 links are directed to back-end switches and the remaining 4 links are directed to front-end switches. The channel aggregation is again achieved through the Linux bonding driver.

The front-end servers do not need any internal hard disk because they are network booted by OpenQRM, using DHCP, TFTP and iSCSI or NFS protocols with a minimal Linux distribution with only the software required to run a KVM virtual machine.

Because of virtualization, the front-end servers host a multi-level network stack (Figure 4.3). The first level is composed by several physical network devices (8 in our case), half of which is connected to back-end switches, while the other half is connected to front-end switches. The second level is given by link aggregation with bonding driver. We created two bond devices: `bond0` (directed to back-end switches) and `bond1` (directed to front-end switches). The first bond is used for storage access, while the second one is used for service delivery because the front-end switches are attached to a generic internet gateway. Finally, the third level is the bridge, which allows virtual machines to use networking. Virtual machines support two networking configurations: `bridge` and `NAT`. The simplest configuration is bridge, because it involves network layers from physical to Logical Link Control (LLC), belonging to the Data Link layer. However, because of its simplicity, bridging configuration can only be used when complex network segmentation and isolation are not needed, because they can only be achieved using VLANs. VLANs imply switches configuration and,

when the network becomes large, they can be very hard to manage. On the other hand, using NAT we raise the OSI layer to Network/Transport, so we can use more complex firewall rules to manage subnets communications at the price of losing the ability to use non TCP and non UDP applications.

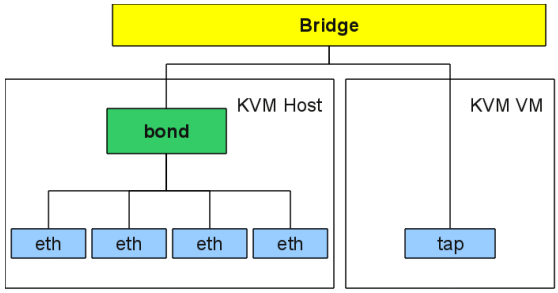


Figure 4.3: Network stack of the front-end servers.

Each front-end server has two network stacks: the first for data access and directed to storage gateways through back-end switches, and the latter for service delivery.

4.2 PaaS Layer

The PaaS layer is entirely implemented with OpenESB [83]. OpenESB is an Open Source project that delivers a platform for SOA business integration and Enterprise Application Integration (EAI). It is based on a large number of standards, such as Java EE, SOAP, WS-* and, in particular, JBI (Java Business Integration) [51]. JBI is a messaging-based pluggable architecture, whose components describe their capabilities through WSDL. Its major goal is to provide an architecture and an enabling framework that facilitates the dynamic composition and deployment of loosely coupled participating applications and service-oriented integration components. The key components of

the JBI environment are:

- the Service Engines (SEs) that enable pluggable business logic;
- the Binding Components (BCs) that enable pluggable external connectivity;
- the Normalized Message Router (NMR), which directs normalized messages from source to destination components according to specified policies.

OpenESB implements and extends JBI because it enables a set of distributed JBI instances to communicate as a single logical entity that can be managed through a centralized administrative interface. GlassFish application server is the default runtime environment, although OpenESB can be integrated in several JEE application servers.

Applications built around an ESB usually consist of multiple parts; therefore any business application may have multiple points of failure. For example, an application could consist in several components: a BPEL process, some EJB module, JMS and so on. Failure of any component may have a serious affect on the business applications availability. Therefore one important consideration is to identify and manage all single points of failure. Clustering is a way of deploying a number of components, allowing them to work together to improve availability and performance, compared to what can be achieved by deploying just one component. Clusters can provide added reliability using failover capabilities, which means that when a component fails, another takes over and provides the same set of services, but can also improve performances when multiple instances of the same component work together. This is done transparently so that clients are not aware of what happens behind.

OpenESB supports clustering by grouping together application server instances. Each cluster is considered to be a logical managed unit and instances share a common

configuration, host identical applications and can be managed and monitored centrally. Each JBI component is specifically designed to work in a cluster environment, thus providing a platform that eases the implementation of clustered applications. In Section 4.3 we will show how we designed and built MOSES following the JBI specifications with OpenESB.

4.3 SaaS Layer

In this section we present the architecture of MOSES. We will start with an overview of the MOSES components and then we will show how they are designed to fit into the JBI framework exposed by OpenESB. MOSES has been designed with scalability in mind: it can be completely executed by a single virtual machine when subject to small load, but it can also scale using the clustered architecture presented in Section 4.3.4. Finally, we will present an analysis of the overheads introduced by MOSES with respect to the execution of a workflow with static bindings and without runtime adaptation features.

4.3.1 Overview of the MOSES Architecture

Figure 4.4 shows the MOSES architecture, whose core components are organized in parts according to the MAPE-K cycle. The Execute part comprises the *Composition Manager*, *BPEL Engine*, and *Adaptation Manager*. The first component receives from the brokering service administrator a new BPEL process to be deployed inside MOSES and builds its corresponding behavioral model. To this end, it interacts with the Service Manager to identify the concrete services that implement the functionalities required by the service composition. Once created, the behavioral model, which also includes

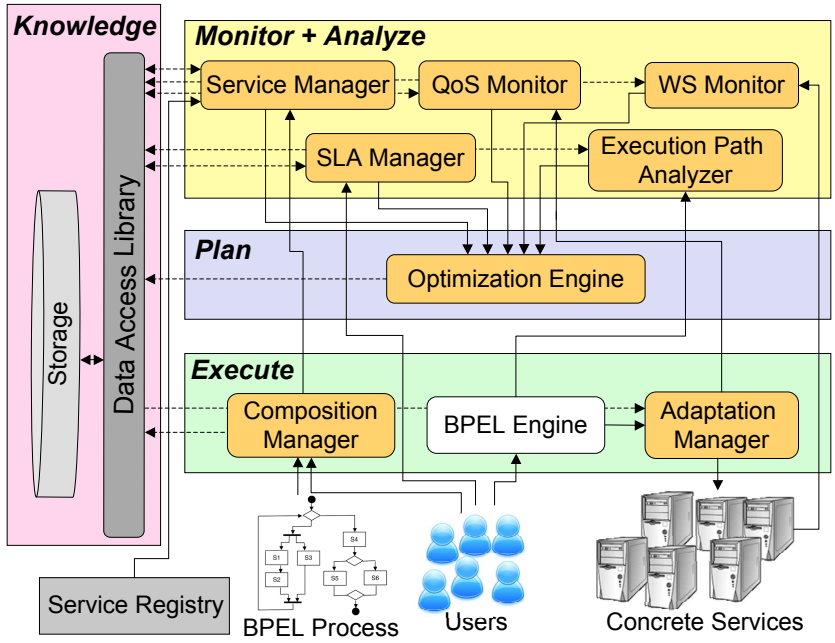


Figure 4.4: MOSES high-level architecture.

information about the discovered concrete services, is stored in the Knowledge part to make it accessible to the other system components.

While the Composition Manager is invoked rarely, the BPEL Engine and Adaptation Manager are the core modules for the execution and runtime adaptation of the composite service. The first is the software platform that actually executes the business process and represents the user front-end for the composite service provisioning. It interacts with the Adaptation Manager to invoke the proper component services: for each abstract functionality required during the process execution (i.e., `invoke` BPEL activity), the Adaptation Manager dynamically binds the request to the real endpoint that represents the service. The latter is identified by the solution of one of the imple-

mented optimization problems [12, 23, 27] and can be either a single service instance or a subset of service instances coordinated through some pattern. MOSES currently supports as coordination patterns the 1-out-of-n parallel redundancy and the alternate service [23]. With the former, the Adaptation Manager invokes the concurrent execution of the concrete services in the subset identified by the solution of the optimization problem, waiting for the first successful completion. With the latter, the Adaptation Manager sequentially invokes the concrete services in the subset, until either one of them successfully completes, or the list is exhausted.

The *Optimization Engine* realizes the planning aspect of the autonomic loop. It solves the optimization problem, which is based on the behavioral model initially built by the Composition Manager and instantiated with the parameters of the SLAs negotiated with the composite service users and the concrete services. The model is kept up to date by the monitoring activity carried out by the components in the Monitor-and-Analyze part. The problem solution provides indications about the adaptation actions that must be performed to optimize the use of the concrete services with respect to the utility goal of the brokering service and within the SLA constraints.

The Monitor-and-Analyze part comprises all the components that capture changes in the MOSES environment and, if they are relevant, modify at runtime the behavioral model and trigger a new adaptation plan. Specifically, the *QoS Monitor* collects and analyzes information about the QoS levels perceived by the composite service users and offered by the concrete services providers. The *WS Monitor* checks periodically the concrete services availability. The *Execution Path Analyzer* monitors variations in the usage profile of the composite service functionalities by examining the business process executed by the BPEL Engine; it determines the expected number of times that

each functionality is invoked by each service class. The *Service Manager* and the *SLA Manager* are responsible for the SLA negotiation processes in which the brokering service is involved. Specifically, the first negotiates the SLAs with the concrete services, while the latter is in charge to add, modify, and delete users SLAs and profiles. The SLA negotiation process towards the user side includes the admission control of new users [4]; to this end, it involves the use of the Optimization Engine to evaluate MOSES capability to accept the incoming user, given the associated SLA and without violating already existing SLAs. Since the Service and SLA Managers can determine the need to modify the behavioral model and solve a new instance of the optimization problem, we have included them within the Monitor-and-Analyze part.

In the current MOSES prototype, each component in the Monitor-and-Analyze part, independently from the others, senses the composite service environment, checks whether some relevant change has occurred on the basis of event-condition-action rules and, if certain conditions are met, triggers the solution of a new optimization problem instance. Tracked changes include the observed variations in the SLA parameters of the concrete services (QoS Monitor), addition/removal of concrete services corresponding to functionalities of the abstract composition (WS Monitor and Service Manager), variations in the usage profile of the functionalities in the abstract composition (Execution Path Analyzer) and, only for the per-flow optimization described in [23], the arrival/departure of a user with the associated SLA (SLA Manager).

Finally, the Knowledge part is accessed through the Data Access Library, which allows to access the parameters of the composite service operations and environment, among which the solution of the optimization problem and the monitored model pa-

[4]Currently the only service selection solution that supports admission control is the per-flow described in [23]

rameters.

4.3.2 MOSES Design within OpenESB

Each MOSES component is executed by one Service Engine, that can be either Sun BPEL Service Engine for executing the business processes logic and internal orchestration needs, or J2EE Engine for executing the business logic of all the MOSES components except the BPEL Engine. Developing the components with J2EE Engine improves the flexibility, because they can be accessed either as standard Web services or as EJB modules through the NMR.

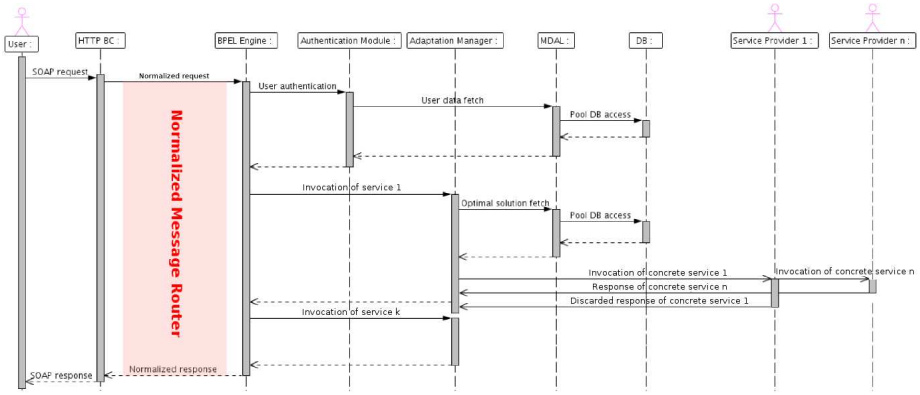


Figure 4.5: Typical execution flow in the ESB-based MOSES prototype.

The typical execution flow of a request to the composite service managed through the per-flow adaptation strategy is illustrated by the sequence diagram in Fig. 4.5. With per-request adaptation an initial invocation to the Optimization Engine is added to compute the optimal solution for the current request. As first step, the registered user issues a SOAP request to the MOSES front-end, that is the HTTP BC; the request format follows what expected by the BPEL process to whom the request is addressed.

The HTTP BC normalizes the HTTP request and sends it to the BPEL Engine through the NMR. Upon receipt of the message, the BPEL Engine de-normalizes the message and starts to serve the request. The first task performed within the process is the invocation of the authentication module (not shown in the high-level architecture of MOSES) to verify that the user issuing the request is properly registered. If not, an exception is forwarded to the user. Otherwise, for each invoke activity within the BPEL process, the Adaptation Manager reads the solution of the optimization problem from the storage layer and for that abstract functionality invokes the subset of concrete services using the coordination pattern as determined by the solution (Fig. 4.5 shows the use of the 1-out-of-n parallel redundancy pattern for one service invocation). Finally, when the response is ready for the user (these steps are not shown in Fig. 4.5), the BPEL Engine puts the response message on the NMR, the HTTP BC de-normalizes it, obtaining a plain SOAP response message that is finally forwarded to the user.

Alternative execution flows can be split in monitoring and administration flows. The former denotes each flow that is related to the resources monitoring and can trigger the execution of the Optimization Engine to determine a new optimal solution. The WS Monitor, QoS Monitor, and Execution Path Analyzer are periodically invoked by the Scheduler BC, and each of them can trigger the Optimization Engine when a new adaptation plan is needed. The Service Manager can be invoked either by the Scheduler BC or by the Composition Manager when new concrete services are needed. The SLA Manager is invoked by users when they register or establish new SLAs with MOSES; the Composition Manager is invoked by the MOSES administrator to manage new BPEL processes.

We observe that MOSES requires that only the BPEL Engine, the Adaptation Man-

ager and the storage layer must be up and running to complete the request-response cycle. When only these components work, the broker can orchestrate the composite service (although in a sub-optimal way, being not able to solve a new instance of the optimization problem), but it still succeeds in providing a response to the users.

4.3.3 MOSES Components

We analyze in detail only the Adaptation Manager and storage layer design, because these are the components that mostly influence the MOSES performance and scalability. We have designed and implemented all the other components, except the Service Manager; their detailed description can be found in [28]. We note that all inter-module communications exploit the NMR presence: message exchanges are faster than those based on SOAP communication, because they are "in-process", thus avoiding to pass through the network protocol stack. However, thanks to OpenESB we can expose every MOSES component as a Web service. The tasks of the Adaptation Manager are to modify the request payload in order to make it compatible with the subset of invoked concrete services and to invoke these services according to the coordination pattern determined by the solution of the optimization problem.

Being the Adaptation Manager the MOSES component that receives the highest request rate, its design is crucial for scalability and availability. We have investigated three alternative solutions for its implementation. The first realizes the component directly in BPEL, but we discarded it because the Sun BPEL Service Engine does not currently support the `forEach` BPEL structured activity with the attribute `parallel` set to yes. We needed this activity to realize in BPEL the 1-out-of-n coordination pattern. With the second alternative we investigated how to realize the Adaptation Manager as

a Java EE Web service. We found a feasible solution (based on the Provider interface offered by the JAX-WS API) but we discarded it because it causes a non negligible and useless performance overhead for the service invocation itself. The solution we finally implemented realizes the Adaptation Manager as a Java class which is directly invoked inside the BPEL process. The advantage is the higher communication efficiency and the consequent reduction of the response time perceived by the users of the composite service.

The storage layer represents a critical component of a multi-tier distributed system, because the right tradeoff between responsiveness and other performance indexes (like availability and scalability) has to be found. We have investigated various alternatives to implement the MOSES storage layer and decided to rely on the well-known relational database MySQL, which offers reliability and supports clustering and replication. However, to free the MOSES future developers from knowing the storage layer internals, we have developed a data access library, named MOSES Data Access Library (MDAL), that completely hides the data backend. This library currently implements a specific logic for MySQL, but its interfaces can be enhanced with other logics.

4.3.4 MOSES Clustered Architecture

In designing the clustered architecture of MOSES we made a tradeoff between flexibility and performance. By flexibility we mean the ability to distribute the MOSES components at the finest level of granularity (i.e., each component on a different machine); however, we have found that having a high degree of flexibility impacts negatively on the overall MOSES performance [28]. Therefore, we have carefully distributed the MOSES components in order to minimize the network overheads for inter-module

communications and storage access. Following this guideline, we have collocated the BPEL Engine and the Adaptation Manager on the same machine; in such a way, for each invoked external service whose binding is executed at runtime by the Adaptation Manager, the BPEL Engine does not need to communicate through the network. In addition, being these two components executed by the same JVM, the Adaptation Manager is called as a Java class rather than as a Web service, with consequent performance speedup.

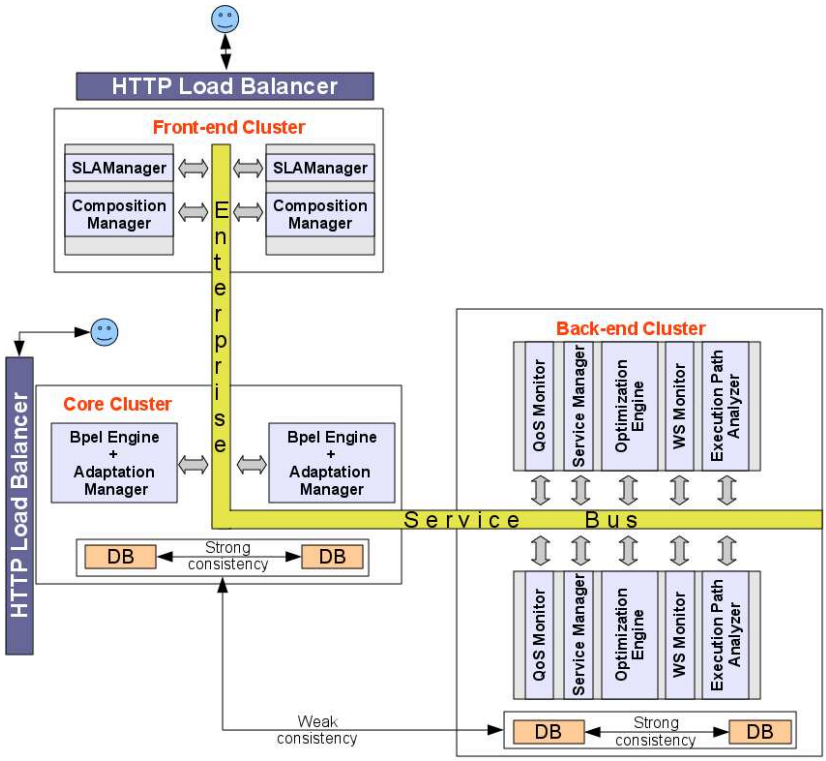


Figure 4.6: MOSES clustered architecture.

Figure 4.6 illustrates the MOSES clustered architecture composed by three clusters, where each one owns two replicas of the components placed in that cluster. The BPEL Engine and the Adaptation Manager constitute the core cluster, while the other two clusters provide additional features that are not mandatory for the basic execution. The front-end cluster provides the broker with the ability to receive new BPEL processes and negotiate SLAs with users. The back-end cluster comprises the components to monitor and analyze the environment and to determine a new adaptation plan. In front of those clusters that are accessed by the composite service users, there is an HTTP load balancer that distributes the requests among the replicas.

As regards the distribution of the storage layer, the core cluster hosts its own high available DB server with strong consistency to execute the DB queries as fastest as possible. The back-end clusters DB is instead synchronized with the core clusters DB using an external weak consistency policy and an internal strong consistency policy. Finally, the front-end cluster does not own a DB at all: we assume that the request rate directed to it is much lower than that directed to the core cluster; therefore, we prefer to pay a penalty for the DB accesses generated by the front-end cluster rather than having on it a new MySQL instance with its own replication strategy and related overhead.

4.3.5 MOSES Overheads

The runtime adaptation management introduces in MOSES different types of overheads, that may affect the response time of the composite service and can be classified according to the MOSES macro-components:

- overhead due to the Plan macro-component (*i.e.*, the Optimization Engine);
- overhead of the Execution macro-component (*i.e.*, the Adaptation Manager) due

to the runtime binding of the task endpoints to concrete implementations;

- overhead due to the Monitor and Analyze macro-components.

For the first type of overhead, we must distinguish between per-request and per-flow optimization strategies. In the first case, indeed, the Optimization Engine calculates a new adaptation policy for each invocation to the application and the optimization overhead must be totally summed to the overall execution time. In the second case, instead, we observe that the Optimization Engine calculates a new adaptation policy asynchronously with respect to the service execution flow, while incoming service requests are served by the Adaptation Manager according to the previously calculated policy. Only when the new adaptation policy is stored in the database, the Adaptation Manager begins to use it. Hence, the Optimization Engine only interferes with those requests that are being served while the new solution of the optimization problem has to be stored. However, the time taken to calculate a new adaptation policy affects the MOSES ability to promptly react to changes in the environmental conditions. The second kind of overhead affects each request to the composite service as many times as the number of `invoke` activities executed in the BPEL process. For every invocation of an abstract task, the Adaptation Manager, which is stateless, retrieves the current adaptation policy kept in the storage layer and, according to it, determines the coordination pattern to be used and the actual operation(s) to implement the abstract task. We will measure in Section 5.3.1 the overhead introduced by the Adaptation Manager to execute the runtime binding.

Finally, for the third kind of overhead, we should distinguish between Monitor and Analyze macro-components impact. We point out that only Monitor affects the overall service time perceived by a user, while Analyze does not affect it, since this

function is executed asynchronously with respect to the business process. The most time consuming and frequent monitoring activity is that performed with respect to the SLA parameters offered by the operations. In this case, the monitoring overhead is about one millisecond for each `invoke` activity, as it only involves inserting the operation response time in a table of the MOSES database: for each operation invocation, MOSES gets the timestamp before and after the invocation itself, and then stores the observed response time, together with a flag reporting whether the operation execution failed. Such values are asynchronously read by the QoS Monitor in the Analyze macro-component, that could run on a different machine with respect to that assigned to the BPEL execution not to interfere with the Execution macro-component. The QoS Monitor is invoked at a fixed, configurable frequency and its task is to analyze stored monitoring data in order to find out whether some SLA has been violated. It performs two steps:

1. for each invoked operation, it computes statistics like average response time and standard deviation;

2. it compares computed statistics with SLA parameters and, in case of violation, it issues a call to the Optimization Engine.

5
Performance Evaluation

Contents

5.1 Testing Environment 121
5.2 Workload Generator 121
5.3 Performance Evaluation 123
5.3.1 Runtime Binding Overhead Analysis 123
5.3.2 Performance of MOSES ESB Clustered 125
5.4 Effectiveness Evaluation 126
5.4.1 Effectiveness of Per-Flow Adaptation Policy 126
5.4.2 Improving Reliability through Web Service Monitoring . . 136
5.4.3 Comparison of Per-Request and Per-Flow Approaches . . . 142
5.4.4 Comparison of Per-Request and Load-Aware Per-Request Approaches 151

In this chapter we will illustrate:

- the performance of MOSES with respect to the execution of a SOA application with static binding and without adaptation features;
- the performance of a SOA application served by MOSES using different adaptation policies.

For the former we will present two sets of experiments. The first aims at evidencing the overhead introduced by the runtime binding, the second at showing the scalability properties of MOSES.

For the latter we will describe four different set of experiments. In the first set of experiments we will prove the effectiveness of the *per-flow* adaptation policy described in Section 3.2.6.2 (see [23] for more details) without any reactive component. That is, we suppose that the concrete services we use in this evaluation behave exactly as they declare into their SLA. We will demonstrate how MOSES is able to serve several flows of requests in compliance with the established SLAs. This set of experiments also includes an experiment aimed at showing the computational cost of the *per-flow* optimization algorithm.

The second set of experiments goes one step further: we relax the assumption of concrete services behaving exactly like declared into their SLAs. We expect therefore that some service could fail during the execution of the experiment. Such a failure could lead in turn to many SOA application execution failures if not correctly addressed. For this reason, we enable in this second set of experiments the reactive component *WS-Monitor* (presented in Section 4.3.1), which is able to detect concrete services failures.

In the third set of experiments we compare the *per-request* and *per-flow* optimization approaches described in Sections 3.2.6.1 (see [12] for more details) and 3.2.6.2, respectively. We will show the scalability limits of the former: since it is completely stateless, it does not consider the clients aggregate request rate. Therefore, the candidate operations could get overloaded and the SLAs established with the clients could not be satisfied. On the contrary, since the *per-flow approach* is stateful, there is no overload of concrete services because MOSES starts dropping new contract requests from clients when the aggregate request rate in no more sustainable.

Finally, in the last set of experiments, we will compare the *per-request* optimization policy with the *load-aware per-request* optimization policy described in Section 3.2.6.3

(see [27] for more details), which keeps track of the load submitted to each concrete service. This feature allows the policy to scale-out the multiple implementations of the same functionality, therefore allowing a load-balancing of the concrete services.

5.1 Testing Environment

For every set of experiments, the testing environment consisted of 3 Intel Xeon quad-core servers (2 Ghz/core) with 8 GB RAM each (nodes 1, 2, 3), and 1 KVM Virtual Machine with 1 CPU and 1 GB RAM (node 4); a Gbit Ethernet connected all the machines. The MOSES prototype has been deployed as follows: node 1 hosted all the components of the Execute subsystem, node 2 the storage layer together with the candidate concrete services; node 3 the components in the Monitor, Analyze and Plan subsystems. Finally, node 4 hosted the workload generator. For every scenario we considered the SOA application defined by the workflow shown in Figure 5.1, composed by 6 stateless tasks, while the number of concrete services implementing such tasks has been varied according to the objective of each experiment. In every experiment, unless otherwise specified, we generated the MOSES load through a load generator, described in the following section.

5.2 Workload Generator

To issue requests to the composite service managed by MOSES and to mimic the behavior of users that establish SLAs before accessing the service, we have developed a workload generator. It is based on an open system model, where users requesting a given service class $k \in K$ offered by MOSES arrive at mean *user inter-arrival rate* Λ_k. Each class k user u is characterized by its SLA parameters and by the *contract duration*

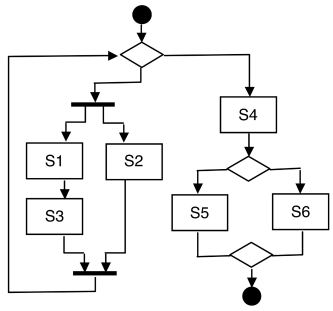


Figure 5.1: Workflow of the composite service managed by MOSES

t_u^k. If we are using the per-flow adaptation policy, each incoming user is subject to an admission control, carried out by the SLA Manager as follows: the user arrival rate λ_u^k is added to the aggregate flow L^k of class k requests currently served by MOSES, and the so obtained new instance of the optimization model is solved by the Optimization Engine. If a solution exists, the user is admitted and starts generating requests to the composite service according to the rate λ_u^k until its contract ends. Otherwise, its SLA request is rejected, because MOSES does not hold sufficient resources to manage it and the already admitted users with their SLAs, and the user terminates. In case we are instead using the per-request or load-aware per request adaptation policies, the previous step is skipped. Differently from traditional Web workload, SOA workload characterization has been not deeply investigated up to now (some preliminary results can be found in [78]). Therefore, in our workload model we assume exponential distributions of parameters Λ_k and $1/t_k$ for the user inter-arrival time and contract duration, respectively. We also assume that the request inter-arrival rate and the operations service time follow a Gaussian distribution, where m_k and σ_k are the parameters of the former, and r_{ij} and $r_{ij}/12$ are the parameters of the latter.

The workload generator has been implemented in C language using the Pthreads library. Multiple independent random number streams have been used for each stochastic component of the workload model.

5.3 Performance Evaluation

5.3.1 Runtime Binding Overhead Analysis

We point out that this kind of overhead is present in every system that provides runtime binding capabilities as MOSES does, irrespectively of the methodology used to determine the adaptation policy.

We have performed a stress test of the MOSES prototype under an open system model, where the requests to the composite service have been generated at an increasing rate through the `httperf` tool [46]. The overall experiment consists of 120 runs, each one lasting 300 seconds during which `httperf` generates requests to the composite service at a constant rate. The adaptation policy is determined at the beginning of each run and is then used for the entire duration of the run without being recalculated, because the goal of this experiment is to measure the additional overhead the runtime binding adds to a plain BPEL engine. The main performance metric we collected for each run is the mean response time, i.e., the time spent on average for the entire request-response cycle.

For increasing values of the request arrival rate to the composite service, Figure 5.2 compares the response time achieved by MOSES, which executes the runtime binding according to the adaptation directives, to that obtained by the standard GlassFish ESB with Sun BPEL Engine, which only provides the composite service execution with a static binding to a given operation. As expected, MOSES is able to sustain lower

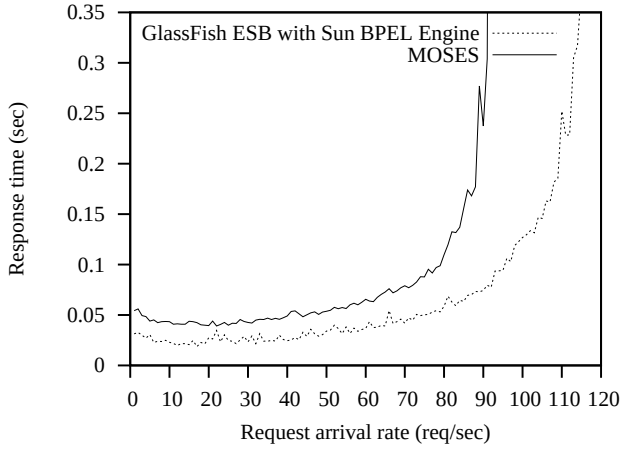


Figure 5.2: MOSES response time.

load levels than GlassFish ESB before reaching the saturation point, because of the overhead introduced by the Adaptation Manager for each abstract task. Until the request arrival rate does not reach the MOSES saturation point (around 80 req/sec), the MOSES response time is on average 74% higher than that provided by GlassFish ESB (the percentage increase ranges from a minimum of 13% to a maximum of 127%). In the experiments presented above, the composite service workflow corresponds to that shown in Figure 5.1. In general, we observe that the runtime binding overhead is related to the size of the managed composite service. In case of static binding, the binding execution complexity depends only on the number of abstract tasks, i.e., $O(m)$. In case of MOSES runtime binding, for each invoked abstract task S_i the Adaptation Manager needs to retrieve from the database the specific records of the table that store the current adaptation policy $\boldsymbol{x}_i^k$. Since B-trees are commonly used in databases,

the time complexity for searching the implementation sets is logarithmic in the number of the table entries. Therefore, the overall execution complexity in MOSES is $O(m \log(m|K| \max_i |\Im_i|))$, where the logarithmic factor is the overhead introduced by the Adaptation Manager.

5.3.2 Performance of MOSES ESB Clustered

The experiments for the clustered version of MOSES ESB have been based on both the open and closed system models. These sets of experiments were executed with the same hardware already used for the non-clustered version, but we slightly changed the component deployment schema. We used 5 machines, where nodes 1 and 2 hosted a GlassFish instance, node 3 the data backend and the concrete services, node 4 the load balancer, and node 5 either The Grinder [44] or httperf. GlassFish allows the system administrator to choose between two load balancers: Sun Java Web Server or Apache Web Server with a load balancing plugin. The first is a closed-source Web server; therefore, we have chosen the latter being open-source. Nevertheless, we were constrained to use a closed-source plugin in order to have an active load-balancing subsystem, which allows to react at the load-balancer level to any failure of the connected GlassFish instances, for example by re-issuing the request to an active instance.

Figure 5.3 shows the throughput improvement achieved by adding a Glass-Fish instance to the MOSES cluster. The load balancer introduces a negligible overhead and the overall performance is incremented by almost a factor of 2. Figure 5.4 compares the clustered version of MOSES ESB with its non-clustered counterpart using the open system model. Similarly to what obtained in the closed-model experiment, we can see that for a low request load, the clustered version is a bit slower than the non-clustered

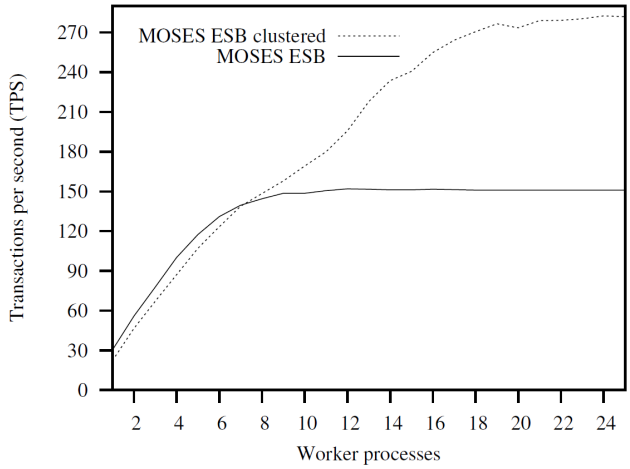


Figure 5.3: Throughput in the closed model.

one because of the load balancer component. However, this gap is rapidly filled starting from the request rate equal to 50. After this point, the clustered version is clearly the winner, achieving a response time that halves that of the non-clustered prototype.

5.4 Effectiveness Evaluation

5.4.1 Effectiveness of Per-Flow Adaptation Policy

In this section we illustrate the result of the adaptation directives issued by MOSES under two different broker goals:

1. the maximization of the average reliability;
2. the minimization of the average cost.

In both the experiments, we also analyze the effectiveness of considering redundancy patterns described in Section 3.2.2 for the task implementation. To this end, we com-

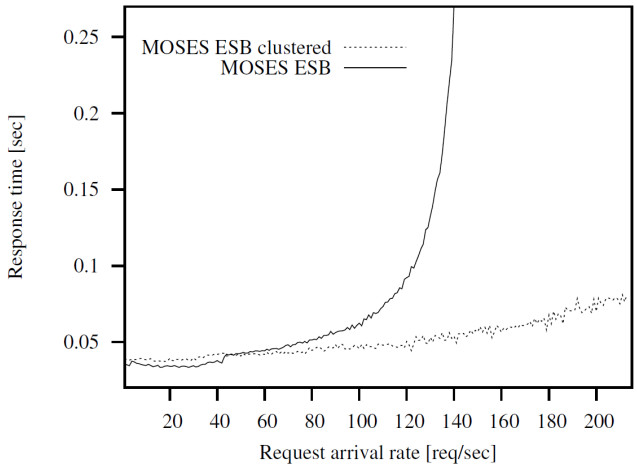


Figure 5.4: Response time in the open model.

pare the performance of MOSES when the redundancy patterns *par_or* and *alt* are enabled with that obtained with only the *single* pattern enabled. The results are summarized in Table 5.3, which shows for each class the measured values of the SLA parameters for the with- and w/o-Redundancy approaches in the two scenarios, reporting the mean values along with the 95% confidence interval.

For the sake of simplicity we assume that two candidate operations (with their respective SLAs) have been identified for each task, except for task S_2 for which four operations have been identified. The respective SLAs differ in terms of cost, reliability, and response time (being the latter measured in seconds). Table 5.1 summarizes the SLA parameters $\langle r_{ij}, c_{ij}, d_{ij} \rangle$ for each operation op_{ij}. They have been chosen so that for task S_i, operation op_{i1} represents the best implementation, which at a higher cost guarantees higher reliability and lower response time with respect to operation op_{ij} for $j \geq 2$, which costs less but has lower reliability and higher response time. For all

Oper.	c_{ij}	d_{ij}	r_{ij}
op_{11}	6	0.995	2
op_{12}	3	0.99	4
op_{21}	4.5	0.99	1
op_{22}	4	0.99	2
op_{23}	2	0.95	4
op_{24}	1	0.95	5
op_{31}	2	0.995	1
op_{32}	1.8	0.995	2
op_{41}	1	0.995	0.5
op_{42}	0.8	0.99	1
op_{51}	2	0.99	2
op_{52}	1.4	0.95	4
op_{61}	0.5	0.99	1.8
op_{62}	0.4	0.95	4

Table 5.1: Operation SLA parameters.

operations, $L_{ij} = 10$ invocations per second.

On the user side, we assume a scenario with four classes of the composite service managed by MOSES. The SLAs negotiated by the users are characterized by a wide range of QoS requirements as listed in Table 5.2, with users in service class 1 having the most stringent requirements, $D^1_{min} = 0.95$ and $R^1_{max} = 7.1$ and users in service class 4 the least stringent requirements $D^4_{min} = 0.85$ and $R^4_{max} = 18.1$. The SLA cost parameters for these classes have been set accordingly, where service class 1 has the highest cost per request, $C^1 = 25$, while service class 4 only $C^4 = 12$. The rightmost column of Table 5.2 reports the values for L_k, that is the aggregate rate of class-k requests to the composite service. The usage profile of the different user service classes is given by the following values for the expected number of service invocations: $V^k_1 = V^k_2 = V^k_3 = 1.5$, $V^k_4 = 1$, $k \in K$; $V^k_5 = 0.7$, $V^k_6 = 0.3$, $k \in \{1, 3, 4\}$; $V^2_5 = V^2_6 = 0.5$. In other words, all classes have the same usage profile except for users in service class 2, who invoke the tasks S_5 and S_6 with different intensity. The values of the parameters that characterize the user workload model are $t_k = 100$ and $(m_k, \sigma_k) = (3, 1)$, $\forall k \in K$. For the experiments presented in the next Section, the

Class k	C^k	$D^k_{\min}$	$R^k_{\max}$	L^k
1	25	0.95	7.1	1.5
2	18	0.9	11.1	1
3	15	0.9	15.1	3
4	12	0.85	18.1	1

Table 5.2: Class SLA parameters.

changes detected by MOSES and that trigger the Optimization Engine include only the arrival/departure of users, that cause a variation of the load and QoS requirements addressed to the composite service.

5.4.1.1 Maximization of the Average Reliability

In the first experiment, the broker goal is to maximize the users' reliability. In this setting, the solution provided by the Optimization Engine is bounded by the maximum cost the broker is willing to pay for each user (which defines its profit margin). Only for the w/o-Redundancy approach, the solution is also bounded by the single operations available to implement the services. Both approaches succeed in respecting the SLA values (see left side of Table 5.3). We observe that with respect to the w/o-Redundancy approach, the with-Redundancy approach allows achieving a higher level of satisfaction of the reliability parameter (the mean values for the four classes range from 0.9983 to 0.9991) at a higher cost, whose mean value is saturated to the maximum agreed in the SLA (see Table 5.2). This is particularly evident for class 1, which requires the most stringent performance requirements at the highest cost (the mean cost ranges from 21.149 for the w/o-Redundancy approach to 25.051 for the with-Redundancy approach, being 25 the cost settled in the SLA). The improvement of the reliability is achieved thanks to the additional patterns *par_or* and *alt* exploited

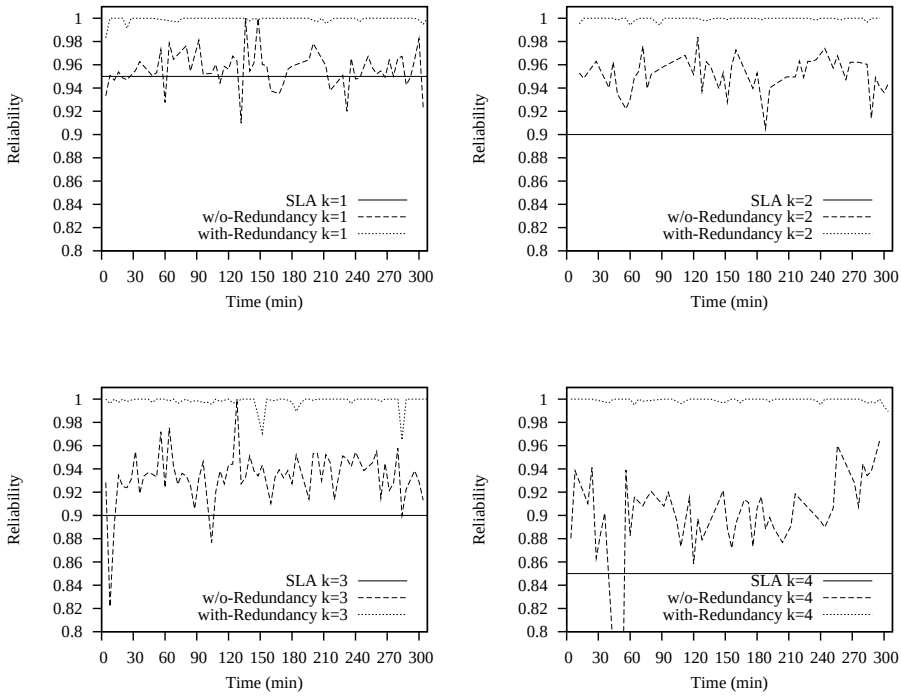
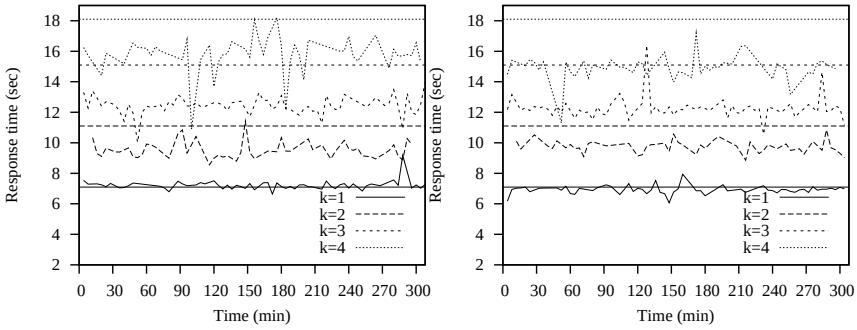


Figure 5.5: ($w_d = 1$): reliability for all classes over time.

by the with-Redundancy approach.

To compare in more detail the w/o- and with-Redundancy approaches with respect to the reliability QoS parameter, Figure 5.5 shows how in the first scenario the reliability of the composite service varies over time for the four classes. The horizontal line is the agreed reliability, as reported in Table 5.2. We observe that the w/o-Redundancy approach leads to some violations of the agreed reliability, while the with-Redundancy approach allows the broker to offer always a reliability much better than that agreed.



(a) w/o-Redundancy.

(b) with-Redundancy.

Figure 5.6: ($w_d = 1$): response time for all classes over time.

The exploitation of the redundancy coordination patterns improves the reliability but it can determine an increase in the response time when the *alt* pattern is selected. Figure 5.6 shows how in the first scenario the response time of the composite service varies over time for the four classes, being the horizontal lines the agreed response times, as reported in Table 5.2. We observe that the with-Redundancy approach leads to a response time that is slightly higher than that achieved by the w/o-Redundancy approach. However, for classes from 2 to 4 the response time is always much lower than that agreed, while for class 1, which requires the most stringent performance requirements, it reaches the maximum agreed in the SLA.

5.4.1.2 Minimization of the Average Cost

In this experiment the broker goal is to minimize the expected cost (which in turn maximizes the broker profit). In this setting, the broker has no incentive to guarantee to the users more than the minimum required. As a result, the solution provided by the Optimization Engine guarantees only the minimum required level of reliability (see

	Scenario 1 (w_d=1) - w/o-Redundancy		
	C^k	D^k	R^k
$k=1$	21.149 ± 0.148	0.955 ± 0.0028	6.934 ± 0.037
$k=2$	18.173 ± 0.155	0.9514 ± 0.0035	9.741 ± 0.075
$k=3$	14.808 ± 0.072	0.9339 ± 0.0024	12.194 ± 0.058
$k=4$	11.744 ± 0.093	0.9017 ± 0.0049	14.936 ± 0.122
	Scenario 1 (w_d=1) - with-Redundancy		
	C^k	D^k	R^k
$k=1$	25.051 ± 0.184	0.9991 ± 0.0004	7.182 ± 0.045
$k=2$	18.427 ± 0.137	0.9991 ± 0.0004	9.509 ± 0.068
$k=3$	14.97 ± 0.074	0.9987 ± 0.0003	12.641 ± 0.064
$k=4$	11.953 ± 0.087	0.9983 ± 0.0004	16.001 ± 0.121
	Scenario 2 (w_c=1) - w/o-Redundancy		
	C^k	D^k	R^k
$k=1$	20.973 ± 0.172	0.9539 ± 0.0033	7.007 ± 0.044
$k=2$	15.866 ± 0.117	0.934 ± 0.0036	10.899 ± 0.079
$k=3$	12.255 ± 0.062	0.9032 ± 0.003	14.491 ± 0.076
$k=4$	10.659 ± 0.09	0.8623 ± 0.0058	17.651 ± 0.135
	Scenario 1 (w_c=1) - with-Redundancy		
	C^k	D^k	R^k
$k=1$	20.843 ± 0.172	0.9555 ± 0.0033	7.135 ± 0.051
$k=2$	15.891 ± 0.141	0.9308 ± 0.0044	11.023 ± 0.1
$k=3$	12.144 ± 0.053	0.9024 ± 0.0026	14.747 ± 0.066
$k=4$	10.426 ± 0.091	0.8625 ± 0.0062	17.76 ± 0.146

Table 5.3: Measured values for SLA parameters (mean and 95% confidence interval).

right side of Table 5.3), with increasing costs for increasing reliability levels.

Let us now consider how in the second experiment the reliability of the composite service varies over time, as shown in Figure 5.7. As expected, we find that the reliability level achieved with the with-Redundancy approach is lower with respect to the first experiment. The motivation is that, when the broker minimizes the cost of the composite service, the solution of the optimization problem exploits less frequently the redundancy coordination patterns *par_or* and *alt* as they may cost more than the *single*

pattern.

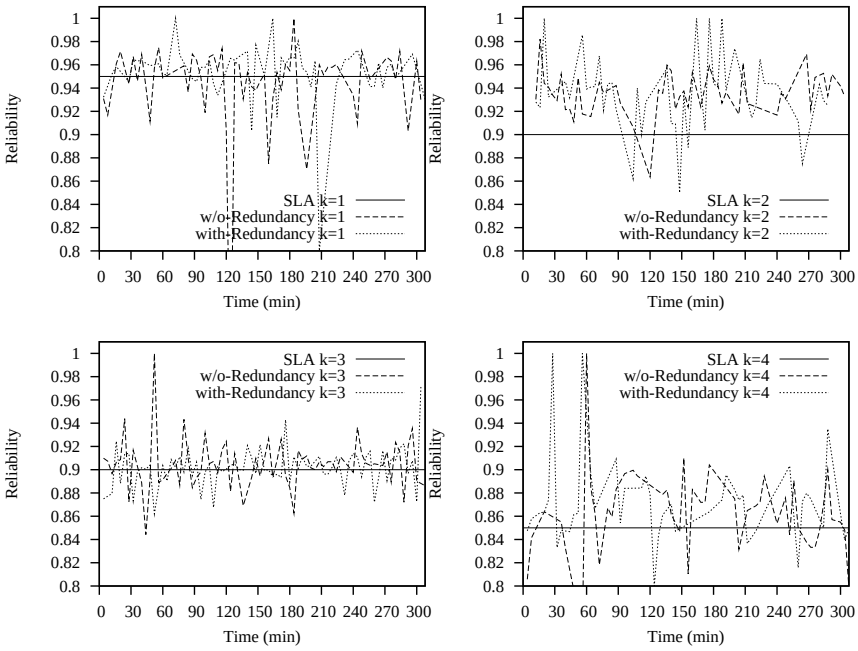


Figure 5.7: ($w_c = 1$): reliability for all classes over time.

5.4.1.3 Adaptation Policy Computational Cost

The per-flow optimization algorithm is implemented in MATLAB®. To assess the algorithm computational cost, we executed the algorithm on 2.00GHz Intel(R) Xeon(R) CPU E5504 quad-core with 8GB RAM on randomly generated problem instances and measured the solution execution time. The results are reported in Figures 5.8-5.9 for different values of number of composite service tasks m, number of service classes $|K|$, number of operations implementing a task n_i, and different maximum degree of

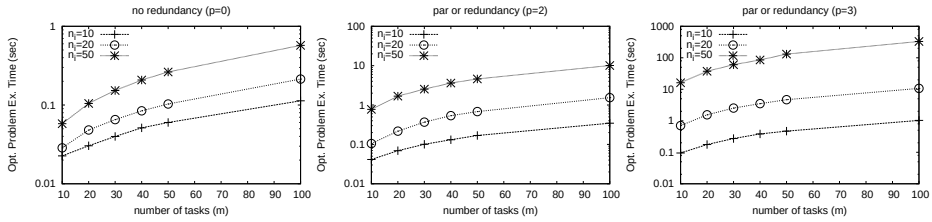


Figure 5.8: Optimization problem execution time for different values of maximal redundancy: (a) no redundancy ($p = 0$); (b) at most two concrete services using the *par_or* pattern ($p = 2$) and (c) at most three concrete services using the *par_or* pattern ($p = 3$).

redundancy a, p. For the sake of simplicity, and without loss of generality, in the following we consider only the *par_or* pattern as redundancy pattern for the analysis of the computational complexity. Similar results apply to the *alt* pattern and to the simultaneous use of both patterns.

In Figure 5.8 we plot the execution time vs the number of service tasks m for different level of *par_or* redundancy: $p = 0$, no redundancy, i.e., service selection only, $p = 2$, at most two concrete services using the *par_or* pattern, and $p = 3$, at most three concrete services using the *par_or* pattern and for different numbers of available operations implementing a given task n_i ($n_i = 10, 20$ and 50). In these set of experiments, we consider only one class of service, *i.e.*, $|K| = 1$. From the plots, we can observe that for fixed p and n_i, the execution time grows almost linearly with the number of tasks m (about one order of magnitude increase of the execution time for one order of magnitude increase in the number of tasks). At closer inspection we verified this holds true for execution times below one second; for larger values the execution time is actually proportional to m^3 which is consistent with the the fact that the problem size

n grows linearly with m (and $|K|$) and the per iteration cost of interior points methods is $O(n^3)$. We will return to this later.

By comparing the different plots we note that, as expected, the execution time is greatly affected by the absence/presence of redundancy patterns and the number of available implementations: without redundancy (Figure 5.8(a)), the execution time is always below 1 second; if we consider redundancy with the *par_or* pattern with at most two services (Figure 5.8(b)), the execution time increases up to few seconds for the larger instances; by increasing the maximum number of redundant operations to three (Figure 5.8(c)), the execution time grows significantly up to 5 minutes for large values of n_i. This behavior can be explained by observing that the use of the redundancy patterns, coupled with a high number of concrete operations, yields a large number of possible implementations and thus a large number of variables since n is proportional to n_i^p: in the range of values considered, while the smallest problem instance has only 100 variables, the largest one grows up to 2,087,500. This has, of course, a significant impact on the problem execution time.

In Figure 5.9 we vary the number of service classes $|K|$ and study the impact of $|K|$ on execution time for different values of n_i and maximal redundancy level p. The number of tasks is again fixed to $m = 50$. Not surprisingly, the same remarks above on the influence of m hold true for the number of service classes: for fixed p and n_i, the execution time grows almost linearly with $|K|$ for smaller instances and proportionally to $|K|^3$ otherwise. We observe that this behaviour is consistent with the $O(n^3)$ iteration cost and $O(n^{\frac{3}{2}} \log \frac{n}{\epsilon})$ worst case iteration complexity of interior points methods. Indeed, in our experiments we observed a relatively low number of iterations for convergence, which grew only slightly from about 10 to 100 (hence much less than the

O($n^{\frac{3}{2}} \log \frac{n}{\epsilon}$), the worst iteration cost for the Mehrotra algorithm) which explains the $O(n^3)$ overall cost.

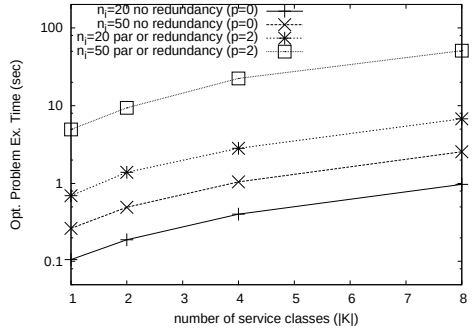


Figure 5.9: Optimization problem execution time as function of the number of service classes.

We remark that since the optimization problem is solved asynchronously with respect to MOSES operations, this large value does not directly impact on the broker responsiveness to user requests; it only affects the time it takes to update the adaptation policy. In other words, it only affects the interval of time during which, while a new solution is being computed, the broker uses the old, sub-optimal policy for the ongoing requests.

5.4.2 Improving Reliability through Web Service Monitoring

The experiments presented in the previous Section proved how MOSES is able to serve a process with QoS constraints when every involved concrete service behaves as declared within its SLA. However, in a real world scenario, we may expect SLA violations and therefore, in order not to propagate the violations to MOSES end-users, we have to detect them and to properly react. In this section we illustrate how MOSES

is able to adapt its behavior with respect to the churn of the services it can use to offer the SOA application. In all the following experiments, the utility function of the service broker is to maximize the reliability of the SOA application according to the per-flow optimization problem. The first experiment simulates an ideal scenario, where the concrete services behave exactly as declared into their SLAs with the service broker. Therefore, it provides a baseline performance result against which we compare the results obtained in the two other experiments. In this first experiment, only the components of the Execute subsystem are involved, because there is no actual need to monitor and/or analyze the environment. Therefore, the same service selection policy holds unchanged for the whole experiment.

In the second experiment we introduce some churn with respect to the baseline experiment, by letting concrete services gracefully fail and recover over time. The failure/recovery model follows a two-state discrete Markov chain, with stationary probability distribution prunning, pfailed=0.95, 0.05, in which state changes can occur on average every 60 seconds. The gracefulness is given by the fact that the concrete services notify their state to MOSES, therefore allowing it to compute a new service selection policy including (excluding) the restored (failed) concrete services. This second experiment employs the components of the Plan and Execute phases of the MAPE-K loop. In particular, whenever a concrete service fails or recovers, the Optimization Engine solves a new instance of the service selection optimization problem.

In the third experiment we assume a real world scenario, where concrete services do not notify their clients (i.e., MOSES) of a failure, but we disable the Monitor phase of the control loop. From the MAPE-K point of view, we can consider that the components in the Plan and Execute are enabled, although the Plan phase is never executed

because it is not triggered by the Analyze step. In other words, as in the first experiment, the same service selection policy holds for the whole experiment.

Finally, in the fourth experiment, we prove the effectiveness of the MAPE-K loop by activating the monitoring of the candidate concrete services performed by the WS Monitor component. The latter is configured to probe all the known concrete services every 5 seconds to find out which services are currently available. Whenever the WS Monitor finds that some service changed its state (going from running to failed or vice-versa), it sends a trigger to the Optimization Engine, which in turn computes the new service selection policy that will be applied by the Execute subsystem. In each set, every experiment lasted 30 minutes and has been repeated twice, using a client request rate equal to 5 and 10 requests/seconds (in the following, referred to as low and high request rates) to show the behavioral differences that arise when MOSES is subject to different loads.

We assume that 10 candidate operations (with their respective SLAs) have been identified for abstract tasks S1 and S3, while 8 candidate operations have been identified for any other task. Their respective SLA parameters, shown in Table 5.4, differ in terms of cost c_{ij}, reliability d_{ij}, and response time r_{ij} (in sec). We also suppose that MOSES offers to its clients the SLA R_{max}, D_{min}, C_{max}=7 sec, 0.95, 15. For simplicity, we consider only a single service class. The usage profile of this service classes is given by the following values for the expected number of service invocations: V1 = V2 = V3 = 1.5, V4 = 1, V5 = V6 = 0.5.

5.4.2.1 Baseline Scenario

We first present the results of the baseline scenario. The Baseline curves in Figures 5.10 and 5.11 show how the reliability of the SOA application varies over time, when the

Oper.	c_{ij}	d_{ij}	r_{ij}
$op_{1[1,6]}$	6	0.995	2
$op_{1[2,7]}$	6	0.99	1.8
$op_{1[3,8]}$	6.5	0.99	2
$op_{1[4,9]}$	4.5	0.995	3
$op_{1[5,10]}$	3	0.99	4
$op_{2[1,5]}$	2	0.995	1
$op_{2[2,6]}$	1.8	0.995	2
$op_{2[3,7]}$	1.8	0.99	1.8
$op_{2[4,8]}$	1	0.99	3
$op_{3[1,6]}$	5	0.995	1
$op_{3[2,7]}$	4.5	0.99	1
$op_{3[3,8]}$	4	0.99	2
$op_{3[4,9]}$	2	0.95	4
$op_{3[5,10]}$	1	0.95	5

Oper.	c_{ij}	d_{ij}	r_{ij}
$op_{4[1,5]}$	1	0.995	0.5
$op_{4[2,6]}$	0.8	0.99	0.5
$op_{4[3,7]}$	0.8	0.995	1
$op_{4[4,8]}$	0.6	0.95	1
$op_{5[1,5]}$	3	0.995	1
$op_{5[2,6]}$	2	0.99	2
$op_{5[3,7]}$	1.5	0.99	3
$op_{5[4,8]}$	1	0.95	4
$op_{6[1,5]}$	1	0.99	1.8
$op_{6[2,6]}$	0.8	0.995	2
$op_{6[3,7]}$	0.6	0.99	3
$op_{6[4,8]}$	0.4	0.95	4

Table 5.4: Operation SLA parameters.

	SLA	Average reliability	95% confidence interval
Low request rate	0.95	0.9664	0.0074
High request rate	0.95	0.9646	0.0054

Table 5.5: Average reliability and 95% confidence interval for the baseline experiment

QoS attribute is measured at the client-side by aggregating the values every 20 seconds.

The horizontal lines represent the SLA stipulated with the clients and the average reliability perceived by the clients over all the experiment duration. We can observe that the reliability fluctuates over time; most of the time it stays well above the SLA value, but occasionally it attains lower values. Nevertheless, as also shown in Table 5.5, where we report the average reliability of the baseline experiment along with the 95% confidence interval, MOSES is able to fulfill the reliability level agreed in the SLA.

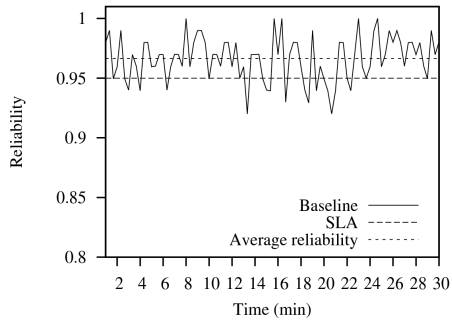


Figure 5.10: Baseline reliability over time under low request rate

	SLA	Average reliability	95% confidence interval
Low request rate	0.95	0.9692	0.0071
High request rate	0.95	0.9659	0.0053

Table 5.6: Average reliability and 95% confidence interval for experiment with graceful failures

5.4.2.2 Graceful Failure and Join of Concrete Services

In the second set of experiments we let the service providers gracefully fail, thus simulating, for instance, service programmed downtimes. The results in Figures 5.12 and 5.13 show how the reliability of the SOA application fluctuates over time; however, the average reliability is well above the agreed SLA. The experimental values in Table 5.6 show that the average reliability, as well as the 95% confidence interval under the second scenario are perfectly comparable to those of the baseline experiment. Therefore, we can conclude that graceful leaves and joins do not affect the reliability performance since MOSES is able to adapt to the changed environment by re-computing the service selection policy.

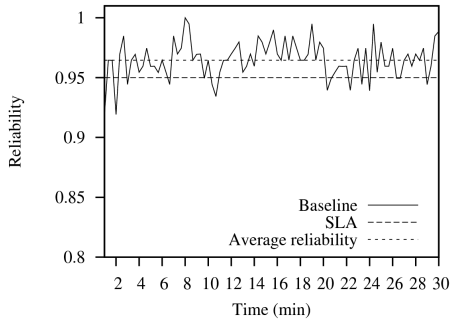


Figure 5.11: Baseline reliability over time under high request rate

5.4.2.3 Undetected Failure/Join of Web Services

Figures 5.14 and 5.15 show how the reliability of the SOA application varies over time when the concrete service providers exhibit the same churn rate of the second experiment, but without signaling their state to MOSES. The reliability levels fall down and the SLA stipulated by MOSES with its clients is no longer fulfilled. This experiment demonstrates that, if there are changes in the execution environment and no adaptation actions are taken to address these changes, the system is not able to satisfy the required QoS. It also points out that reliability levels are higher when the request rate is higher. The motivation is due to the fact that the service selection policy binds each abstract task to a small subset of concrete services when the incoming request rate is low. On the other hand, with a higher request rate, the request load on any abstract task is balanced over a larger set of concrete services, depending on their capacity. Since we set the capacity of every concrete service to 10 req/sec, it is likely to have a single concrete service selected for any abstract task when the incoming request rate is equal to 5 req/sec, while it is likely to have two or more concrete services selected for any

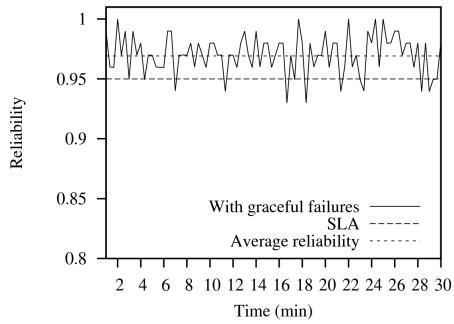


Figure 5.12: Reliability over time when services are subject to graceful failures under low request rate

abstract task when the incoming request rate is 10 req/sec.

5.4.2.4 Detection of Web Service Failures to Improve Reliability

The purpose of the last experiment is to show the improvement achieved thanks to the WS Monitor component. Figures 5.16 and 5.17 show how the reliability of the SOA application varies over time when the service providers exhibit the same churn rate of the third experiment without signaling their state to MOSES, but now with the WS Monitor enabled on MOSES. As shown in Table 5.7, MOSES does not succeed in fulfilling the SLA stipulated with its clients, but the provided reliability has a significant improvement with respect to the results shown in Figures 5.14 and 5.15, when the WS Monitor was disabled.

5.4.3 Comparison of Per-Request and Per-Flow Approaches

In the previous Sections we illustrated several experiments aimed at showing the effectiveness of the per-flow optimization policy. However, thanks to its modular architec-

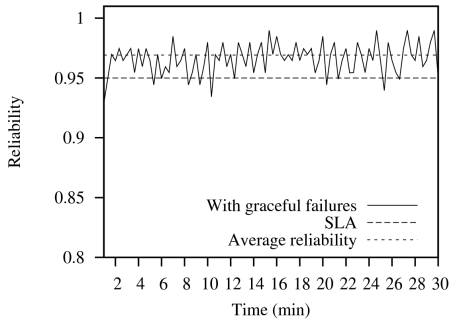


Figure 5.13: Reliability over time when services are subject to graceful failures under high request rate

	SLA	Average reliability	95% confidence interval
Low request rate without WS Monitor	0.95	0.7151	0.0187
Low request rate with WS Monitor	0.95	0.9101	0.0118
High request rate without WS Monitor	0.95	0.7798	0.0122
High request rate with WS Monitor	0.95	0.8974	0.0089

Table 5.7: Comparison of the average reliability and 95% confidence interval for the experiments with and without the WS Monitor

ture, MOSES can implement different optimization strategies. In this section we will show the performance of the per-request optimization algorithm illustrated in Section 3.2.6.1, in comparison with the per-flow one described in Section 3.2.6.2. What we will point out are the scalability limits of the per-request approach, from two different point of views: (i) the computational cost to calculate an optimal solution and (ii) the effectiveness of the optimal solution at runtime with heavy load. We will start by comparing the execution times of the *per-flow* optimization algorithm with the ones provided by [6] on the *per-request*. Then we will compare the response time of a SOA application served by MOSES using both the *per-request* and *per-flow* optimization

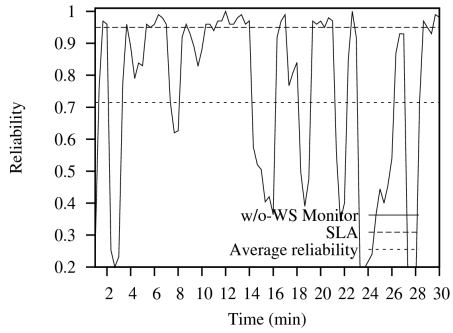


Figure 5.14: Reliability over time when services are subject to failures, without WS Monitor under low request rate

strategies.

We assume that 4 candidate operations (with their respective SLAs) have been identified for each task, except for tasks S_1 and S_3 for which 5 implementations have been identified. The respective SLA parameters, shown in Table 5.8(left), differ in terms of cost c_{ij}, reliability d_{ij}, and response time r_{ij} (in sec). The candidate operations are simple stubs; however, their non-functional behavior conforms to the guaranteed levels expressed in their SLA. The perceived response time is obtained by modeling each service as a M/G/1/PS queue implemented inside the Web service deployed in the Apache Tomcat container. For all concrete services the load threshold L_{ij} is equal to 10 req/sec and the response time knee is beyond it.

On the user side, we assume a scenario with four classes of the composite service managed by MOSES. The SLAs negotiated by the users are characterized by a range of QoS requirements as listed in Table 5.8(right), with users in class 1 having the most stringent performance requirements (being willing to pay the highest cost) and users in class 4 the least stringent ones (being willing to save money). The usage profile of the

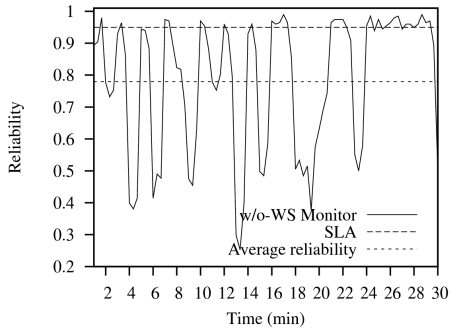


Figure 5.15: Reliability over time when services are subject to failures, without WS Monitor under high request rate

service classes is given by the following values for the maximum number of service invocations: $V^k_{\alpha,1} = V^k_{\alpha,2} = V^k_{\alpha,3} = 3$, $V^k_{\alpha,4} = 1$, $k \in K$; $V^k_{\alpha,5} = 0.7$, $V^k_{\alpha,6} = 0.3$, $k \in \{1, 3, 4\}$; $V^2_{\alpha,5} = V^2_{\alpha,6} = 0.5$, being $\alpha = 0.96$ the 96-percentile of the distribution of reiterating the loop.

5.4.3.1 Per-Request and Per-Flow Optimization Time Comparison

We compare the per-flow approach with the per-request approaches presented in [6, 12] which are among the most representative contributions in the literature. The data, also shown in Table 5.9, are taken from [11] and have been obtained on an equivalent machine, according to CINT and SpecCPU2006 benchmarks (lines $(m, n_i) = (100, 10) - (10000, 10)$ report values from [12], while the rest report values from [6]). The results show that MOSES per-flow adaptation policy has execution times comparable to those in [12] and about one order of magnitude larger than those in [6]. We can argue that in a lightly loaded and/or small scale system, it may be effective to address the adaptation to each single request, independently of other concurrent requests,

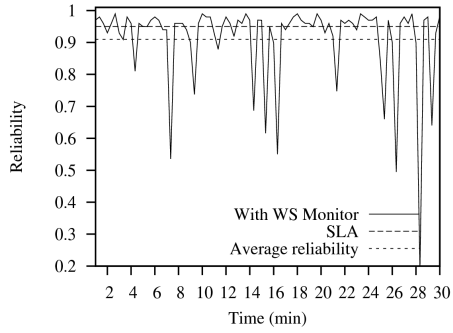


Figure 5.16: Reliability over time when services are subject to failures, with WS Monitor under low request rate

to customize the system with respect to that single request. However, in a large scale system subject to a quite sustained flow of requests, performing a *per-request* rather than a *per-flow* adaptation could cause an excessive computational load. In this kind of scenarios, per-flow adaptation is likely to be more effective, even if it loses the potentially finer customization features of per-request adaptation. Moreover, per-request adaptation could also incur in stability and management problems, since the "local" adaptation actions could conflict with adaptation actions independently determined for other concurrent requests.

5.4.3.2 Per-Request and Per-Flow Execution Time Comparison

To compare the per-flow and per-request service selection approaches, we consider two different workload scenarios.

In the *first scenario*, we consider each service class per time (*i.e.*, in a specific experiment the requests pertain only to one of the service classes in Table 5.8(right)) and we stress the MOSES system by progressively increasing the request rate. To this

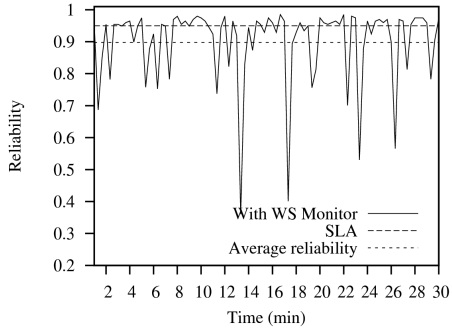


Figure 5.17: Reliability over time when services are subject to failures, with WS Monitor under high request rate

end, we set for all the contracts a fixed duration equal to 100 sec and L_u^k=1 req/sec, while the contract interarrival rate ranges from 0.01 to 0.3 contracts/sec for each step of the overall experiment: this setting corresponds to an overall request arrival rate L^k from 1 to 30 req/sec. Each single step (corresponding to a given request rate) lasts 15 minutes. At each step, to avoid overwhelming a just started GlassFish instance, which has a significant setup time, the workload generator does not immediately issue requests at the required request rate but within a ramp (set to 100 sec), during which the request rate is linearly incremented until it reaches the desired value.

We focus our analysis on the most sensitive SLA parameter to the workload increase, *i.e.*, the response time, obtained by the requests of class 1, which has the most stringent SLA requirements. Figure 5.18a shows the response time of the composite service achieved by the two alternative approaches for an increasing request rate and with the MOSES monitoring modules disabled (except the SLA Manager). We observe that while the per-flow response time remains well below the agreed SLA value (equal to 14 sec), for the per-request approach (denoted by perReq_w/o-QoSM) the response

op	r_{ij}	d_{ij}	c_{ij}
op_{11}	2	0.995	6
op_{12}	1.8	0.99	6
op_{13}	2	0.99	5.5
op_{14}	3	0.995	4.5
op_{15}	4	0.99	3
op_{21}	1	0.995	2
op_{22}	2	0.995	1.8
op_{23}	1.8	0.99	1.8
op_{24}	3	0.99	1
op_{31}	1	0.995	5
op_{32}	1	0.99	4.5
op_{33}	2	0.99	4
op_{34}	4	0.95	2
op_{35}	5	0.95	1

op	r_{ij}	d_{ij}	c_{ij}
op_{41}	0.5	0.995	1
op_{42}	0.5	0.99	0.8
op_{43}	1	0.995	0.8
op_{44}	1	0.95	0.6
op_{51}	1	0.995	3
op_{52}	2	0.99	2
op_{53}	3	0.99	1.5
op_{54}	4	0.95	1
op_{61}	1.8	0.99	1
op_{62}	2	0.995	0.8
op_{63}	3	0.99	0.6
op_{64}	4	0.95	0.4

Class k	$R^k_{\max}$	$D^k_{\min}$	$C^k_{\max}$
1	14	0.9	39
2	17	0.88	35
3	19	0.86	32
4	22	0.84	29

Table 5.8: SLA parameters for candidate operations (top) and service classes (bottom)

time increases exponentially approximately at the candidate operations' load threshold (set to 10 req/sec). In a lightly loaded system, the per-request approach is effective to address the adaptation to each single request. However, when the workload increases, it incurs in stability and management problems, since it takes adaptation actions just for a single request, independently of the other concurrent requests. Therefore, the candidate operations identified as the best ones by the per-request deterministic policy are overwhelmed by the requests. On the other hand, the probabilistic per-flow policy chooses the best implementations only until their load threshold is not exceeded; at that

Table 5.9: Performance comparison with the per-flow approach of [11] and per-request approaches of [6, 12] (time measured in seconds).

m	n_i	MOSES	per-flow [11]	per-request [6, 12]
100	10	0.11	8.10	0.17
100	20	0.21	9.54	0.63
100	25	0.27	9.98	0.58
100	50	0.58	14.30	0.29
1000	10	1.40	19.60	2.10
1000	20	3.03	144.30	5.38
1000	25	4.07	149.60	4.54
1000	50	8.64	451.30	19.88
5000	10	11.20	444.90	4.54
5000	20	24.55	1000.05	35.06
10000	10	15.64	970.15	113.92
10	100	0.13	7.90	0.027
10	200	0.25	9.61	0.037
10	300	0.44	9.83	0.053
10	400	0.62	10.80	0.043
10	500	0.83	13.98	0.067
10	600	0.92	15.00	0.121
10	700	1.10	17.50	0.097
10	800	1.45	17.60	0.0186
10	900	1.68	19.80	0.112
10	1000	1.87	20.50	0.170
20	500	1.78	19.30	0.189
40	500	4.47	141.40	0.432
50	500	7.54	147.30	0.560
100	500	19.22	448.70	1.518

point, it distributes the requests among a subset of (possibly all) the available concrete services. This behavior is evident in Figure 5.18a, where the response time increases from around 6 to 7 sec at the candidate operations' load threshold. The stable behavior of the per-flow approach is counterbalanced by an amount of dropped SLA contracts; the rejection percentage ranges from 7% (for 12 req/sec) to 59% (for 30 req/sec).

To improve the performance of the per-request approach, we activate the QoS Monitor, so that after a SLA violation the agreed values of the candidate operations' parameters are updated in the system model with the new measured values and the triggered Optimization Engine calculates a new solution of the *per-request* problem. The SLA violation is detected when the data monitored during one time window exceed by 20% the SLA agreed by MOSES with the service providers. We can see in Figure 5.18b that the monitoring activity and the subsequent reaction improve the per-request behavior: when the best implementation for a given task becomes overloaded, the requests are shifted towards another concrete service determined by the new adaptation policy. However, the improvement is achieved at a cost of having a very reactive system, characterized by a quite frequent monitoring activity, because the monitored data are analyzed either to 2 or even 0.7 sec, denoted by perReq_withQoSM_2s and perReq_withQoSM_0.7s in Figure 5.18b.

Let us now consider how in the first scenario the SLA is satisfied: Figure 5.18c shows the percentage of violations for the response time agreed with the users. While under the per-flow approach only few requests suffer from a SLA violation, the percentage dramatically increases for the per-request service selection, even with a frequent monitoring activity.

In the *second scenario* we consider a mixed workload in which MOSES offers si-

multaneously the composite service to all the service classes in Table 5.8(right). We assume exponential distributions of parameters λ_k and $1/t_k$ for the contract inter-arrival time and duration and a Gaussian distribution of parameters (μ_k, σ_k) for the request inter-arrival L_u^k. Each user u generates its requests to the composite service according to an exponential distribution with parameter L_u^k. The values of the workload model parameters are $d_k = 100$ and $(\mu_k, \sigma_k) = (3, 1)\ \forall k$; λ_k, t_k, and μ_k values have been set so that for Little's formula $L_k = \lambda_k \mu_k t_k$ and therefore on average $L = (L^k) = (1.5, 1, 3, 1)$. We analyze how the response time of the composite service varies over time for the most demanding class 1, as shown in Figures 5.19a-5.19c (the horizontal line is the agreed response time, as reported in Table 5.8). Although in the second scenario the system is only subject to a moderate workload intensity (the average overall request rate is 6.37 req/sec, being 20.4 req/sec the peak and 4.29 req/sec the standard deviation), we find that the response time level achieved by the per-flow approach has a much more stable trend and does not suffer from the SLA violations of the per-request service selection. The percentage of dropped contracts by the per-flow approach is 12%.

5.4.4 Comparison of Per-Request and Load-Aware Per-Request Approaches

In the previous Section we proved the scalability limits of the per-request optimization approach. These limits are overcome by the per-flow approach which cannot, however, customize every single client request as the per-request optimization approach does. In this section, we present the experimental analysis we have conducted using the MOSES prototype to demonstrate:

- the effectiveness of the proposed load-aware per-request approach with respect

to the traditional per-request approach proposed by Ardagna and Pernici in [12];

- the effectiveness of the adaptive Cusum algorithm (see Section 3.4 for more details), and in turn of the whole theoretical framework for a self-adaptive SOS.

In this set of experiments, we assume that 4 candidate operations (with their respective SLAs) have been identified for each task, except for tasks S_1 and S_3 for which 5 implementations have been identified. The respective SLA parameters, shown in Table 5.10, differ in terms of cost c_{ij}, reliability d_{ij}, and response time r_{ij} (measured in sec). In the experiments, we used this *baseline* set composed of 26 concrete services, as well as an enlarged set of concrete services where we doubled the baseline set (in the following, we refer to the latter as *2x baseline*).

op	r_{ij}	d_{ij}	c_{ij}
op_{11}	2	0.995	6
op_{12}	1.8	0.99	6
op_{13}	2	0.99	5.5
op_{14}	3	0.995	4.5
op_{15}	4	0.99	3
op_{21}	1	0.995	2
op_{22}	2	0.995	1.8
op_{23}	1.8	0.99	1.8
op_{24}	3	0.99	1
op_{31}	1	0.995	5
op_{32}	1	0.99	4.5
op_{33}	2	0.99	4
op_{34}	4	0.95	2
op_{35}	5	0.95	1

op	r_{ij}	d_{ij}	c_{ij}
op_{41}	0.5	0.995	1
op_{42}	0.5	0.99	0.8
op_{43}	1	0.995	0.8
op_{44}	1	0.95	0.6
op_{51}	1	0.995	3
op_{52}	2	0.99	2
op_{53}	3	0.99	1.5
op_{54}	4	0.95	1
op_{61}	1.8	0.99	1
op_{62}	2	0.995	0.8
op_{63}	3	0.99	0.6
op_{64}	4	0.95	0.4

Table 5.10: SLA parameters for candidate operations

The concrete services are simple stubs, without internal logic; however, their non-functional behavior conforms to the guaranteed levels expressed in their SLA. The per-

Service class k	$R^k_{\max}$	$R^k_{\min}$	$C^k_{\max}$
1	16	0.88	55
2	18	0.85	50
3	20	0.82	45
4	22	0.79	40

Table 5.11: SLA parameters for service classes

ceived response time is obtained by modeling each concrete service as a $M/D/m/PS$ queue implemented inside the Web service deployed in a Tomcat container. The $M/D/m/PS$ model is parameterized in such a way to have an average CPU usage between 65% and 70% when the request rate is equal to 10 req/sec. Table 5.11 shows the SLAs offered by MOSES to the composite service users according to different service classes.

We conducted five experiments to evaluate the service selection policies under different scenarios and loads as well as the monitoring and analysis components. The first experiment was performed to point out the scalability problems of the traditional per-request approach; the second was carried out to compare the performance of the traditional per-request policy versus the load-aware one; the third was performed to analyze the scalability of the load-aware per-request policy. In the first three experiments, each test is composed by several runs lasting 15 minutes each, during which the workload generator generates requests corresponding to the service class 2 (see the k=2 row in Table 5.11 for its SLA) at a constant rate. The request rate is then increased run by run until the system keeps stable. The fourth experiment was performed in order to prove the effectiveness of our load-aware policy. Specifically, for every service class we generated a constant request rate for the first half of the experiment, then increasing the request rate only for class 2 in the second half on the experiment. Finally, the

last experiment was performed to demonstrate the effectiveness of the adaptive Cusum algorithm, when some services do not behave according to the stipulated SLAs. As in the first three experiments, each test is composed by several runs lasting 15 minutes each. The request rate submitted to MOSES is the same in each test, but an external load, increased run by run, is submitted to op_{11}.

The main performance metric we measured is the response time of the composite service. We also measured the CPU utilization of the candidate operations to analyze the different effects of the request load distribution among the candidate operations achieved by the traditional and the load-aware per-request policies.

5.4.4.1 Per-Request Approach

In this first experiment, we ran three load tests on MOSES using the traditional per-request policy in [12]. In the first test, we used the baseline set of concrete services without instrumenting any of the MOSES modules in the Monitor and Analyze macro-components. In the second test, we used the previous configuration, but we exploited the 2x baseline set of candidate operations. Finally, in the last test we used the 2x baseline set of candidate operations and we added the support of the QoS Monitor, in order to detect SLA violations of the response time of the concrete services and, in positive case, to determine a new service selection policy that exploits different candidate operations.

Figure 5.20 shows the average response time perceived by the users of the composite service for different request rates submitted to MOSES. We observe that for all the three tests the response time is nearly constant until the request rate reaches 7 req/sec. From this point on, the response time of both the tests without QoS Monitor (*[12], no QoS, baseline* and *[12], no QoS, 2x baseline* curves), regardless of the used candidate

operations set, rapidly grows because the per-request service selection does not exploit the presence of different service implementations, always using the same service identified as the best one. In the test with the QoS Monitor enabled (*[12], QoS, 2x baseline* curve), the response times of the candidate operations are collected, their average calculated every 2 sec. and analyzed; if the QoS Monitor finds out that the currently used operations do not have an adequate performance (*i.e.,* their are violating the response time contractualized in the SLA), it triggers the Optimization Engine to compute a new optimal policy $\boldsymbol{x}$, using the actual response times of the candidate operations instead of those declared into the SLAs. As a result, the currently used overloaded operations will not be used in the near future, but they will be candidate for re-usage when the new selected operations will in their turn become overloaded. However, the introduction of the QoS Monitor provides only a modest performance improvement; even if the QoS Monitor invocation frequency is relatively high (every 2 sec.), the reaction is not quick enough to address higher request rates. We can conclude that the traditional per-request approach is not able to scale out the available services implementations, and thus it is unable to sustain higher request rates than those sustainable by the bottleneck candidate operations.

5.4.4.2 Comparison between Per-Request and Load-Aware Per-Request Approaches

The second experiment compares the traditional per-request and the load-aware per-request selection policies. This experiment uses the baseline set of candidate operations and does not involve any Monitor or Analyze MOSES component. Figure 5.21 compares the average response time according to the request rate submitted to MOSES when using the two different policies. We observe that the response times achieved by the two policies perfectly overlap until the request rate reaches the saturation point of

the traditional per-request policy. From this point on, the former (*[12], no QoS, baseline* curve) is not able to exploit the available implementations, while the load-aware policy (*no QoS, baseline* curve) performs better, scaling out the available candidate operations. Therefore, the load-aware approach is able to sustain higher request rates than the traditional per-request, given that there are available candidate operations to be exploited.

To show the load balancing effectiveness, we monitored the CPU usage of the candidate operations during the experiments. Since every concrete service is implemented as a $M/D/m/PS$ queue, the CPU usage has been computed with the formula $\frac{\lambda_{ij} T_{ij}}{nCPU_{ij}}$, where λ_{ij} is the request rate directed to the j-th implementation of S_i (that is, op_{ij}), T_{ij} its service time, and $nCPU_{ij}$ the number of CPUs available to that service implementation.

Figure 5.22 shows the CPU usage of op_{13}, which is the single operation used by the traditional per-request optimization approach to implement S_1. We can see that the load increases almost linearly until it reaches the CPU usage equal to 85%; at that value, the system becomes unstable (see Figure 5.21).

Figure 5.23 shows the CPU usage of the candidate operations used by the load-aware policy to implement the abstract task S_1. Differently from the traditional strategy, with the load-aware policy multiple operations can be used to implement the same task. In particular, when the request rate is low (from 1 req/sec to 6 req/sec), there is no need to use multiple operations (we recall that each candidate operations is modeled so to have an average CPU usage between 65% and 70% when the request rate is equal to 10 req/sec). Therefore, for the low request rate only op_{13} is used, like in the traditional per-request policy. When the request rate increases from 7 req/sec to 9

req/sec, the operations op_{13} and op_{11} are both used. From 10 req/sec on, op_{15} is also used to implement S_1, therefore the load is balanced across three operations. We observe that the cumulative load does not increase monotonically, because the candidate operations model different underlying hardware: op_{11} and op_{15} have 29 CPUs each, while op_{13} has 25 CPUs. Therefore, when more load is directed to an operation with a larger capacity, the overall load decreases.

5.4.4.3 Scalability of Load-Aware Per-Request Approach

We carried out the third experiment to show the scalability capabilities of the load-aware per-request policy. In this experiment we used both the baseline and the 2x baseline sets of concrete services, without deploying the QoS Monitor module. Figure 5.24 shows the scaling capabilities of the load-aware per-request service selection: until one operation at a time can sustain the load (*i.e.*, around 7 req/sec), it does not matter to have a larger number of available implementations; therefore, the results with the baseline set of concrete implementations resemble those with the 2x baseline set. However, at higher request rates, the availability of a larger set of candidate services provides better response times and allows to manage the request rates without incurring in overloading, because the load can be better shared among the available implementations.

5.4.4.4 Effectiveness of Load-Aware Per-Request Approach

We conducted the fourth experiment to study the effectiveness of the proposed load-aware per-request approach. We simulated several concurrent users characterized by different service classes. The goal is to prove the effectiveness of the MOSES adaptation under the load-aware per-request policy despite variations in the submitted work-

Service class	Light load	Heavy load
1	5.514 sec	6.254 sec
2	5.485 sec	6.350 sec
3	5.509 sec	6.357 sec
4	5.794 sec	8.112 sec

Table 5.12: Average response times of the load-aware per-request policy for all service classes under light and heavy loads

load. To this end, each service class submits requests at a constant rate equal to 1 req/sec, except class 2 for which we increased the request rate from 1 to 10 req/sec in the second half of the test. Therefore, in the first half of the experiment, the aggregate workload is equal to 4 req/sec, which can be also easily managed by the traditional per-request policy; on the other hand, in the second half of the experiment we submitted to the SOS an aggregated workload equal to 13 req/sec, which cannot be sustained by the traditional per-request policy (see the candidate operations model described in Section 5.4.4). The overall experiment lasted 1 hour.

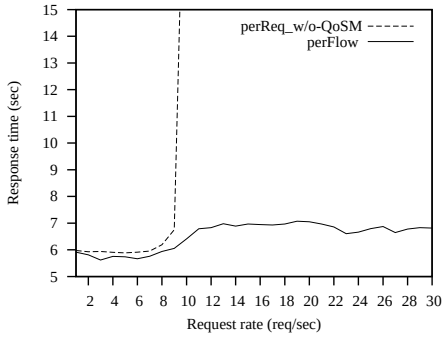
Figures 5.25a-5.25d show that the perceived response times are far below the response times agreed in the SLAs and represented by the horizontal lines; this can be explained by observing that the average behavior is very different from the worst case considered in the formulation of the optimization policy. This latter issue could be addressed by considering SLAs where the response time constraint is specified in terms of bounds on the percentile.

Table 5.12 shows the average response times perceived by the users when issuing requests either to a lightly loaded or to an heavy loaded system according to the service class. When the system is subject to a light load, there are not appreciable differences among the service classes. On the other hand, when the load increases, the average

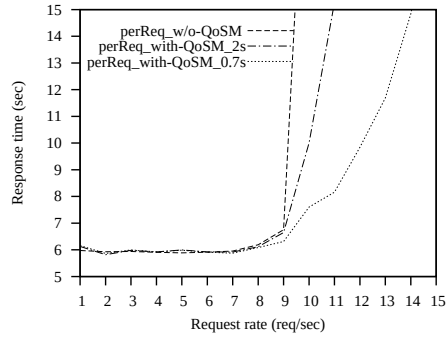
response time perceived by class 4 (which has the least stringent SLA) suffers more the load increase. The motivation is that class 4 requests can only exploit a limited number of candidate operations, because of the lowest maximum cost in the SLA (see Table 5.11); therefore, to satisfy the cost constraint they cannot be distributed among all the available candidate operations.

5.4.4.5 Effectiveness of the Adaptive Cusum Algorithm

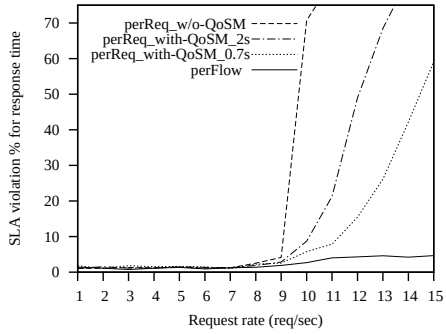
Finally, in the fifth and last experiment we demonstrate the effectiveness of the adaptive Cusum algorithm. In the first test, we used the baseline set of concrete services without instrumenting any of the MOSES modules in the Monitor and Analyze macro-components. Conversely, in the second test we added the support of the QoS Monitor, in order to detect SLA violations of the response time of the candidate operations and, in positive case, to determine a new service selection policy that exploits different concrete services implementations. Both the test were conducted submitting requests corresponding to the service class 2 at a constant rate equal to 4 req/sec. We also submitted an external load to the operation op_{11} in order to overload it. The external load has been incremented by one unit every 15 minutes, starting from 1 req/sec. Figure 5.26 shows the result of the first test without the QoS Monitor. We can see that the response time of the composite service keeps constant until the external request rate is less than 7 req/sec. After this threshold, the operation op_{11} begins to be overloaded and therefore the response time of the composite service sharply increases. The SOS performance changes significantly when the QoS Monitor with the adaptive Cusum algorithm is turned on, as shown in Figure 5.27. The response time of the composite service is constant, regardless of the external request rate submitted to op_{11}.



(a) Per-flow vs per-request approach

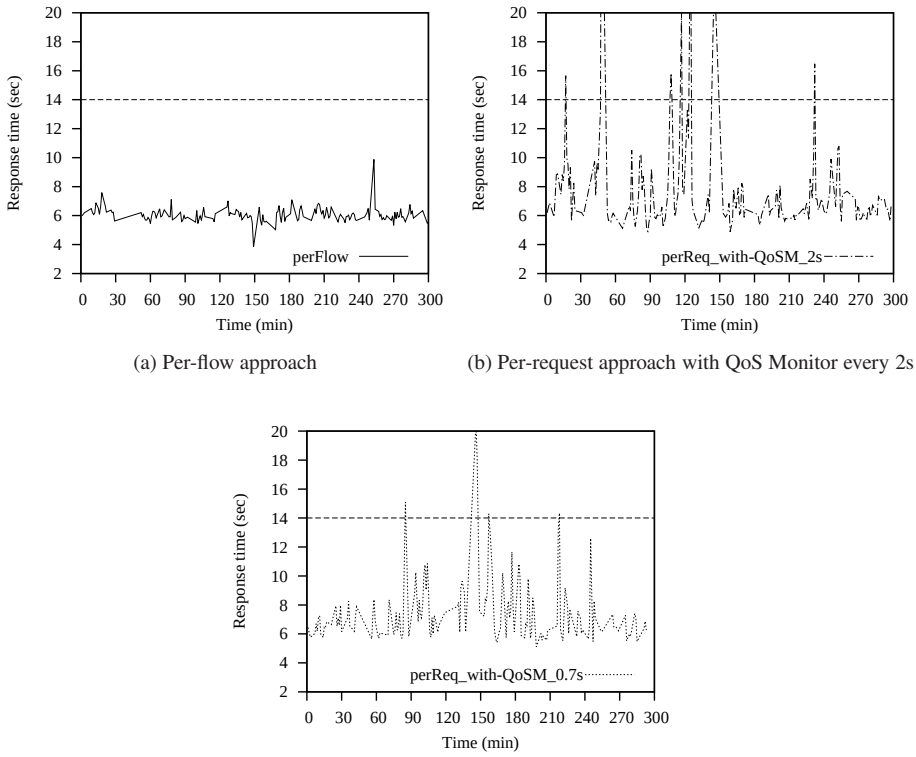


(b) Per-request approach with and without QoS Monitor



(c) Percentage of SLA violation

Figure 5.18: Scenario 1: response time of the composite service for class 1



(a) Per-flow approach

(b) Per-request approach with QoS Monitor every 2s

(c) Per-request approach with QoS Monitor every 0.7s

Figure 5.19: Scenario 2: response time of the composite service over time for class 1

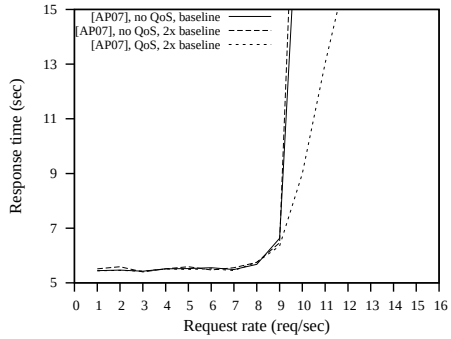


Figure 5.20: Response time of the traditional per-request service selection policy

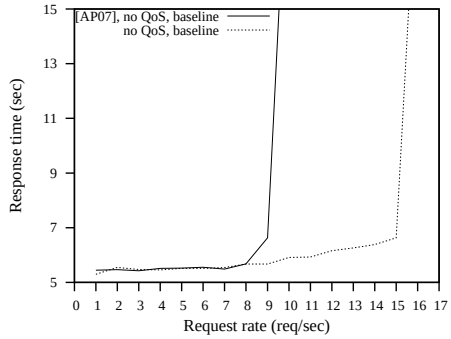


Figure 5.21: Response time of the traditional versus load-aware per-request service selection policies

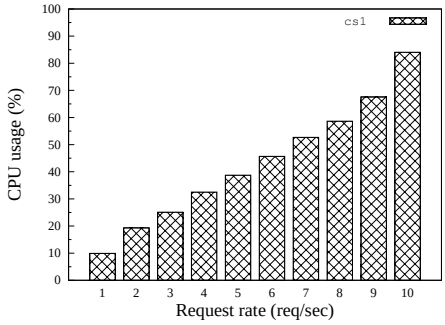


Figure 5.22: CPU usage of the concrete service selected for S_1 by the traditional per-request policy

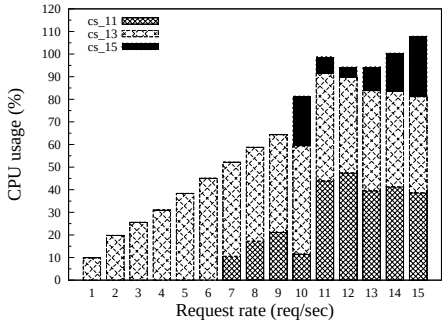


Figure 5.23: CPU usage of the concrete services selected for S_1 by the load-aware per-request policy

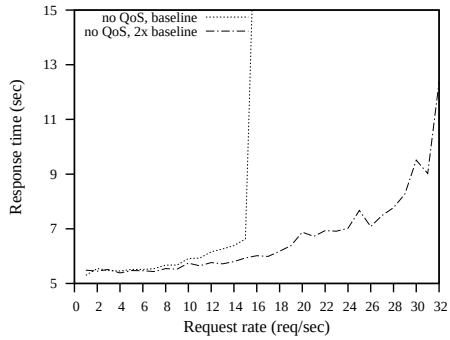
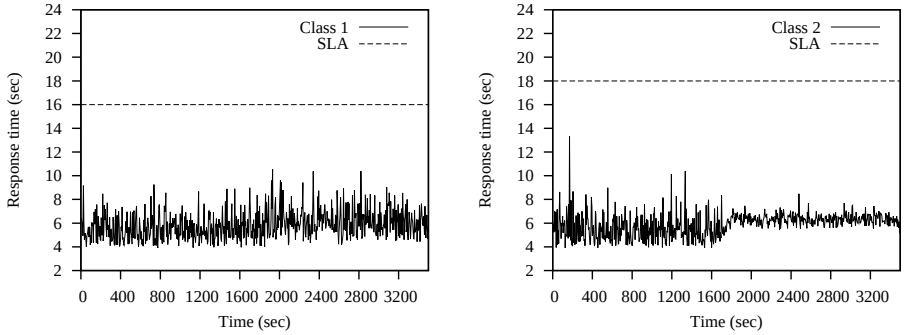
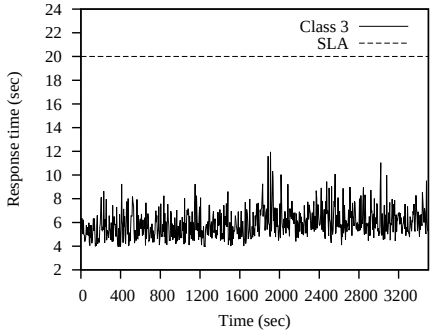


Figure 5.24: Response time of the load-aware per-request policy under the two sets of concrete services

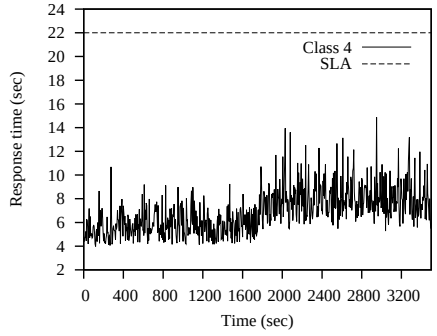


(a) Service class 1

(b) Service class 2



(c) Service class 3



(d) Service class 4

Figure 5.25: Response time of the load-aware per-request policy for all service classes over time

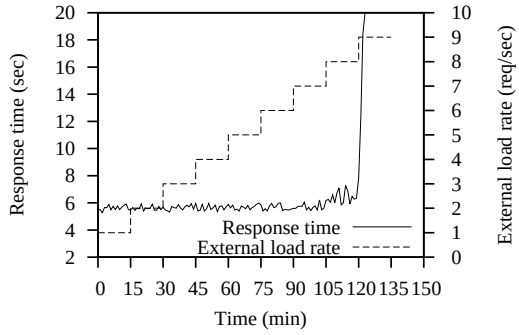


Figure 5.26: Response time of the load-aware per-request policy under external load without QoS Monitor

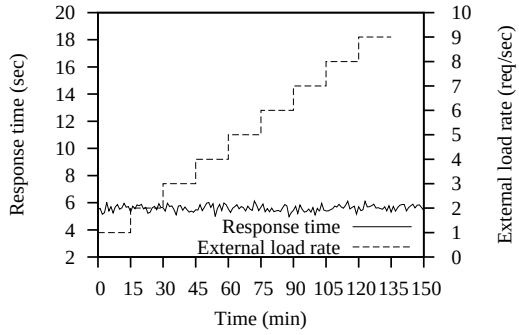


Figure 5.27: Response time of the load-aware per-request policy under external load with QoS Monitor

6 Conclusions

Contents

6.1 Summary 167
6.2 Future Work 169

6.1 Summary

In this thesis we addressed the problem of providing SOSs that satisfy QoS requirements. To this end, we designed and realized MOSES, an autonomic service-oriented broker which is able:

- to self-optimize the execution of SOA applications according to a given service selection strategy;
- to self-configure its components according to a possibly changing environment;
- to self-heal its components and the applications it serves in case of failures.

Since our aim was to realize a software product which could be able to work in real operating environment, we devoted much attention both to its modularity and ease of use and to the performance aspects. The broker is now mature enough and it is being released to the community so to provide a real testbed where experimenting different service selection strategies or even adaptation features.

We used MOSES to evaluate existing service selection strategies. In detail, we focused on per-request and per-flow approaches described in Chapter 3, for which we conducted an extensive evaluation as described in Chapter 5. The ability to test the capabilities of such approaches on a real environment made it possible for us to demonstrate their advantages and disadvantages. The advantage of the per-request approach is that it fulfills QoS requirements for each application invocation; however, it is unable to exploit multiple implementations of the same abstract service. On the other hand, the proposed per-flow approach is able to scale over multiple implementations, but it cannot guarantee QoS levels for each invocation. Therefore, we proposed the load-aware per-request service selection solution that trade-offs the pros and cons of the two more traditional approaches.

This new strategy is able to both exploit multiple implementations of the same abstract service and to fulfill QoS requirements for each application invocation, until there are enough resources provided by concrete services. In Section 5.4.4 we conducted an extensive test to demonstrate its effectiveness through a direct comparison with the existing per-request approach [12]. Results show that both the service selection approaches perform the same until the submitted request rate is sustainable by a single implementation of each abstract service. At higher request rates the per-request approach is unable to serve requests fulfilling QoS as the load-aware per-request does.

In order to evaluate the performance of MOSES and the effectiveness of the adaptation strategies, we needed to use non-shared hardware. To this end and to easily scale and deploy MOSES components, we designed and realized a cloud architecture for a private IaaS provider as described in Section 4.1. This architecture, which has been entirely implemented with opensource software, has been employed as the foundation

for a cloud stack providing MOSES at the SaaS layer.

6.2 Future Work

The research work addressed in this thesis leaves some open questions for future research. In the following, we devise some of these aspects that we plan to address.

Although MOSES can scale by exploiting underlying Cloud resources, it still remains a centralized broker. A decentralized approach is proposed in [41], where a single workflow is split into several sub-workflows. The authors therefore do not only consider service selection as a mean to reach the desired QoS, but they also investigate on how to partition the workflow and where to deploy resulting sub-workflows. However, such an approach only targets one of the aspects of our envisioned decentralized version of MOSES: we speculate a distribution of the whole MAPE loop among multiple MOSES brokers. Under the hypothesis of federated cooperating brokers, this would require to devise a distributed solution of the overall optimization problem. Under the hypothesis of competing brokers, MOSES should be more deeply restructured. In this respect, we note that our characterization of the problem space of self-adaptation for SOA systems evidences that the case of several self-adaptive SOA systems under cooperating or non-cooperating scenarios is not yet satisfactorily covered by current literature. Hence, investigating how to cope with these issues is a timely and promising indication for our future work on the MOSES framework.

Besides this, there are several other directions along which we plan to continue our work on the MOSES framework, as we outline below. A first direction consists in dealing with requirements concerning higher moments and percentiles of QoS attributes. In this respect, a first step towards the inclusion of percentile-based SLAs in MOSES is

presented in [26]. Moreover, we are investigating how to extend the set of assumptions under which MOSES currently works. This includes: relaxing the synchronous invocation assumption; considering alternative failure models (e.g., Byzantine failures, which require different kinds of redundancy patterns); including additional orchestration patterns for service composition, with respect to those matching the grammar presented in Section 3.2.1.

A further direction is related to the assumption, in the current MOSES framework implementation, of a known pool of candidate concrete services, without considering how this pool can be selected and possibly changed at runtime, and the relevant SLA parameters which can be dynamically negotiated. This is a relevant issue, and dealing with it should be one of the tasks of the upper layer of MOSES, according to the three-layers model presented in [57].

Finally, as regards the service selection strategy, it would be interesting to investigate network awareness, that is to take into account the network location of the available service implementations with respect to the broker. Existing policies, including those we proposed, usually neglect network-related QoS. Although some proposals of self-adaptive network-aware service composition have been recently presented [47, 55, 94], they do not consider the exploitation of multiple coordination patterns; furthermore, a performance evaluation on a real testbed is still missing. Such network awareness would also allow to address new scenarios for service composition that arise from the convergence between Internet of Things, Fog computing [17] and Cloud computing.

Bibliography

[1] W. Aalst, A. Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[2] G. Aceto, A. Botta, W. De Donato, and A. Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013.

[3] V. Agarwal and P. Jalote. From specification to adaptation: an integrated qos-driven approach for dynamic adaptation of web service compositions. In *Proc. of 2010 IEEE Int'l Conf. on Web Services (ICWS '10)*, pages 275–282. IEEE, 2010.

[4] T. Ahmed and A. Srivastava. Minimizing waiting time for service composition: A frictional approach. In *Proc. of 2013 IEEE Int'l Conf. on Web Services*, ICWS'13, pages 268–275, June 2013.

[5] G. H. Alférez, V. Pelechano, R. Mazo, C. Salinesi, and D. Diaz. Dynamic adaptation of service compositions with variability models. *Journal of Systems and Software*, 91:24–47, 2014.

[6] M. Alrifai and T. Risse. Combining global optimization with local selection for efficient qos-aware service composition. In *Proceedings of the 18th International Conference on World Wide Web*, pages 881–890. ACM, 2009.

[7] M. Alrifai, D. Skoutas, and T. Risse. Selecting skyline services for qos-based web service composition. In *Proceedings of the 19th International Conference on World Wide Web*, pages 11–20. ACM, 2010.

[8] J. Andersson, R. de Lemos, S. Malek, and D. Weyns. Modeling dimensions of self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 27–47. Springer, 2009.

[9] D. Ardagna, L. Baresi, S. Comai, M. Comuzzi, and B. Pernici. A service-based framework for flexible business processes. *IEEE Software*, 28(2), 2011.

[10] D. Ardagna, M. Comuzzi, E. Mussi, B. Pernici, and P. Plebani. Paws: A framework for executing adaptive web-service processes. *IEEE Software*, 24(6):39–46, 2007.

[11] D. Ardagna and R. Mirandola. Per-flow optimal service selection for web services based processes. *Journal of Systems and Software*, 83(8):1512–1523, 2010.

[12] D. Ardagna and B. Pernici. Adaptive service composition in flexible processes. *IEEE Trans. Softw. Eng.*, 33(6):369–384, 2007.

[13] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[14] A. Barkat, A. D. d. Santos, and T. T. N. Ho. Open stack and cloud stack: Open source solutions for building public and private clouds. In *Proc. of 16th Int'l Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 429–436. IEEE, 2014.

[15] A. Bellucci, V. Cardellini, V. Di Valerio, and S. Iannucci. A scalable and highly available brokering service for SLA-based composite services. In *Service-Oriented Computing*, pages 527–541. Springer, 2010.

[16] R. Berbner, M. Spahn, N. Repp, O. Heckmann, and R. Steinmetz. Heuristics for qos-aware web service composition. In *Proc. of 2006 IEEE Int'l Conf. on Web Services (ICWS'06)*, pages 72–82. IEEE, 2006.

[17] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the Internet of Things. In *Proc. of 1st Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16. ACM, 2012.

[18] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web Services Architecture (WSA). W3C Working Group Note 11 Feb. 2004.

[19] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change: Research articles. *J. Softw. Maint. Evol.*, 17(5):309–332, 2005.

[20] R. Buyya, C. Vecchiola, and S. T. Selvi. *Mastering Cloud Computing: Foundations and Applications Programming*. Morgan Kaufmann, 2013.

[21] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qos management and optimization in service-based systems. *IEEE Transactions on Software Engineering*, 37(3):387–409, 2011.

[22] G. Canfora, M. Di Penta, R. Esposito, and M. Villani. A framework for QoS-aware binding and re-binding of composite web services. *J. Syst. Softw.*, 81(10):1754–1769, 2008.

[23] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. Lo Presti, and R. Mirandola. MOSES: a framework for QoS driven runtime adaptation of service-oriented systems. *IEEE Trans. Softw. Eng.*, 38(5):1138–1159, Sept. 2012.

[24] V. Cardellini, E. Casalicchio, V. Grassi, and F. Lo Presti. Flow-based service selection forweb service composition supporting multiple qos classes. In *Proc. of 2007 IEEE Int'l Conf. on Web Services (ICWS 2007)*, pages 743–750. IEEE, 2007.

[25] V. Cardellini, E. Casalicchio, V. Grassi, F. Lo Presti, and R. Mirandola. Qos-driven runtime adaptation of service oriented architectures. In *Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 131–140. ACM, 2009.

[26] V. Cardellini, E. Casalicchio, V. Grassi, and F. L. Presti. Adaptive management of composite services under percentile-based service level agreements. In *Service-Oriented Computing*, pages 381–395. Springer, 2010.

[27] V. Cardellini, V. Di Valerio, V. Grassi, S. Iannucci, and F. Lo Presti. Qos driven per-request load-aware service selection in service oriented architectures. *International Journal of Software and Informatics*, 7(2):195–220, 2013.

[28] V. Cardellini and S. Iannucci. Designing a broker for qos-driven runtime adaptation of soa applications. In *Proc. of 2010 IEEE Int'l Conf. on Web Services (ICWS 2010)*, pages 504–511. IEEE, 2010.

[29] V. Cardellini, F. Lo Presti, V. Grassi, and E. Casalicchio. Scalable service selection for web service composition supporting differentiated qos classes. *Technical Report RR-07.59, Dip. di Informatica, Sistemi e Produzione*, Feb. 2007.

[30] J. Cardoso. Complexity analysis of bpel web processes. *Software Process: Improvement and Practice*, 12(1):35–49, 2007.

[31] S. Casolari, S. Tosi, and F. Lo Presti. An adaptive model for online detection of relevant state changes in internet-based systems. *Perform. Eval.*, 69(5):206–226, May 2012.

[32] A. Charfi and M. Mezini. Ao4bpel: An aspect-oriented extension to bpel. *World Wide Web*, 10(3):309–344, 2007.

[33] B. Chen, X. Peng, Y. Yu, and W. Zhao. Requirements-driven self-optimization of composite services using feedback control. *IEEE Transactions on Services Computing*, pages In–press, 2014.

[34] B. H. C. Cheng, H. Giese, P. Inverardi, J. Magee, and R. de Lemos. 08031 – software engineering for self-adaptive systems: A research road map. In *Software Engineering for Self-Adaptive Systems*, volume 08031 of *Dagstuhl Seminar Proceedings*. IBFI, 2008.

[35] M. Colombo, E. D. Nitto, and M. Mauri. Scene: A service composition execution environment supporting dynamic changes disciplined through rules. In *Proc. ICSOC '06*, volume 4294 of *LNCS*, pages 191–202. Springer, 2006.

[36] M. Colombo, E. D. Nitto, M. D. Penta, D. Distante, and M. Zuccalà. Speaking a common language: A conceptual model for describing service-oriented systems. In *Proc. ICSOC '05*, volume 3826 of *LNCS*, pages 48–60. Springer, 2005.

[37] W. R. Cook, S. Patwardhan, and J. Misra. Workflow patterns in Orc. In *Proc. COORDINATION '06*, volume 4038 of *LNCS*, pages 82–96. Springer, 2006.

[38] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms 2nd Edition*. 2001.

[39] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 1(2):223–259, 2006.

[40] DRBD. `http://www.drbd.org/`.

[41] D. Efstathiou, P. McBurney, S. Zschaler, and J. Bourcier. Flexible qos-aware service composition in highly heterogeneous and dynamic service-based systems. In *IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), 2013*, pages 592–599. IEEE, 2013.

[42] O. Ezenwoye and S. M. Sadjadi. Robustbpel2: Transparent autonomization in business processes through dynamic proxies. In *2007 IEEE Eighth International Symposium on Autonomous Decentralized Systems (ISADS'07)*, pages 17–24. IEEE, 2007.

[43] O. Ezenwoye and S. M. Sadjadi. A proxy-based approach to enhancing the autonomic behavior in composite services. *Journal of Networks*, 3(5):42–53, 2008.

[44] The Grinder. `http://sourceforge.net/projects/grinder/`.

[45] Q. He, J. Yan, H. Jin, and Y. Yang. Quality-aware service selection for service-based systems based on iterative multi-attribute combinatorial auction. *Software Engineering, IEEE Transactions on*, 40(2):192–215, 2014.

[46] httperf. `http://www.hpl.hp.com/research/linux/httperf/`.

[47] J. Huang, G. Liu, Q. Duan, and Y. Yan. Qos-aware service composition for converged network-cloud service provisioning. In *Proc. of IEEE 2014 Int'l Conf. on Services Computing (SCC)*, pages 67–74. IEEE, 2014.

[48] M. C. Huebscher and J. A. McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv.*, 40(3), 2008.

[49] C. Hwang and K. Yoon. *Multiple Criteria Decision Making, Lecture Notes in Economics and Mathematical Systems*. Springer, 1981.

[50] A. Immonen and D. Pakkala. A survey of methods and approaches for reliable dynamic service compositions. *Service Oriented Computing and Applications*, 8(2):129–158, 2014.

[51] JBI. `https://jcp.org/en/jsr/detail?id=208`.

[52] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[53] D. Kitchin, A. Quark, W. R. Cook, and J. Misra. The Orc programming language. In *Proc. FMOODS/FORTE '09*, volume 5522 of *LNCS*, pages 1–25. Springer, 2009.

[54] A. Klein, F. Ishikawa, and S. Honiden. Efficient qos-aware service composition with a probabilistic service selection policy. In *Service-Oriented Computing*, pages 182–196. Springer, 2010.

[55] A. Klein, F. Ishikawa, and S. Honiden. SanGA: A self-adaptive network-aware approach to service composition. *IEEE Transactions on Services Computing*, 7(3):452–464, 2014.

[56] M. Koning, C.-a. Sun, M. Sinnema, and P. Avgeriou. Vxbpel: Supporting variability for web services in bpel. *Information and Software Technology*, 51(2):258–269, 2009.

[57] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering, 2007. FOSE'07*, pages 259–268. IEEE, 2007.

[58] P. Leitner, W. Hummer, and S. Dustdar. Cost-based optimization of service compositions. *Services Computing, IEEE Transactions on*, 6(2):239–251, 2013.

[59] P. Leitner, B. Wetzstein, D. Karastoyanova, W. Hummer, S. Dustdar, and F. Leymann. Preventing SLA violations in service compositions using aspect-based fragment substitution. In *Service-Oriented Computing*, pages 365–380. Springer, 2010.

[60] Q. Liang, X. Wu, and H. Lau. Optimizing service systems based on application-level QoS. *IEEE Trans. Serv. Comput.*, 2(2):108–121, Apr. 2009.

[61] Z. Maamar, Q. Z. Sheng, and B. Benatallah. Interleaving web services composition and execution using software agents and delegation. In *Proc. WSABE '03*, 2003.

[62] S. Martello and P. Toth. Algorithms for knapsack problems. *North-Holland Mathematics Studies*, 132:213–257, 1987.

[63] P. Martin, W. Powley, K. Wilson, W. Tian, T. Xu, and J. Zebedee. The wsdm of autonomic computing: experiences in implementing autonomic web services. In *Proc. SEAMS '07*, 2007.

[64] P. McKinley, S. Sadjadi, E. Kasten, and B. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, 2004.

[65] D. Menascé, E. Casalicchio, and V. Dubey. On optimal service selection in service oriented architectures. *Perform. Eval.*, 67(8):659–675, Aug. 2010.

[66] D. Menasce, H. Gomaa, S. Malek, and J. P. Sousa. Sassy: A framework for self-architecting service-oriented systems. *IEEE Software*, 28(6):78–85, 2011.

[67] D. A. Menascé. Qos issues in web services. *IEEE Internet Comp.*, 6(6):72–75, 2002.

[68] D. A. Menascé, J. M. Ewing, H. Gomaa, S. Malex, and J. P. Sousa. A framework for utility-based service oriented design in sassy. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, pages 27–36. ACM, 2010.

[69] D. A. Menascé, H. Ruan, and H. Gomaa. Qos management in service-oriented architectures. *Performance Evaluation*, 64(7):646–663, 2007.

[70] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. End-to-end support for qos-aware service selection, binding, and mediation in vresco. *IEEE Transactions on Services Computing*, 3(3):193–205, 2010.

[71] R. Mirandola and P. Potena. A qos-based framework for the adaptation of service-based systems. *Scalable Computing: Practice and Experience*, 12(1), 2011.

[72] D. Montgomery. *Introduction to Statistical Quality Control*. Wiley, 2008.

[73] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for WS-BPEL. In *Proc. WWW '08*, pages 815–824. ACM, 2008.

[74] O. Moser, F. Rosenberg, and S. Dustdar. Domain-specific service selection for composite services. *IEEE Transactions on Software Engineering*, 38(4):828–843, 2012.

[75] A. Mosincat, W. Binder, and M. Jazayeri. Achieving runtime adaptability through automated model evolution and variant selection. *Enterprise Information Systems*, 8(1):67–83, 2014.

[76] A. Mostafa, M. Zhang, and Q. Bai. Trustworthy stigmergic service composition and adaptation in decentralized environments. *IEEE Transactions on Services Computing*, 2014.

[77] A. Moustafa and M. Zhang. Multi-objective service composition using reinforcement learning. In *Service-Oriented Computing*, pages 298–312. Springer, 2013.

[78] P. Nagpurkar, W. Horn, U. Gopalakrishnan, N. Dubey, J. Jann, and P. Pattnaik. Workload characterization of selected jee-based web 2.0 applications. In *Proc. IEEE IISWC '08*, pages 109–118, 2008.

[79] R. Nelson. *Probability, stochastic processes, and queueing theory*. Springer-Verlag, New York, 1995.

[80] Nimbus. `http://www.nimbusproject.org/`.

[81] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. In *Proc. of IEEE/ACM CCGRID '09*, 2009.

[82] OASIS. Web Services Business Process Execution Language Version 2.0, Jan. 2007.

[83] OpenESB. `http://www.open-esb.net/`.

[84] OpenNebula. `http://opennebula.org/`.

[85] openQRM. `http://www.openqrm.com/`.

[86] OpenStack. `http://www.openstack.org/`.

[87] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[88] S. Rosario, A. Benveniste, S. Haar, and C. Jard. Probabilistic qos and soft contracts for transaction-based web services orchestrations. *IEEE Transactions on Services Computing*, 1(4):187–200, 2008.

[89] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz. Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments. In *Software Engineering for Self-Adaptive Systems*, pages 164–182. Springer, 2009.

[90] R. Rouvoy, F. Eliassen, J. Floch, S. Hallsteinsen, and E. Stav. Composing components and services using a planning-based adaptation middleware. In *Software Composition*, pages 52–67. Springer, 2008.

[91] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.

[92] D. Schuller, M. Siebenhaar, R. Hans, O. Wenge, R. Steinmetz, and S. Schulte. Towards heuristic optimization of complex service-based workflows for stochastic qos attributes. In *Web Services (ICWS), 2014 IEEE International Conference on*, pages 361–368. IEEE, 2014.

[93] P. Sempolinski and D. Thain. A comparison and critique of Eucalyptus, OpenNebula and Nimbus. *Proc. of IEEE CloudCom '10*, pages 417–426, 2010.

[94] J. Siebert, J. Cao, Y. Lai, P. Guo, and W. Zhu. LASEC: A localized approach to service composition in pervasive computing environments. *IEEE Transactions on Parallel and Distributed Systems*, 2014. to appear.

[95] A. Strunk. Qos-aware service composition: A survey. In *Web Services (ECOWS), 2010 IEEE 8th European Conference on*, pages 67–74. IEEE, 2010.

[96] L. Sun, S. Wang, J. Li, Q. Sun, and F. Yang. Qos uncertainty filtering for fast and reliable web service selection. In *Proc. of 2014 IEEE Int'l Conf. on Web Services (ICWS'14)*, pages 550–557. IEEE, 2014.

[97] Y. Syu, S.-P. Ma, J.-Y. Kuo, and Y.-Y. FanJiang. A survey on automated service composition methods and related techniques. In *IEEE 9th International Conference on Services Computing (SCC), 2012*, pages 290–297. IEEE, 2012.

[98] D. Teigland and H. Mauelshagen. Volume managers in Linux. *Proc. of USENIX ATC '01*, 2001.

[99] W.-T. Tsai, X. Sun, and J. Balasooriya. Service-oriented cloud computing architecture. In *Proc. of 7th Int'l Conf. on Information Technology: New Generations (ITNG 2010)*, pages 684–689. IEEE, 2010.

[100] H. Wada, J. Suzuki, Y. Yamano, and K. Oba. A multiobjective optimization framework for SLA-aware service composition. *IEEE Transactions on Services Computing*, 5(3):358–372, 2012.

[101] X. Wen, G. Gu, Q. Li, Y. Gao, and X. Zhang. Comparison of open-source cloud management platforms: OpenStack and OpenNebula. In *Proc. of 9th Int'l Conf. on Fuzzy Systems and Knowledge Discovery (FSKD)*, pages 2457–2461. IEEE, 2012.

[102] S. Wind. Open source cloud computing management platforms: Introduction, comparison, and recommendations for implementation. In *Proc. of IEEE ICOS '11*, pages 175–179, 2011.

[103] Q. Wu, Q. Zhu, X. Jian, and F. Ishikawa. Broker-based SLA-aware composite service provisioning. *Journal of Systems and Software*, 96:194–201, 2014.

[104] T. Yu, Y. Zhang, and K.-J. Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Trans. Web*, 1(1):1–26, 2007.

[105] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnamam, and H. Chang. QoS-aware middleware for web services composition. *IEEE Trans. Softw. Eng.*, 30(5), May 2004.

[106] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, May 2010.

[107] Z. Zheng and M. R. Lyu. Collaborative reliability prediction of service-oriented systems. In *Proc. of 32nd ACM/IEEE Int'l Conf. on Software Engineering*, volume 1, pages 35–44. ACM, 2010.