Adaptive Web Services for Modular and Reusable Software Development:

Tactics and Solutions

Guadalupe Ortiz
University of Cádiz, Spain

Javier Cubo
University of Málaga, Spain

Information Science
REFERENCE

Managing Director: Lindsay Johnston
Senior Editorial Director: Heather A. Probst
Book Production Manager: Jennifer Romanchak
Publishing Systems Analyst: Adrienne Freeland
Managing Editor: Joel Gamon
Development Editor: Hannah Abelbeck
Assistant Acquisitions Editor: Kayla Wolfe
Typesetter: Travis Gundrum
Cover Design: Nick Newcomer

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Adaptive web services for modular and reusable software development: tactics and solutions / Guadalupe Ortiz and Javier Cubo, editors.
p. cm.
Includes bibliographical references and index.
Summary: “The book comprises chapters that present tactics and solutions for modular and reusable software development in the field of adaptive Web services”--Provided by publisher.
ISBN 978-1-4666-2089-6 (hardcover) -- ISBN 978-1-4666-2090-2 (ebook) -- ISBN 978-1-4666-2091-9 (print & perpetual access)
1. Web services. 2. Computer software--Reusability. 3. Component software 4. Computer software--Development. I. Ortiz, Guadalupe, 1977- II. Cubo, Javier, 1978-
TK5105.88813.A365 2012
006.7’8--dc23
2012013952

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 6
Service-Oriented Systems for Adaptive Management of Service Composition

Valeria Cardellini
University of Roma "Tor Vergata," Italy

Stefano Iannucci
University of Roma "Tor Vergata," Italy

Valerio Di Valerio
University of Roma "Tor Vergata," Italy

Francesco Lo Presti
University of Roma "Tor Vergata," Italy

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 makes it 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. In this chapter, the authors first review some representative frameworks that have been proposed for SOSs able to adaptively manage a SOA application with QoS requirements. These frameworks are commonly architected as self-adaptive systems following the MAPE-K (Monitor, Analyze, Plan, Execute, and Knowledge) reference model for autonomic computing. The chapter organizes the review using a specific taxonomy for each MAPE-K phase, with the aim to classify the different strategies and mechanisms that can be applied. Even if a self-adaptive system requires every MAPE-K phase, the authors then focus on the Plan phase, which is the core of each adaptation framework, presenting both optimal and sub-optimal approaches that have been proposed to effectively face the adaptation task at runtime.

DOI: 10.4018/978-1-4666-2089-6.ch006

INTRODUCTION

As a case study of SOS for the adaptive management of service composition, we present the main features of a prototype that follows the MAPE-K reference model. We analyze through a set of experiments the different degrees of reliability achieved by a SOA application able or not to detect and adapt its behavior with respect to the churn of the services used to compose it. Our experimental results show that the SOA application managed by the SOS achieves a reliability improvement up to 20% with respect to its unmanaged counterpart.

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 (Salehie & Tahvildari, 2009), 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. In this chapter, we

focus on non-functional requirements, expressed as Quality of Service (QoS) attributes of SOSs.

The adaptation of non-functional requirements can follow either the best effort or *QoS-constrained* strategy. The former aims to improve non-functional attributes (e.g., response time or reliability) of the overall SOA application without ensuring any kind of guarantee, while the latter aims to provide a SOA application with predictable QoS attributes. In the last years both approaches have been largely investigated, e.g., (Ezenwoye & Sadjadi, 2007; Michlmayr, Rosenberg, Leitner & Dustdar, 2010) for the best effort strategy and (Ardagna, & Pernici, 2007; Cardellini, Casalicchio, Grassi, Iannucci, Lo Presti & Mirandola, in press; Menascè, Casalicchio & Dubey, 2010) for the QoS-constrained 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 above efforts have focused 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 (Kephart & Chess, 2003) is a conceptual guideline for realizing self-adaptable systems (Salehie et al., 2009) 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 maximize some utility function, and executes these actions. In literature, the same approach is also referred to as CADA (Dobson et al., 2006), which stands for Collect, Analyze, Design, and Act.

In this chapter we first present and classify the different frameworks for adaptive management of service composition. Since they can all be regarded as instances of the MAPE-K framework we present each phase of the MAPE-K loop and discuss how the self-adaptation frameworks for SOA applications, so far proposed, implement it.

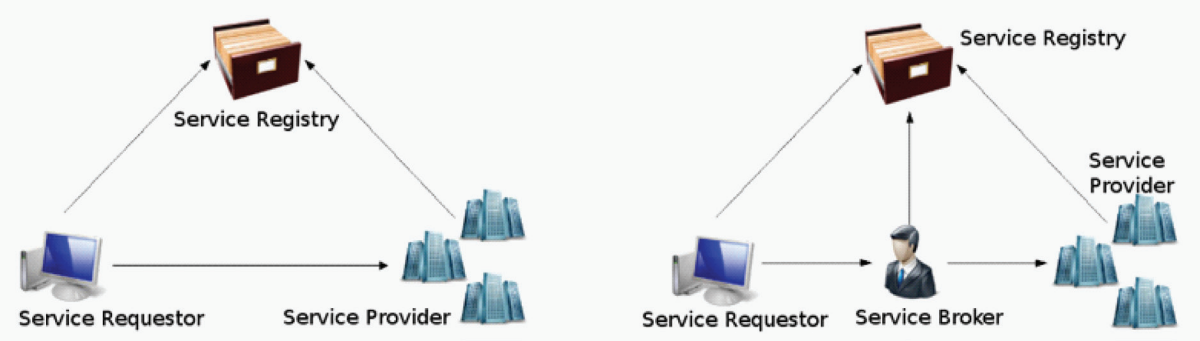
We then concentrate on the adaptation strategies themselves. In particular, since we focus on fulfilling the non-functional requirements of the SOA application, we analyze the approaches for service selection. Given a service composition, the service selection's aim is to identify those component services that provide the best implementation of the needed functionalities in order to satisfy the QoS requirements of the SOA application. Although service selection is not the only adaptation mechanism, our presentation focuses on it because it is leveraged by most of the self-adaptation frameworks we consider in our review.

As case study of SOS for the adaptive management of service composition, we then present MOSES (Cardellini et al., in press). MOSES is a methodology and a software tool implementing it to support the QoS-driven adaptation of a service-oriented system and represents a working example of framework organized according to the MAPE-K loop.

We validate on a motivating scenario characterized by a varying operating environment the runtime adaptation features provided by MOSES. Specifically, we analyze through a set of experiments conducted using the MOSES prototype the different degrees of reliability achieved by a SOA application able or not to detect changes and adapt its behavior with respect to the component services churn, i.e., to the change in the set of component services due to joins, graceful leaves, and failures. Our experimental results show that the runtime adaptation carried out by MOSES is able to improve the SOA application reliability even in a highly varying operating environment.

The rest of the chapter is organized as follows. The section below introduces some basic terminol-

Figure 1. a) SOA reference model; b) SOA reference model with broker



ogy used throughout the chapter. In the section after that we review the distinguishing features of the MAPE-K cycle and present our taxonomy for each specific MAPE-K phase. In the following two sections, we focus on the service selection approaches used in the Plan phase of MAPE-K, analyzing some representative formulations of optimization problems and heuristics. In the sixth section, we present our case study of adaptive SOS and analyze the experimental results. Finally, we present possible avenues for future work and conclude the chapter with some remarks.

SOA REFERENCE MODEL

Prior to analyze how to realize self-adaptable SOA applications, it is useful to introduce the SOA reference model, so to clarify the basic terminology used throughout the chapter.

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 (in this chapter we will use service requestor and client interchangeably). 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 1a illustrates the SOA reference model.

The above reference model is usually extended to include an intermediary entity. When the offered service is actually a service composition which adopts some adaptation mechanism, we refer to its provider as *service broker*. As shown in Figure 1b, the service broker is as an intermediary actor lying between the service requestors and the service providers, willing to offer to the requestors an added-value SOA application obtained by composing the services exposed by those providers.

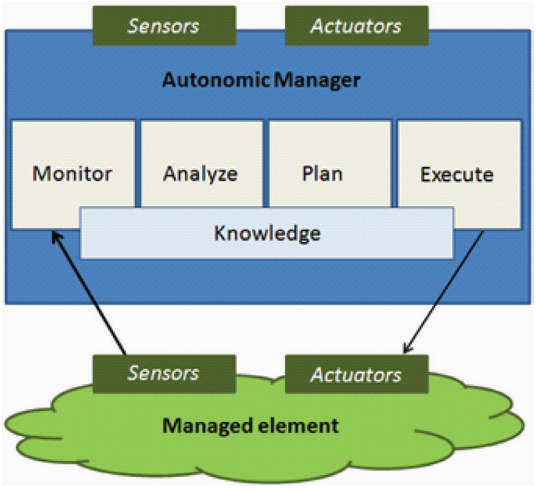
As we will see in this chapter, the notion of service broker is crucial for a self-adaptive SOS: although the service requestor can be enhanced with some adaptation logic, a more consolidated approach is to place this logic at the service broker level.

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

MAPE-K: A CONTROL LOOP FOR SELF-ADAPTIVE FRAMEWORKS

MAPE-K is a reference model for realizing self-adaptable applications: the MAPE-K control loop uses an intelligent agent to perceive the surround-

Figure 2. MAPE-K control loop: adapted from (Kephart & Chess, 2003)



ing 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) the workflow of activities concerning the invocation of external services and their orchestration, (ii) the external services, and (iii) the network interconnecting these activities with the service requestors and the service providers. The autonomic manager is constituted by software components for the different MAPE-K phases.

Figure 2 illustrates the MAPE-K control loop: the four steps of the autonomic manager, the managed element, the sensors, and the actuators. In the SOA context, the managed element is the SOA application, while the autonomic manager is a (possibly complex) software layer overlying the actual SOA application. While the application runs, the manager goes through the different MAPE-K steps:

1. **Monitor:** The application execution is monitored through sensors. In the SOA context, the sensors are implemented by means of probes over external services, with the objective of detecting the actual values of the quality attributes such as response time, reliability, and availability.
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 aims at determining whether some quality attribute has violated (or is going to violate) a previously specified internal policy, usually stored in the Knowledge layer. In the SOA domain, an internal policy can be the violation of a certain threshold for a quality attribute, e.g., for a given service the average response time measured over some interval exceeds the threshold established in the internal policy.
3. **Plan:** After the Analyze phase has detected some kind of violation of the internal policy, the Plan phase computes a new adaptation plan, possibly using the data elaborated by the Analyze phase with the support of the Knowledge layer. In the SOA context, the elaboration of a new adaptation plan can be the selection of different service providers implementing the needed functionalities. Alternatively, it can be an internal workflow re-arrangement so that the internal policy specifying the application requirements can 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 can be a different binding of functionalities to service providers, as well as an application re-deployment.

In the remainder of this chapter we will first describe the different phases of the MAPE-K loop intended for SOA applications; then, we will focus on the planning phase. For each MAPE phase we

arranged a taxonomy to classify the different SOA frameworks that adopt the autonomic control loop to adaptively manage the service composition.

The questions driving the various taxonomies are based on the five Ws and one H concept (Hart, 2011).

- **What**: It identifies the relevant elements for each MAPE-K phase.
- **Where**: It characterizes where a certain phase can happen either at a logical or physical level.
- **When**: It classifies the temporal aspects that characterize each MAPE-K phase.
- **Who**: It identifies the entities involved in the execution of each MAPE-K phase.
- **How**: It describes how each MAPE-K phase can be implemented.

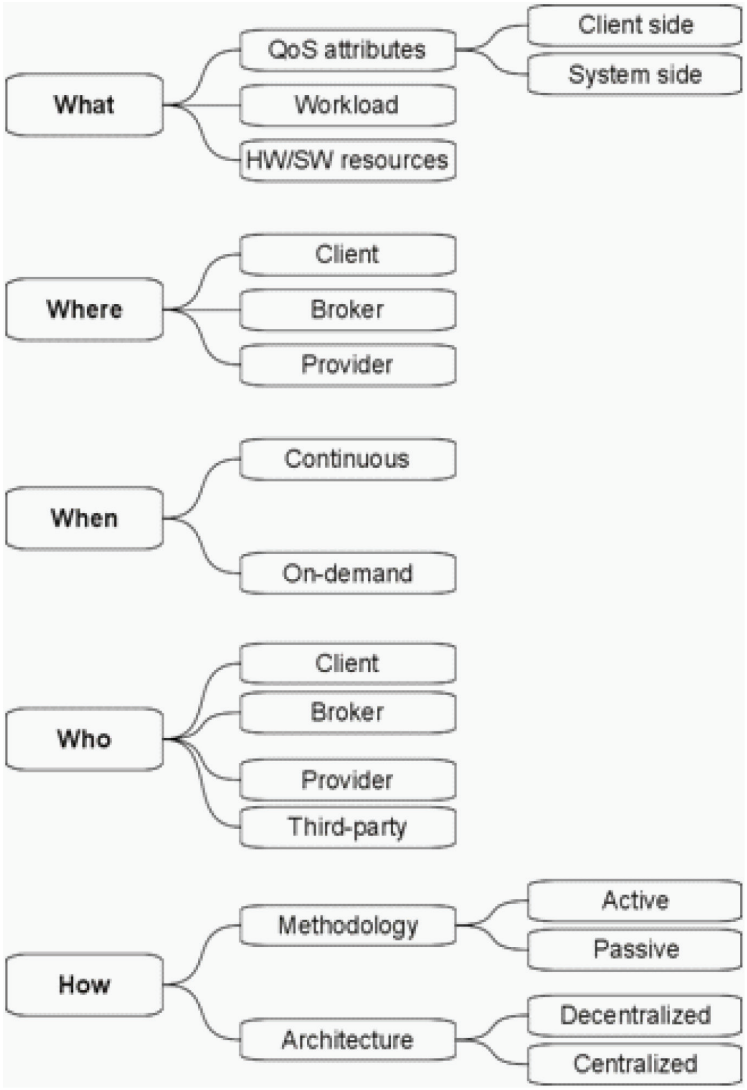
With respect to the five Ws and one H concept, we do not explicitly consider the Why question, because we assume that adaptation is the motivation that drives all the choices.

Monitor Taxonomy

Figure 3 illustrates the taxonomy of the Monitor phase.

- **What:** Monitoring usually targets the 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 can regard the CPU utilization or the amount of available memory, while those regarding the software resources can 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, reputation, and cost, which capture how the clients perceive the application QoS; (ii) system-side parameters, like throughput and cost, which are relevant to the system managers. Reputation, which provides a measure of the service trustworthiness, can be defined as the ratio between the number of service invocations that comply the negotiated QoS over the total number of service invocations (Ardagna & Pernici, 2007) and can be obtained through a collaborative mechanism among the application clients (Zheng, Ma, Lyu & King, 2011). 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 (Mirandola & Potena, 2011; Calinescu, Grunske, Kwiatkowska, Mirandola & Tamburrelli, 2011), 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 (Rouvoy et al., 2009; Menascé, Gomaa, Malek & Sousa, 2011; Bellucci, Cardellini, Di Valerio & Iannucci, 2010; Agarwal & Jalote, 2010; Ardagna, Baresi, Comai, Comuzzi & Pernici, 2011). Furthermore, the workload submitted to the SOA application can also be monitored, for example to derive some useful metric, like response time, from the gathered information. Examples of frameworks that monitor the workload include (Calinescu et al., 2011; Bellucci et al., 2010; Ardagna & Mirandola, 2010).
- **Where:** The monitored data can be collected at various different locations. A first approach is to collect the data at the client side

Figure 3. Monitor taxonomy



of the SOA application, like in (Rouvoy et al., 2009), where the client is responsible for detecting possible Service Level Agreements (SLA) violations. Another approach is to collect the data at the provider side, like 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 (Mirandola et al., 2011; Calinescu et al., 2011; Menascé et al., 2011; Bellucci et al., 2010; Ardagna et al., 2011).

- **When:** The monitoring activity can be accomplished either continuously or on-demand. Although the latter seems to be a reasonable solution, for example the planning phase might choose to start a monitoring activity on a different perspective of the system, in all the frameworks we

consider the monitoring is performed on a time-continuous base. The frequency at which data are collected often depends on the required effort.

- **Who:** Various actors can be interested in the monitoring activity: the client might want to detect SLA violations, the broker to identify changes in the operational environment, and the provider to control the resource utilization. Furthermore, a third-party entity not directly involved in the SOA application might collect data in order to offer them as a 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, (Calinescu et al., 2011; Ardagna et al., 2011; Rouvoy et al., 2009; Mirandola et al., 2011) use the passive approach. The active monitoring can be used to proactively determine the service availability. For example in (Bellucci et al., 2010), besides using a passive approach, the framework periodically checks if the used services are available, in order to reduce the occurrence of a service failure during the invocation issued by a client.
- Finally, the monitoring activities can be performed by a single central entity or by a distributed sensors network.

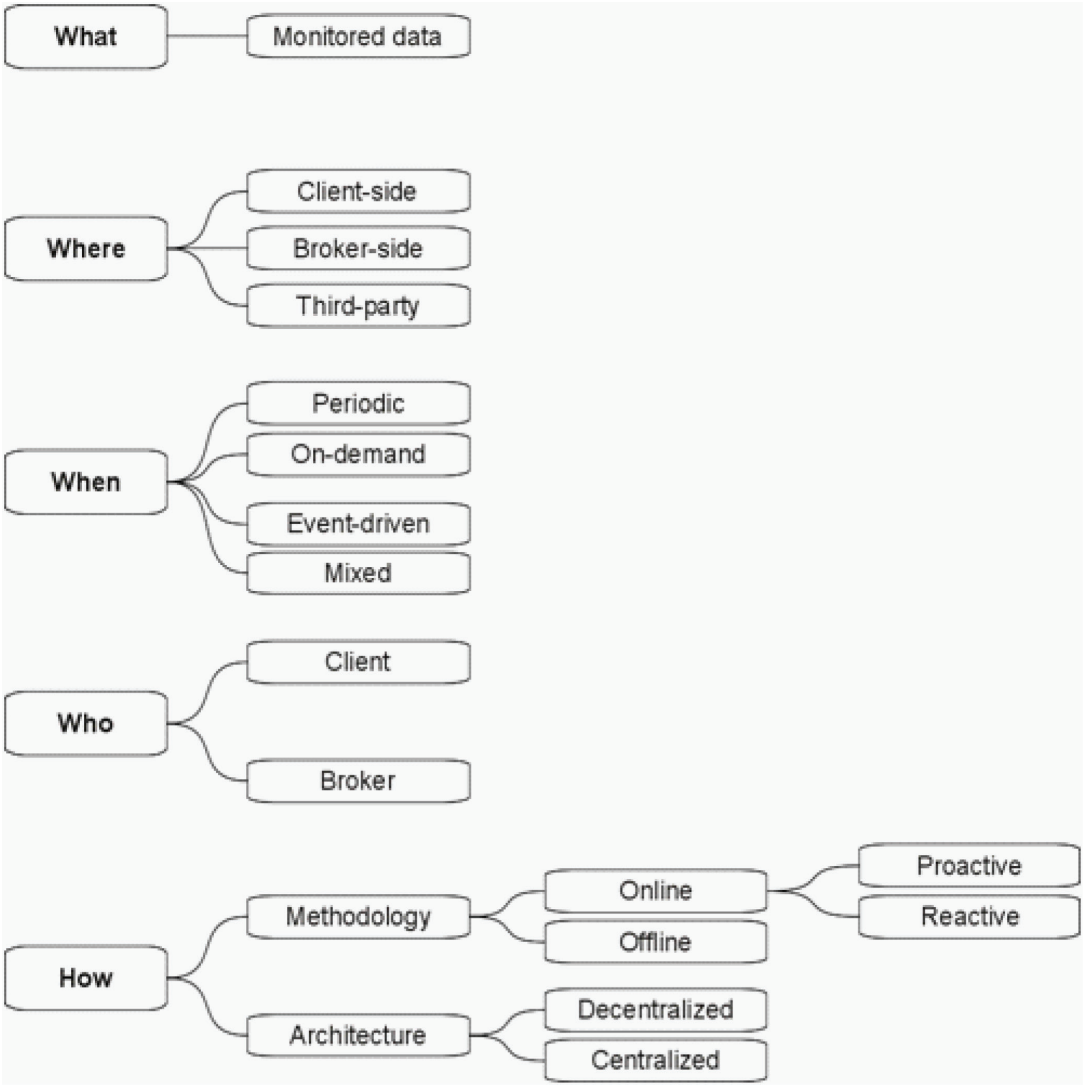
Analyze Taxonomy

Figure 4 depicts the taxonomy for the Analyze phase of the MAPE-K loop.

- **What:** The Analyze phase receives as input the data from the Monitor phase and so it deals with the monitored data. The processed data can be either raw or previously aggregated.
- **Where:** The data analysis can be carried out at different locations: at the client, the broker, and a third-party entity. Client-side analysis is typically carried out in SOA architectures that do not include an intermediary broker; in this case, the analysis of the monitored data is demanded either to a monitoring service under the client control, e.g., (Rosario, Benveniste, Haar & Jard, 2008), or to a third-party collaborative monitoring service as in (Zheng et al., 2011).

  The broker-side analysis is usually performed by those frameworks that involve the broker with the support of either self-collected data or a third-party monitoring system. The latter can be of collaborative type and therefore offers data analysis as a counterpart for receiving monitored data from SOA executors, as in (Zheng et al., 2011).
- **When:** The frequency at which data analysis is performed is often determined as a trade-off between the need to quickly react to significant events and the costs of data processing. The simplest approach is to periodically analyze the data at fixed intervals (Bellucci et al., 2010). In the more sophisticated event-driven analysis, which is usually based on the concept of Continuous Query Processing (CQP), each monitored data is not only stored but might activate a trigger usually based on simple policies, like threshold violations (Calinescu et al., 2011). Event-driven analysis can also occur either after the execution of a specific service, a set of services, or even the whole workflow (Ardagna et al., 2011). The periodic and event-driven analysis approaches

Figure 4. Analyze taxonomy



can be combined to obtain a periodic analysis coupled with an event-driven analysis for critical events detection (Calinescu et al., 2011). Finally, on-demand analysis can also be directly requested by a client, depending on its own analysis policies.

- **Who:** The actors interested in the Analyze phase coincide with those that will plan the adaptation actions, that is, the clients and the broker. A client may be interested in data analysis when it does not rely on an external service broker, while a service broker is always interested in analyzing the monitored data.
- **How:** We distinguish between methodological and architectural issues regarding how the analysis can be accomplished.
- The Analyze 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 (Rosario et al., 2008)), hence not suited to online operations. Offline analysis still plays a significant role as the collected data can be used to identify suitable models of the complex SOA environment.

- Online solutions 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., (Calinescu et al., 2011). 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., (Ardagna et al., 2011).
- 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 are more scalable and fault tolerant.
- The possible data analysis techniques include checking the violation of a threshold, for example by applying the Student t-test statistical significance to determine the probability of a QoS attribute to be violated (Mosincat, Binder, & Jazayeri, 2010), or creating an empirical distribution function that fits the actual QoS parameters distribution as in (Rosario et al., 2008).

Plan Taxonomy

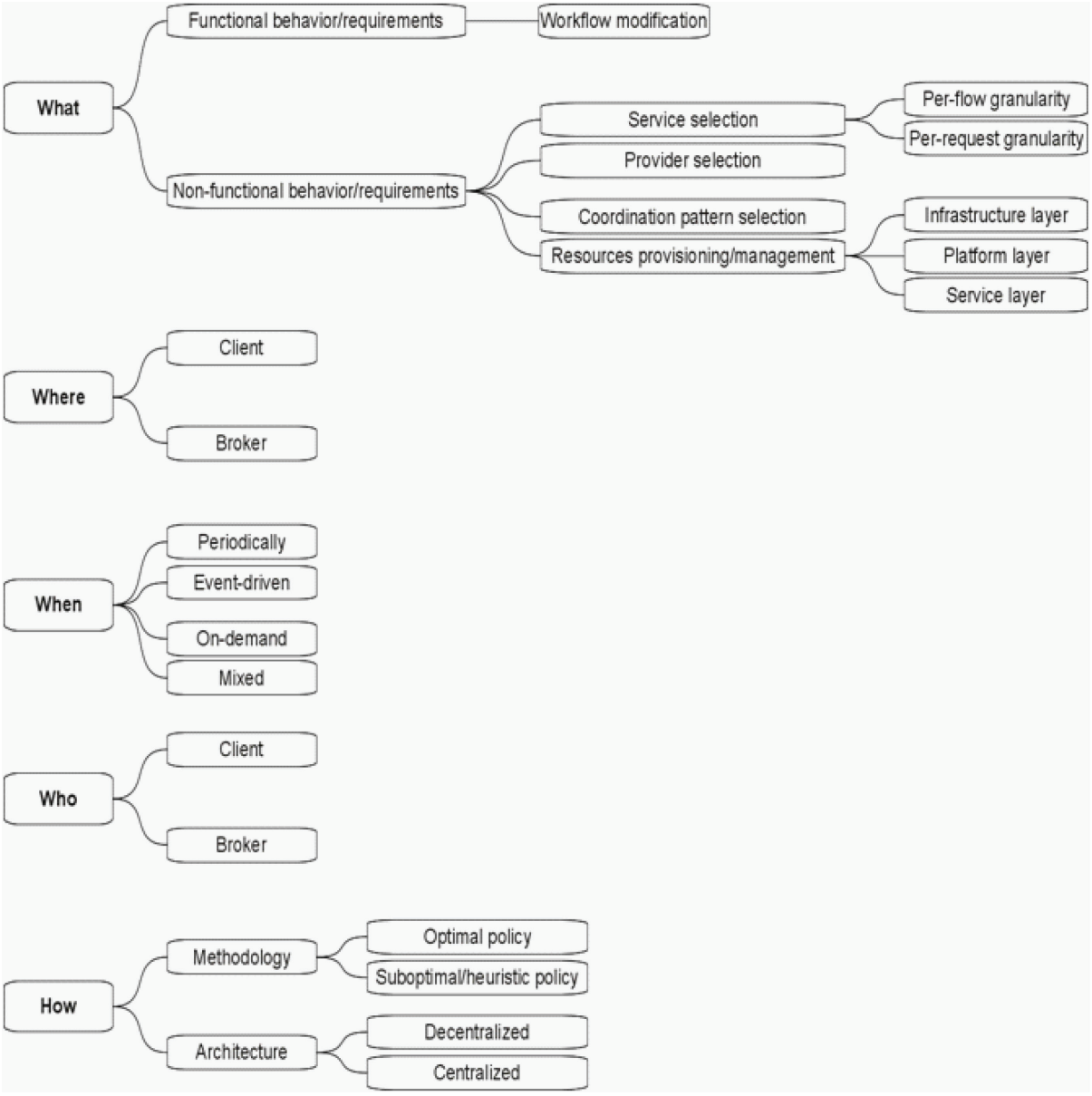
The taxonomy of the Plan phase is shown in Figure 5.

- **What:** The Plan phase is the pivotal phase around which the entire autonomic cycle revolves. The Plan role is to determine and identify the plans and its constituent adaptation 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.

  Different planning methodologies can be applied depending on whether the adaptation cycle considers the functional or the non-functional requirements of the SOA application. When the adaptation concerns the functional behavior, planning the adaptation of the functional behavior means to alter the workflow that defines the business logic of the SOA application. For example, in (Mirandola et al., 2011) the interactions among component services already involved in the workflow can be removed or new ones can be added; furthermore, it is possible to introduce new services with subsequent interactions. Actually, the changes to the workflow are not planned automatically, but the client submits a set of possible solutions to the new functional requirements and the SOS evaluates the QoS of each solution and then chooses the most suitable one with respect to a given utility function.

  The adaptation to satisfy the non-functional requirements has been widely investigated in the last years. In most frameworks, the adaptive management is typically achieved by selecting at runtime the implementation corresponding to each functionality of the abstract composition from a set of candidates and leaving unchanged the composition logic. The overall methodology entails the discovery, identification, and selection of the actual services implementing the SOA application as to satisfy some non-functional requirements while optimizing a suitable utility function.

Figure 5. Plan taxonomy



The service selection can be performed at two different granularity levels. 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 specific request (e.g., minimize the cost of the SOA application), 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 the QoS requirements concerning the global properties of that flow, e.g., to minimize its average response time. Some proposals in the per-request case include (Ardagna et al., 2007;

Ardagna et al., 2011; Canfora, Di Penta, Esposito, Villani, 2008), while (Cardellini, Casalicchio, Grassi & Lo Presti, 2007; Klein, Ishikawa & Honiden, 2010; Ardagna et al., 2010) adopt the per-flow approach. Some frameworks (Bellucci et al., 2010; Menascé et al., 2011) also consider the coordination pattern service selection. For each functionality in the SOA application workflow, these frameworks select a subset of actual services implementing it and a coordination pattern according to which those services are invoked, for example to improve the reliability of the SOA application. Examples of coordination patterns include the parallel invocation 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.

The Plan activity can also entail the selection of the service providers with which bargaining a SLA. The provider selection can be done, for example, to define the set of semantically equivalent services that will serve as candidates for the service selection.

Other approaches plan the provisioning of the manageable resources, e.g., (Calinescu et al., 2011; Mirandola et al., 2011) to adjust the system resources allocated to individual services, for example with the aim to sustain the submitted workload. This approach is feasible only for those resources that are internally managed by the provider of the SOA application, but not for those services offered by external providers.

- **Where:** The Plan phase is usually executed on the broker, and this is the solution adopted in almost all of the frameworks we consider. However, it is also possible to execute the planning on the client, like in (Rouvoy et al., 2009), in case of a brokerless architecture.
- **When:** Similarly to the Analyze phase, the Plan execution is determined by the trade-off between the need to react to significant events, as the arrival or departure of clients or the SLA violations by a service, and the execution time of the adaptation strategy. Planning can be either carried out at fixed time intervals or executed whenever the changes in the environment 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 step. Finally, we can have on-demand planning, which is directly requested by a client depending on its own policies and current perception of the quality attributes of the SOA application.
- **Who:** The entities interested in the Plan phase are the same that perform the Analyze step, that is, the clients and the broker. A client can plan the adaptations actions when it does not rely on an external service broker, while a service broker performs the Plan phase to keep the adaptation decisions under its control.
- **How:** The Plan execution can be accomplished using two different methodologies aimed at computing an optimal or a sub-optimal/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 (LP) as in (Cardellini et al., 2007; Klein et al., 2010), Integer Programming (IP) as in (Alrifai & Risse, 2009), or even Mixed Integer Linear Programming (MILP) as in (Ardagna et al., 2007). To overcome the computational complexity of optimal strategies, especially of integer formulations, the latter type of methodologies rely on heuristics that lead to suboptimal solutions but are faster to solve (Menascé et al.,

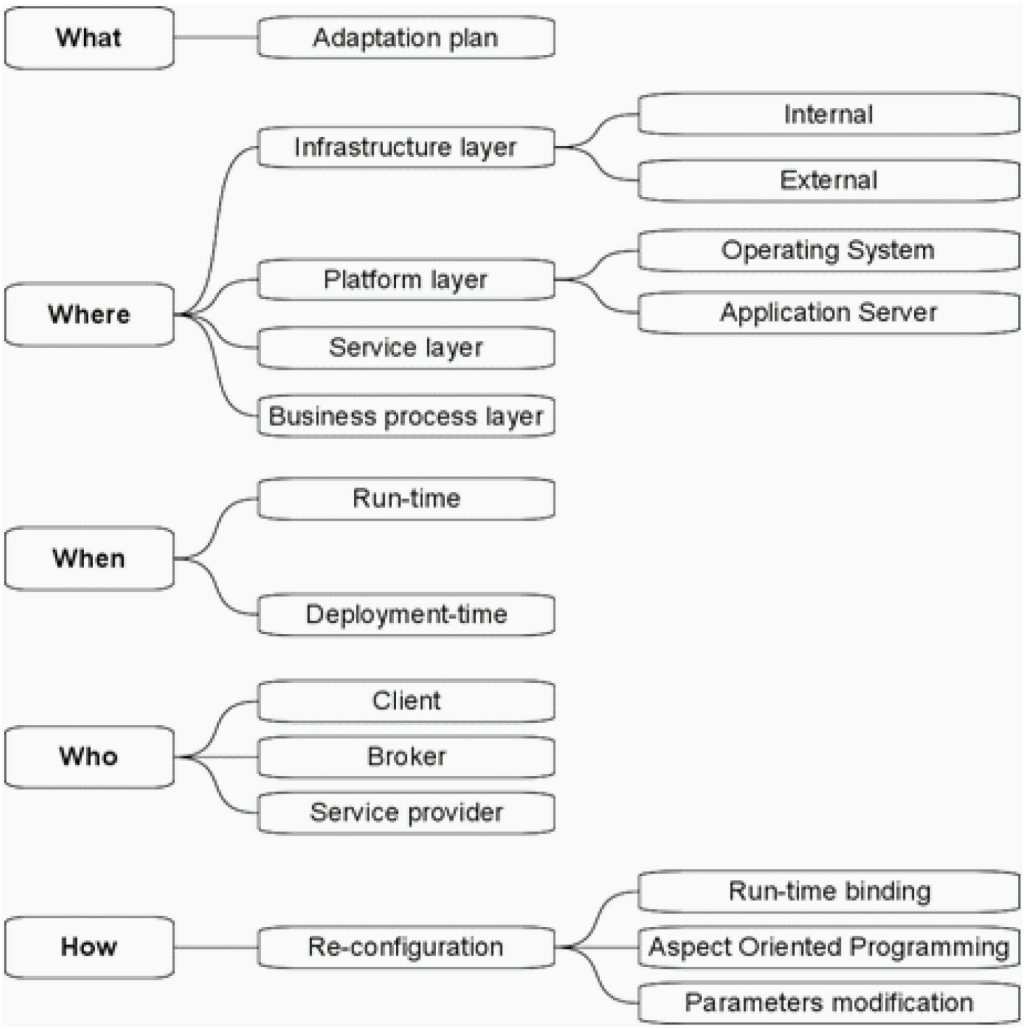
2011). As regards the planner architecture, it is centralized in most of the frameworks, although some decentralized approach exists, as in (Alrifai et al., 2009), where part of the computation is distributed across the network.

Execute Taxonomy

Figure 6 shows the taxonomy of the Execute phase of the MAPE-K loop.

- **What.** In this case, the question assumes a trivial meaning: what we are going to execute coincides with what we have planned in the previous step.
- **Where.** The adaptation plan can be executed at different layers, ranging from the highest business process layer to the lowest infrastructure layer. Starting from the latter, the adaptation actions can be run either on the internal infrastructure (Mirandola et al., 2011; Calinescu et al., 2011) 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. The actions available at the infrastructure layer include adding or removing physical or virtual machines, improving the network connections or the storage system.
  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 (Calinescu et al., 2011). 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 (Bellucci et al., 2010; Menascè et al., 2011; Calinescu et al., 2011).
- **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 either at runtime or 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 solutions 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 runtime case.
- **Who.** The entities involved in the actuation of the 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 can either apply its own computed adaptation plan or rely on some adaptation plan directly provided by the client, as in (Mirandola et al., 2011). Finally, the service provider can modify its behavior according to directives provided by the client or the broker. For example, it can receive an adaptation request issued by a broker that has detected a slowdown in the provider performance.

Figure 6. Execute taxonomy



- **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 execute the adaptation plan: runtime binding, Aspect Oriented Programming (AOP), and parameters modification.
  The runtime binding is the most leveraged approach, as it provides the SOA application with the ability to bind at runtime 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 runtime (Leitner et al., 2010) 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 runtime), *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.

Self-Adaptive Frameworks

As described in the previous subsections, each phase in the MAPE-K loop can be realized in several different ways. However, to design a consistent MAPE-K loop only a subset of the possible combinations is reasonable. For example, if the monitored data are analyzed on an event-driven basis, it is not appropriate to periodically execute the Plan phase. If a service broker monitors its hardware and software resources, it is not possible to plan a service selection for a service composition, unless the used Web services are all in-house, but it is an unreasonable scenario for a SOA application.

When we described the taxonomies of the MAPE loop phases, we referred to some existing frameworks for the self-adaptation of a SOA application. In this section, we analyze the overall mapping of these frameworks on those taxonomies. Specifically, we consider (Calinescu et al., 2011; Ardagna et al., 2011) among the cited frameworks, because they are the most documented; later in the chapter, we will analyze as a case study the MOSES framework (Cardellini et al., in press). In the remainder of this subsection we do not mention the *who* branch of the taxonomies, because it coincides with the service broker for both the frameworks.

Let us start with the Monitor phase. QoSMOS, which stands for QoS Management and Optimization of Service-based systems (Calinescu et al., 2011), focuses on monitoring (*what*) the QoS attributes at the client side, the workload submitted to each service in the service composition, and the resources allocated to the in-house services. The monitoring is executed (*where*) at the broker side, (*when*) on a continuous basis, and (*how*) using a passive methodology.

Discorso, which stands for Distributed Information Systems for Coordinated Service-Oriented Interoperability (Ardagna et al., 2011), differs from QoSMOS only for the what branch of the monitor taxonomy, since it only monitors the QoS attributes at the client side. As discussed below, this slight difference in the Monitor phase affects the design of both the Plan and Execution phases of the MAPE-K loop for the two frameworks.

As regards the Analyze phase, both the frameworks perform (*what*) the analysis of the monitored data (*where*) at the broker side, (*how*) using an online methodology. The difference is in the timeliness: QoSMOS realize a reactive analysis, while Discorso a proactive one. However, this difference does not affect the design of the Plan and Execution phases, but only how the adaptation need is detected. Furthermore, the analysis is performed (*when*) both periodically and event-based for QoSMOS, and only event-based for Discorso.

The design of the Plan phase is affected by the differences in the monitoring. Although both the frameworks perform a non-functional adaptation, Discorso only plans (*what*) the service selection at the per-request granularity, while QoSMOS also the resource provisioning both at infrastructure and platform layers (this difference reflects the different kind of attributes that are monitored). The methodology used by the planning (*how*) is based on optimization models for both the frameworks and the computation is performed using a centralized architecture. In particular, Discorso uses an optimization problem formulated as MILP while QoSMOS an exhaustive research based on a Markovian model of the SOA application.

Eventually, the planning is executed (*where*) at the broker side and (*when*) with the same timeliness of the Analyze phase. Furthermore, QoSMOS performs an iteration of the MAPE-K loop also if a time interval has expired, even if no change is detected in the execution environment. Therefore, the planning phase is executed both periodically and event-based for QoSMOS, and only event-based for Discorso.

The design of the Execute phase is also affected by the design choices made in the previous MAPE-K steps. Indeed, the adaptation actions are executed (*where*) only on the business process layer for the Discorso framework, and also on the infrastructure and platform layers for QoSMOS. These actions are executed (*when*) at runtime, (*how*) using the runtime binding in both frameworks and the parameters modification only in QoSMOS.

SERVICE SELECTION

As previously observed, the Plan phase is the core of the autonomic control loop as it defines the self-adaptation logic. It is no surprise then that many research efforts on adaptive management of SOSs have focused on studying and developing planning strategies.

In the context of SOA applications, the most critical tasks of the planning phase have been identified with the ability to manage, control, and predict the QoS characteristics of the offered SOA applications (Papazoglou, Traverso, Dustdar & Leymann, 2007). Hence, most planning policies have addressed the issue of fulfilling non-functional requirements. Since SOA applications are built by composing loosely coupled services, which are easily replaceable at runtime with dynamic binding, most of the research efforts have focused on devising proper service selection and coordination pattern selection strategies.

In this section we review service selection strategies for SOA applications. In this respect, we can clearly distinguish two broad classes of approaches, depending on whether we deal with the per-request or the per-flow granularity. Indeed, despite addressing similar issues, the two approaches significantly differ in the formulation of the optimization problem. In the first case, we have to deal with 0-1 problems, which are computationally complex, while in the second case we deal with probabilities, which lead to cheaper computations. We will focus on two representative solutions: (Ardagna et al., 2007) for the per-request approach and (Cardellini et al., 2007) for the per-flow approach. Since the optimal strategies for the per-request granularity are computationally expensive, many research efforts have focused on heuristics, which, albeit suboptimal, are computationally efficient; therefore, we will also review some representative examples in the next section.

In the following, we consider a broker that offers a SOA application P. We assume that the broker has negotiated SLAs with its clients and has the main task to fulfill these SLAs, while optimizing a suitable utility function and being constrained by the SLAs it has stipulated as a client with the providers of the services involved in the service composition. Depending on the utilization scenario, the utility function can optimize specific QoS attributes for different clients/service classes, e.g., minimizing the average response time, and/or the broker own utility, e.g., minimizing the overall cost paid by the broker to offer the SOA application. These different, and possibly conflicting, optimization goals can lead to a multi-objective optimization problem. This is usually tackled, e.g., (Ardagna et al., 2007; Cardellini et al., 2007), by considering a single objective function obtained by applying the Simple Additive Weighting (SAW) technique (Hwang & Yoon, 1981), which is the most widely used scalarization method. Following the SAW technique, the utility function can be defined as the weighted sum of the (normalized) QoS attributes.

We denote by S the set of abstract tasks belonging to the service composition P, where $S_i \in S$, $i = 1, \ldots, m$, represents a single task, being m the number of tasks composing P. A task is a functionality required by the SOA application and implemented by a set of services available in the marketplace. For each task S_i, we assume that the broker has identified a pool $I_i = \{cs_{ij}\}$ of concrete services implementing it. Figure 7a shows an example of workflow for a SOA application.

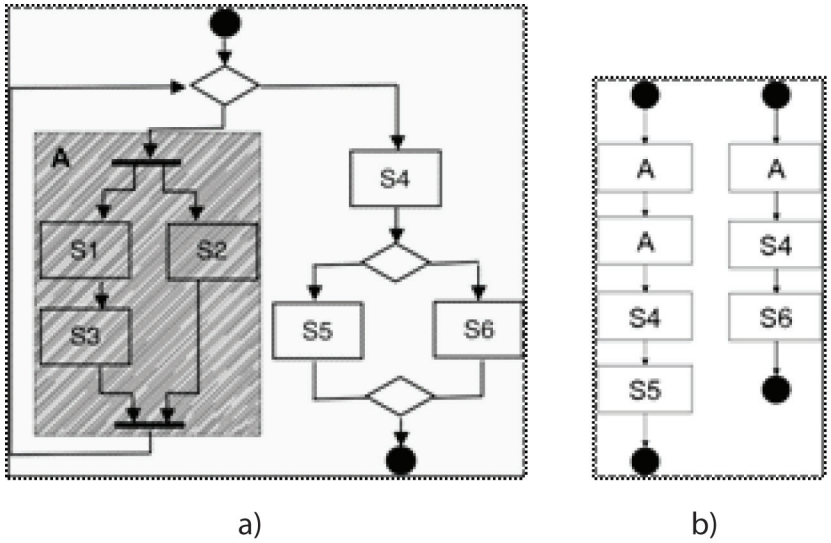
Per-Request Granularity

We first consider the per-request approach in (Ardagna et al., 2007). Let us focus, without loss of generality, on SLAs containing QoS constraints that refer to the following three attributes: (i) response time, defined as the interval of time elapsed from the service invocation to its completion; (ii) reliability, that is, the probability that the service completes its task when invoked; (iii) cost, which is the price charged for the service invocation. Furthermore, let us assume that this QoS model holds for SLAs stipulated by the broker with both its clients and service providers. In the per-request approach, the broker tries to meet the QoS constraints specified in the SLA for each request, irrespective of whether it belongs to some flow generated by one or more clients.

The optimal service selection problem is then formulated as MILP problem. We denote with the vector $x = [x_1, \ldots, x_m]$ the optimal policy for a request to the SOA application, where each entry $x_i = [x_{ij}]$, $x_{ij} \in \{0, 1\}$, $i \in S$, $j \in I_i$, denotes the adaptation policy for task S_i and the constraint $\sum_{j \in Ii} x_{ij} = 1$ holds. That is, x_{ij} is the decision variable, which is equal to 1 if task S_i is implemented by service cs_{ij}, 0 otherwise. As an example, suppose that for the task S_i the broker has individuated 4 concrete services implementing it, namely cs_1, cs_2, cs_3 and cs_4. Assume that the per-request policy x determines that for a given request $x_i = [0, 0, 1, 0]$. It means that, according to this policy, for S_i the broker binds the request to cs_{i3}.

The fulfillment of the QoS constraints on a per-request basis means that the broker needs to take into account all the possible scenarios that might occur during the execution of the SOA application. To this end, the optimal strategy needs to consider all the possible execution paths that might

Figure 7. a) An example of workflow; b) Two different execution paths



arise from the workflow of the SOA application (Ardagna et al., 2007). An execution path ep_n is a multiset of tasks $ep_n = \{S_1, S_2, \ldots, S_l\} \subseteq S$, such that S_1 and S_l are respectively the initial and final tasks of the path and no pair $S_i, S_j \in ep_n$ belongs to alternative branches. An execution path does not contain any loop, because the loops are peeled, but it may contain parallel sequences. Loop peeling involves rewriting the loop as a sequence of branch conditions (the branch conditions that arise from loop peeling produce other execution paths, see the example in Figure 7b). In other words, the set of all the execution paths represents all the possible execution scenarios of a workflow.

Figure 8 shows a simplified version of the problem formulation for the per-request optimization, where x denotes the optimal service selection policy and $U(x)$ the broker utility function. We indicate with T_{max}, R_{min} and C_{max}, respectively the maximum response time, the minimum reliability, and the maximum cost that are allowed, i.e., the QoS constraints specified in the SLAs. On the other hand, $T_n(x)$, $R_n(x)$, and $C_n(x)$ denote the response time, reliability, and cost of the execution path ep_n under the selection policy x.

Per-Flow Granularity

In the per-flow approach, the client requests are considered at the flow granularity level. In this setting, the SLAs and the service selection concern the QoS and the behavior of the aggregated flow of requests generated by the clients. As a consequence, the constraints stated in the SLA do not make any provision on the QoS of each single request, but rather the SLA is concerned with the average value of the QoS attributes computed over the flow of requests generated by a given client.

To account for the existence of multiple concurrent requests made by the different clients, the per-flow approach in (Cardellini et al., 2007) requires to negotiate in the SLA the additional parameter L, which represents a bound on the amount of requests per unit of time a client can generate.

It is also assumed that there is a set K of service classes, with $k \in K$, for each service composition. Therefore, a client bargains its SLA with the broker referring to one of these service classes. Although this could seem a limitation, it actually is not, because the granularity level of the service classes may be arbitrarily fine and, at the finest level, each client could have its own service class.

The optimal service selection problem is then formulated as a LP problem, that is computationally lighter to solve than the MILP formulation of the per-request approach. For each class k, we denote with the vector $x^k = [x^k_1, \ldots, x^k_m]$ the optimal policy, where each entry $x^k_i = [x^k_{ij}]$, $0 \le x_{ij} \le 1$, $i \in S$, $j \in I_i$, denotes the adaptation policy for task S_i and the constraint $\sum_{j \in Ii} x^k_{ij} = 1$ holds. That is, the policy defines a probabilistic binding between S_i

Figure 8. Per-request optimization problem

$$\begin{aligned} \text{Problem } \textbf{per-request: max } & U(\boldsymbol{x}) \\ \textbf{subject to: } & T_n(\boldsymbol{x}) \le T_{max} \quad \forall ep_n \\ & \log R_n(\boldsymbol{x}) \ge \log R_{min} \quad \forall ep_n \\ & C_n(\boldsymbol{x}) \le C_{max} \quad \forall ep_n \\ x_{ij} \in \{0,1\} \ \forall j \in \Im_i, & \sum_{j \in \Im_i} x_{ij} = 1 \quad \forall i \in \mathcal{S} \end{aligned}$$

and its implementation in I_i, whereby each entry x^k_{ij} of x^k_i denotes the probability that the class-k request will be bound to concrete service cs_{ij}. As an example, let us suppose that, as in the per-request case, the broker has individuated for the task S_i the same 4 concrete services implementing it, namely cs_1, cs_2, cs_3 and cs_4. Now assume that the per-flow service selection, for a given class k, determines $x^k = [0, 0.2, 0.5, 0.3]$. It means that, for a class-k request for S_i, the broker will bind cs_2 with probability 0.2, cs_3 with probability 0.5 and cs_4 with probability 0.3.

Figure 9 shows a simplified version of the optimization problem formulation. We indicate with T^k_{max}, R^k_{min} and C^k_{max}, respectively the maximum average response time, the minimum average reliability, and the maximum average cost, that correspond to the QoS constraints specified in the class-k SLA. $T^k(L, x)$, $R^k(L, x)$, and $C^k(L, x)$ are respectively the class-k response time, reliability, and cost, respectively, under the adaptation policy $x = [x^k]$ $k \in K$. Their expressions require the knowledge of V^k_i for each task S_i, that is the average number of times S_i is invoked by a class-k request. In particular, the second-last equation, where $L^k = \sum_u L^k_u$, is the aggregated service request rate of class-k clients (being u a client), ensures that the concrete services used in the SOA application will not be overloaded by the client requests, that is, the client requests will not exceed the volume of invocations l_{ij} agreed with each service provider.

A Brief Comparison between Per-Request and Per-Flow Granularity

The difference between the per-flow and the per-request approaches lies in the service selection policy: in the latter each task is bound to one and only one concrete service, while in the former each task is bound to a set of concrete services and at runtime one of them is probabilistically chosen. As a result, different concrete services can be used for implementing the same task in different executions of the service composition while the same adaptation plan holds. On the other hand, in the per-request approach the same concrete service is used for all similar requests until the same adaptation decision holds.

The study in (Cardellini, Di Valerio, Grassi, Iannucci & Lo Presti, 2011a) presents an experimental comparison between the two approaches, focusing on their impact on the SOS performance in term of service composition's response time. The results show that under a light request load the two approaches perform almost the same, but under a high request load the per-request approach

Figure 9. Per-flow optimization problem

$$\begin{aligned}
&\text{Problem } \textbf{per-flow: max } U(\boldsymbol{x}) \\
&\quad \textbf{subject to: } T^k(L,\boldsymbol{x}) \le T^k_{max} \quad \forall k \in K \\
&\quad \log R^k(L,\boldsymbol{x}) \ge \log R^k_{min} \quad \forall k \in K \\
&\quad C^k(L,\boldsymbol{x}) \le C^k_{max} \quad \forall k \in K \\
&\quad \sum_{k \in K} x^k_{ij} V^k_{\alpha,i} L^k \le l_{ij} \quad \forall j \in \Im_i, \forall i \in \mathcal{S} \\
&\quad x^k_{ij} \ge 0 \quad \forall j \in \Im_i, \sum_{j \in \Im_i} x^k_{ij} = 1 \quad \forall i \in \mathcal{S}
\end{aligned}$$

exhibits scalability problems, while the per-flow approach performs much better. The motivation is as follows: in the per-request approach, all requests to a given task are resolved using the same concrete service until the same service selection solution holds. This works under light loads, but at higher loads the service capacity is eventually saturated and performance degrades. On the other hand, in the per-flow approach, the load is shared among multiple concrete services thanks to the probabilistic service selection without saturating any service thanks to the load constraints which prevent the services' overloading.

However, the main disadvantage of the per-flow approach is that the QoS levels are guaranteed on average for the overall flow; therefore, the performance of a single request is actually unpredictable. In (Cardellini, Di Valerio, Grassi, Iannucci & Lo Presti, 2011b) the interested reader can find a new service selection policy that combines the benefits of both approaches, i.e., the per-request guarantees and the per-flow probabilistic service selection, thus ensuring load balancing and overcoming the per-request scalability issues.

HEURISTICS

The high computational complexity of the optimal per-request 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 (Martello & Toth, 1987). 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 (Berbner et al., 2006; Klein, Ishikawa, & Honiden, 2010) 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 (Berbner et al., 2006) 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 (Klein et al, 2010) 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 (Alrifai et al., 2009). The authors first decompose each global QoS constraint into a set of *m* 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 (Alrifai, Skoutas, & Risse, 2010), where the authors propose a different method for QoS level discretization: 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 (Canfora, Di Penta, Esposito, Villani, 2008), 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 *m* genes and every gene represents a particular instance of concrete services. A different objective is pursued by (Wada et al., in press), 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 (Yu, Zhang & Lin, 2007) 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) (Cormen, Leiserson, Rivest, & Stein, 2001): 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%.

CASE STUDY: MOSES

As a representative case study of SOS that adaptively manages a SOA application adopting the MAPE-K model, we focus on MOSES (MOdel-based SElf-adaptation of SOA systems), which is a framework for the QoS-driven adaptation of a service-oriented system. Although MOSES is a particular instance of SOS, it is a fully functional prototype with a highly modular architecture that allows you to easily realize other solutions proposed in literature by replacing and/or adding a given component with another possible implementation realizing a different approach. For instance, the Plan phase in the original MOSES follows the per-flow approach, but we also implemented the per-request approach in (Ardagna et al., 2007) by replacing some MOSES components in order to perform the comparison presented in (Cardellini et al., 2011a). We will use the MOSES prototype to validate the benefits of a SOS with self-adaptive features under a motivating scenario characterized by a varying operating environment, where component services appear and disappear.

For a comprehensive description of the methodology underpinning MOSES and the software tool that implements it, we refer the reader to (Cardellini et al., in press) and (Bellucci et al., 2010), respectively.

MOSES Architecture

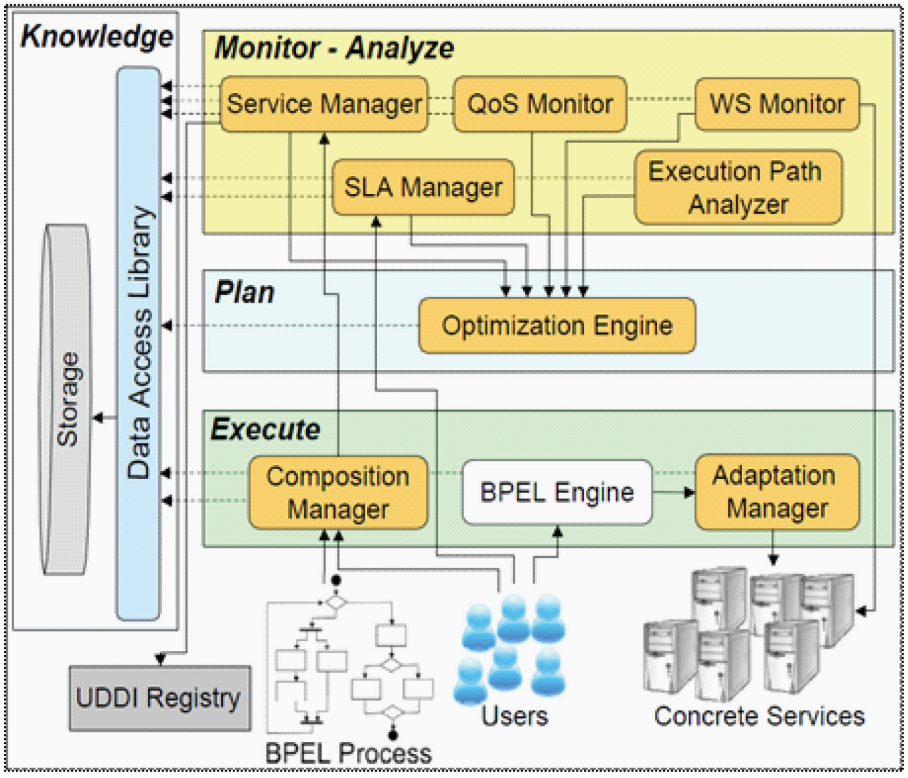
The MOSES architecture represents an instantiation for the SOA environment of a self-adaptive software system, organized according to the MAPE-K loop and focused on the fulfillment of QoS requirements.

Figure 10 shows how the MOSES components are organized according to the MAPE-K control loop.

Monitor–Analyze Phases

The Monitor-Analyze subsystem comprises all those 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* is in charge of measuring and analyzing the QoS attributes of the concrete services used by MOSES to provide the SOA application. The *WS Monitor* periodically checks the availability of the concrete services. The *Execution Path Analyzer* is in charge of monitoring the variations of the usage profile. In case of the service selection at the per-flow granularity, it computes and updates for each abstract task S_i the expected number of times V^k_i that S_i is invoked by service class k. With respect to the Monitor taxonomy in Figure 3, these MOSES components monitor: (*what*) client-side, QoS attributes of services; (*where*) broker side; (*when*) on a continuous time basis; (*how*) using both active and passive methodologies in a centralized architecture. In particular, the *QoS*

Figure 10. MOSES architecture



Monitor and the *Execution Path Analyzer* use a passive monitoring methodology by collecting the service invocation results, while the *WS Monitor* actively checks the services availability.

The Monitor-Analyze subsystem also includes the *Service Manager* and the *SLA Manager*, which are involved in the SLA negotiation processes where the broker acts as an intermediary. Indeed, the Service Manager is in charge of negotiating SLAs with the service providers and discovering candidate services offering the functionalities in the service composition, while the SLA Manager is responsible for the SLAs with the MOSES clients. In addition, the latter manages the client profiles, adding and removing them. We included these components in the Monitor-Analyze phases because they can invoke the Plan phase if new SLAs with clients or service providers are either stipulated or removed. Specifically, for each new SLA request, the SLA Manager performs an admission control to evaluate whether there are enough available resources to accept the incoming client, given the associated SLA and without violating already existing SLAs with other clients.

In MOSES the Analyze phase can be classified as follows on the basis of the taxonomy in Figure 4: (*where*) at the broker-side; (*when*) MOSES adopt all the approaches in Figure 4: the QoS Monitor analyzes periodically the QoS attributes of the concrete services, checking whether their measured values correspond to the stipulated one; the SLA Manager performs an on-demand analysis for client arrivals and departures, while the Service Manager and the WS Monitor adopt an event-driven analysis for discovering services and checking their availability, respectively; (*how*) the analysis is performed online with either proactive or reactive policies using a centralized architecture. In particular, the analysis is reactive for all

the components except the QoS Monitor that can also use a proactive methodology.

Planning Phase

The planning phase is fully executed by the *Optimization Engine*, whose task is to solve the service selection optimization problem. The latter is built from a model of the SOA application workflow, instantiated by the *Composition Manager* and whose parameters are initialized with the values in the SLAs contracted with the clients and the service providers. This model is kept up-to-date at runtime by the monitoring components. For example, the values of the QoS attributes of each concrete service used in the problem can be updated at runtime with the actual measured values and the same holds for the average number of invocations to each task. With respect to the Plan taxonomy, MOSES can be classified as follows: (*what*) non-functional requirements, in particular the service selection and the coordination pattern selection. For sake of simplicity, we previously presented only the service selection problem; to account for the coordination pattern selection, the problem formulation is slightly more complicated as described in (Cardellini et al., in press). Eventually, (*where*) the Plan phase is executed at the broker on an event-driven basis, i.e., when the components in the Monitor–Analyze phases detect a relevant event to be addressed, and (*how*) the methodology adopted to plan the adaptation actions determines the service selection and pattern coordination relying on a centralized architecture.

Execution Phase

The execution phase is carried out by the *Composition Manager*, the *BPEL Engine*, and the *Adaptation Manager*. The *Composition Manager*, given a new service composition to be deployed as a BPEL process (OASIS, 2007), builds the workflow model that the *Optimization Engine* will use in the Plan phase. Furthermore, it modifies the workflow in such a way that all the service invocations are translated into invocations of the *Adaptation Manager*. The latter acts as a proxy that, given the name of an abstract task, invokes the service(s) implementing that task according to the service selection (and the coordination pattern selection) policy computed by the *Optimization Engine*. In turn, the *BPEL Engine* executes the workflow logic and is the front-end component to the client requests. The BPEL Engine and Adaptation Manager represent the core of the MOSES execution and runtime adaptation of the SOA application. Following the Execute taxonomy, MOSES: (*when*) executes the planned actions at runtime, (*where*) acting at the workflow level, and (*how*) using the dynamic binding mechanism.

MOSES Design

The MOSES architecture has been designed on the basis of the Java Business Integration (JBI) specification. 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: (i) the Service Engines (SEs) that enable pluggable business logic; (ii) the Binding Components (BCs) that enable pluggable external connectivity; (iii) the Normalized Message Router (NMR), which directs normalized messages from source to destination components according to specified policies.

As a JBI implementation, MOSES has been implemented within the open-source project *OpenESB* (ESB stands for Enterprise Service Bus), because it is an implementation and extension of the JBI standard. It implements JBI because it provides binding components, service engines, and the NMR; it 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. The GlassFish application server is the default runtime environment, although OpenESB can be integrated in several JEE application servers.

Figure 11 shows how the MOSES components are placed with respect to the JBI architecture: most of the components are executed by the JEE Service Engine, while the business process is executed by the BPEL Engine. The NMR works as a glue between the Service Engines and the Binding Components, having the ability to route messages between these sets of components.

The MOSES architecture is enriched by MDAL, which stands for MOSES Data Access Library. This library allows us to simplify the usage of the underlying Database layer by abstracting low-level queries with high-level methods.

Figure 12 shows the typical scenario in which a client issues a SOA request to MOSES. In the first step, the SOAP request is directed to the HTTP binding component. The received request is then forwarded to the NMR, which in turn routes it to the proper Service Engine, i.e., the BPEL Service Engine. The latter is in charge of executing the required business process, after having performed some client authentication tasks, with the help of the Adaptation Manager when external invocations are needed.

The Adaptation Manager, differently from the other MOSES components, is not implemented as a JBI Service Unit. It is rather implemented as a standard Java class belonging to the application server classpath, and thus it is accessible by any application served by the application server itself. In particular, each invoke activity in the BPEL process, which should be executed by the BPEL Engine, is replaced by a call to the Adaptation Manager's entry method, whose tasks are: to read the most up-to-date service selection plan from the Database using the MDAL library, to invoke the concrete Web service(s), and finally to forward the Web service response to the BPEL Engine.

Once the business process ends its execution, the client response is put on the NMR, which in turn routes it to the HTTP BC, which finally delivers the message to the client.

Figure 11. MOSES architecture and the JBI environment

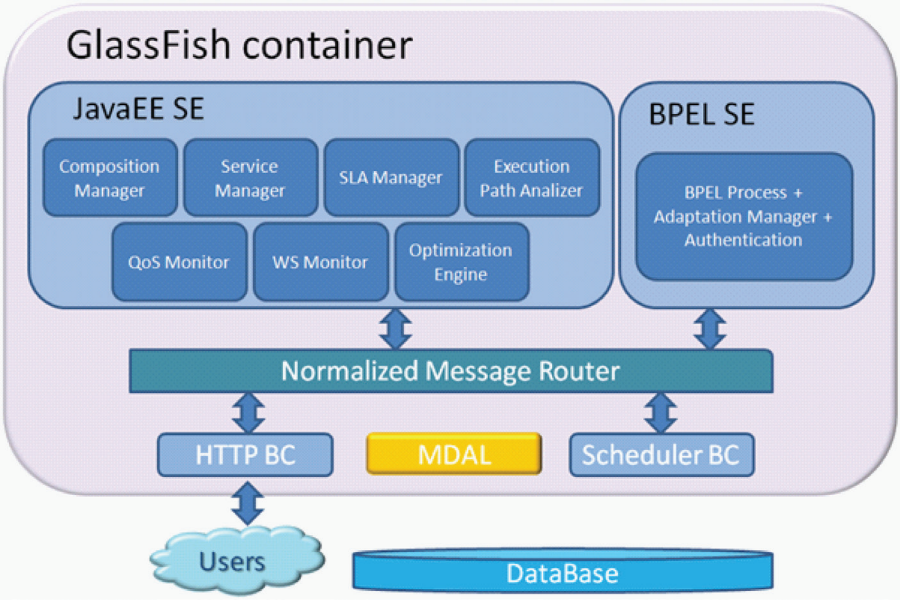
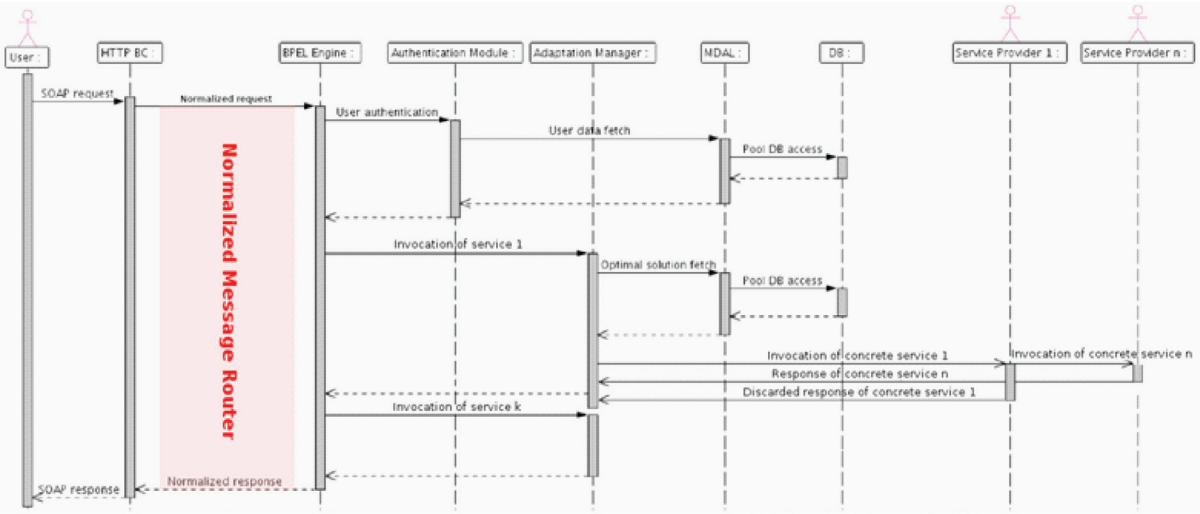


Figure 12. MOSES request-response cycle



Experimental Results

In this section we illustrate the effectiveness of the adaptive management of service composition by analyzing a set of experimental results obtained with the MOSES prototype in controlled experiments. Specifically, we will study 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 sets of experiments, the utility function of the service broker is to maximize the reliability of the SOA application according to the per-flow optimization problem sketched in Figure 9.

The first set of experiments 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 sets of experiments. In this first set, 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 set of experiments 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 $\{p_{running}, p_{failed}\}=\{0.95, 0.05\}$, 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 set of experiments 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 set of experiments 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 set of experiments, the same service selection policy holds for the whole experiment.

Finally, in the fourth set of experiments, 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 what 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.

Experimental Setup

For all the sets of experiments, the testing environment consists of 3 Intel Xeon quadcore servers (2 Ghz/core) with 8 GB RAM each (nodes 1, 2, and 3), and 1 KVM virtual machine with 1 CPU and 1 GB RAM (node 4); a Gb Ethernet connects all the machines. The MOSES prototype is deployed as follows: node 1 hosts all the components of the Execute subsystem, node 2 the storage layer together with the candidate concrete services, and node 3 the components in the Monitor+Analyze and Plan subsystems. Finally, node 4 hosts the workload generator.

We consider the SOA application defined by the workflow in Figure 13, composed of 6 stateless tasks, and assume that 10 concrete services (with their respective SLAs) have been identified for abstract tasks S_1 and S_3, while 8 concrete services have been identified for any other task. Their respective SLA parameters, shown in Table

Figure 13. Workflow of the SOA application used in the experiments

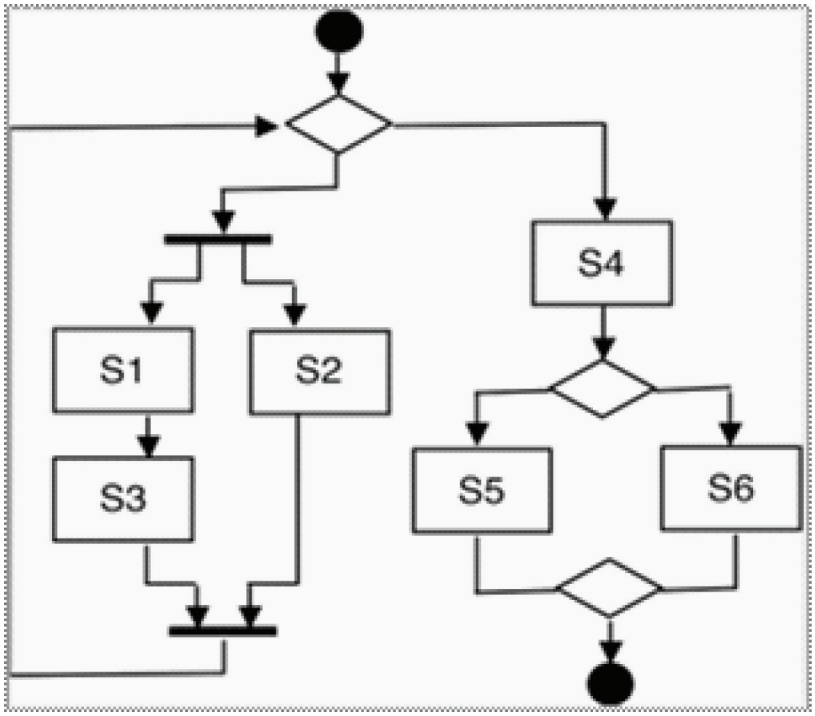


Table 1. SLA parameters for concrete services

cs	t_{ij}	r_{ij}	c_{ij}
$cs_{1[1,6]}$	2	0.995	6
$cs_{1[2,7]}$	1.8	0.99	6
$cs_{1[3,8]}$	2	0.99	5.5
$cs_{1[4,9]}$	3	0.995	4.5
$cs_{1[5,10]}$	4	0.99	3
$cs_{2[1,5]}$	1	0.995	2
$cs_{2[2,6]}$	2	0.995	1.8
$cs_{2[3,7]}$	1.8	0.99	1.8
$cs_{2[4,8]}$	3	0.99	1
$cs_{3[1,6]}$	1	0.995	5
$cs_{3[2,7]}$	1	0.99	4.5
$cs_{3[3,8]}$	2	0.99	4
$cs_{3[4,9]}$	4	0.95	2
$cs_{3[5,10]}$	5	0.95	1

cs	t_{ij}	r_{ij}	c_{ij}
$cs_{4[1,5]}$	0.5	0.995	1
$cs_{4[2,6]}$	0.5	0.99	0.8
$cs_{4[3,7]}$	1	0.995	0.8
$cs_{4[4,8]}$	1	0.95	0.6
$cs_{5[1,5]}$	1	0.995	3
$cs_{5[2,6]}$	2	0.99	2
$cs_{5[3,7]}$	3	0.99	1.5
$cs_{5[4,8]}$	4	0.95	1
$cs_{6[1,5]}$	1.8	0.99	1
$cs_{6[2,6]}$	2	0.995	0.8
$cs_{6[3,7]}$	3	0.99	0.6
$cs_{6[4,8]}$	4	0.95	0.4

1, differ in terms of cost c_{ij}, reliability r_{ij}, and response time t_{ij} (in sec).

We also suppose that MOSES offers to its clients the SLA $\{T_{max}, R_{min}, C_{max}\}=\{7$ 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: $V_1 = V_2 = V_3 = 1.5$, $V_4 = 1$, $V_5 = V_6 = 0.5$.

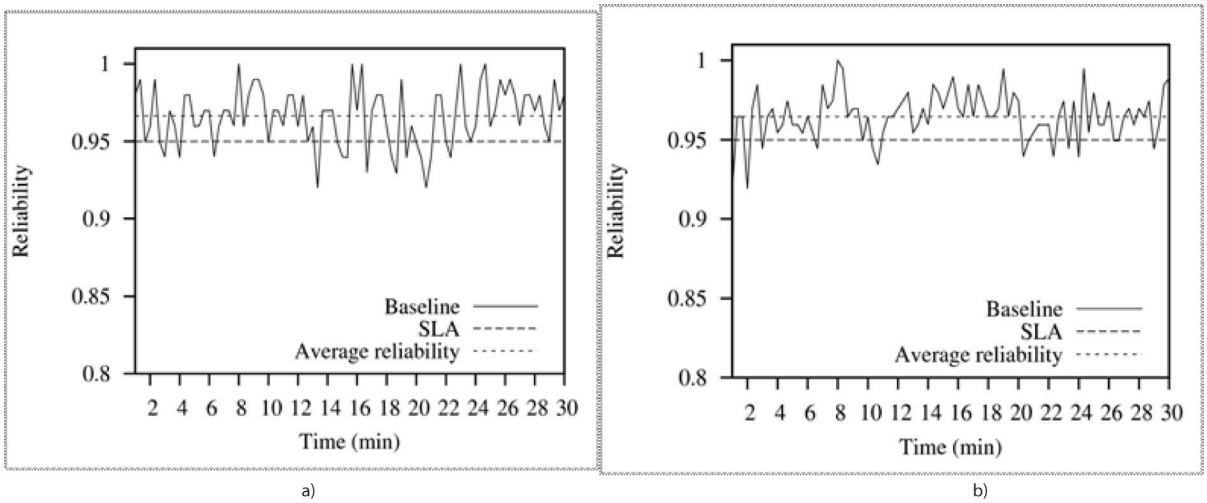
The Adaptation Manager introduces an overhead due to the runtime binding of the task endpoints to their concrete implementations that may affect the response time of the SOA application. In a preliminary test we measured this overhead under the low and high request rates. We found it to be 13.3 ms when MOSES is subject to a low request rate and 20.3 ms for a high request rate. Given the values of the expected number of service invocations above reported, the average number of invoke activities is equal to 6.5. Therefore, the Adaptation Manager introduces a mean per-invocation overhead equal to 2.05 ms when the system is subject to a low request rate and 3.12 ms for the high request rate. A more detailed analysis of the MOSES overhead can be found in (Cardellini et al., in press), where the response time constraint in the optimization problem also accounts for the overhead introduced by MOSES itself in adaptively managing the SOA application.

Experimental Results

We first present the results of the baseline scenario. The Baseline curves in Figures 14a and 14b show how the reliability of the SOA application varies over time, when the 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

Figure 14. a) Baseline reliability over time under low request rate; b) Baseline reliability over time under high request rate



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 2, 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.

In the second set of experiments we let the service providers gracefully fail, thus simulating, for instance, service programmed downtimes. The results in Figures 15a and 15b 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 3 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.

Figures 16a and 16b 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

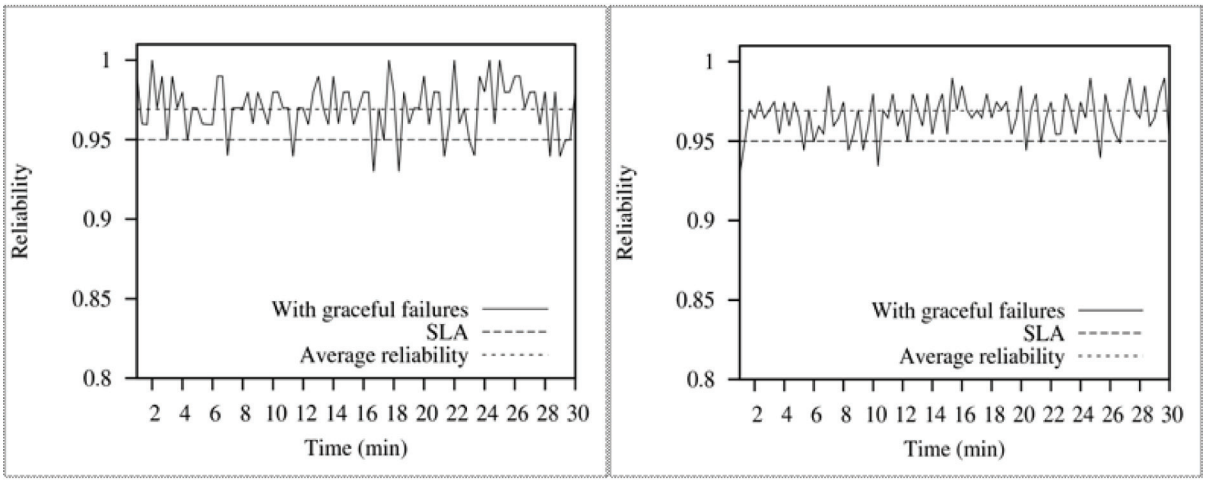
Table 2. Average reliability and 95% confidence interval for the baseline experiment

	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 3. Average reliability and 95% confidence interval for the experiment with graceful failures

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

Figure 15. a) Reliability over time when services are subject to graceful failures under low request rate; b) Reliability over time when services are subject to graceful failures under high request rate



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

Figure 16. a) Reliability over time when services are subject to failures, without WS Monitor under low request rate; b) Reliability over time when services are subject to failures, without WS Monitor under high request rate

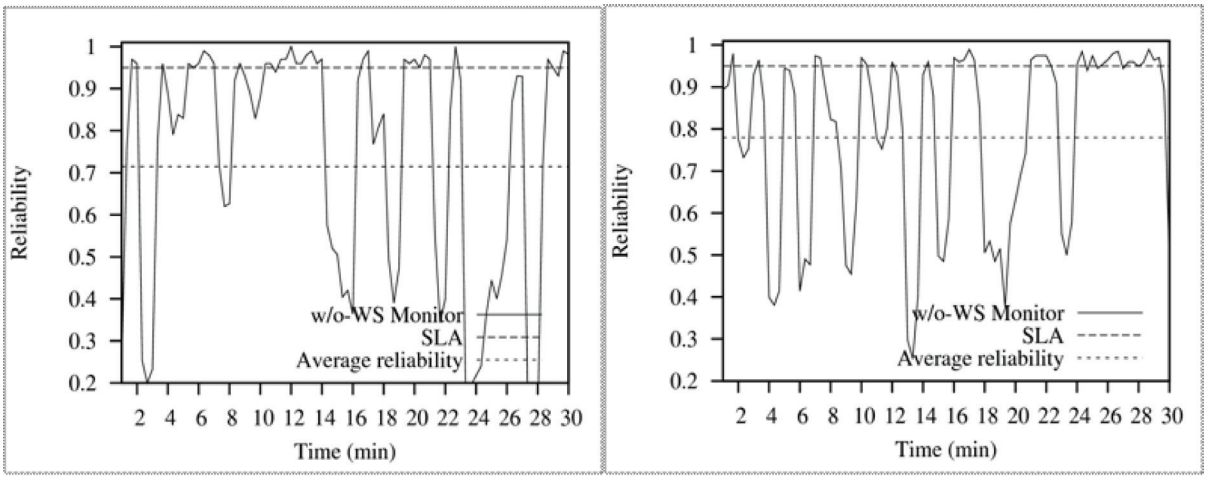


Table 4. Comparison of the average reliability and 95% confidence interval for the experiments with and without the WS monitor

	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

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

The objective of the last set of experiments is to show the improvement achieved thanks to the WS Monitor component.

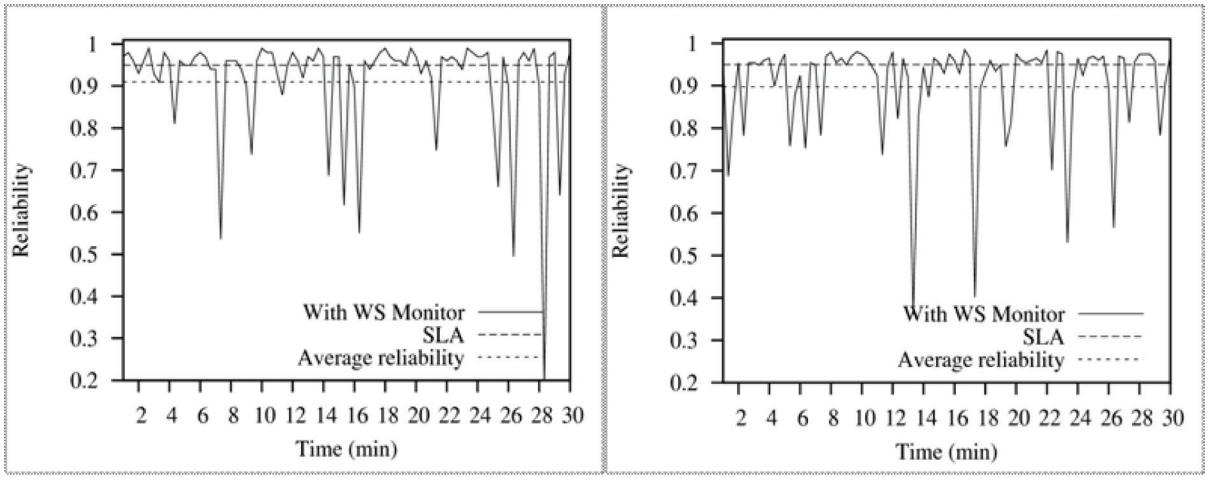
Figures 17a and 17b show how the reliability of the SOA application varies over time when the service providers exhibit the same churn rate of the third experiment without signaling their state to MOSES, but now with the WS Monitor enabled on MOSES.

As shown in Table 4, 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 16a and 16b, when the WS Monitor was disabled.

FUTURE RESEARCH DIRECTIONS

In this section, we briefly discuss some open challenges regarding the design of SOSs for the adaptive management of service composition that

Figure 17. a) Reliability over time when services are subject to failures, with WS Monitor under low request rate; b) Reliability over time when services are subject to failures, with WS Monitor under high request rate



can be explored in future research. Some of these challenges directly stem from our own experience in designing and using the MOSES framework.

Using MOSES for our experimental evaluations, we found that a non–trivial issue is to adequately tune a quite large number of system parameters in the various software tools that we used to implement the MOSES prototype. Designing a self-tunable platform can greatly help the administrator of the service broker. For example, a self-adaptive tuning of the application server parameters according to the actual load of the SOA application can help to improve the resources utilization in the infrastructure layer, therefore allowing to reduce the number of required resources. More generally, future work can address the provisioning and management of the platform and infrastructure layers used by the SOS, also considering cross-cutting issues, for example regarding the SLAs.

Another challenging research issue is the development of decentralized approaches for the Plan phase. The optimization problems that are often used to define the adaptation plan can be computationally intensive applications that need to provide a solution in the shortest time possible, otherwise the service broker can incur in penalties due to the lack of SLA compliance. The centralized approaches that have been so far proposed may suffer from scalability and fault-tolerance issues caused by high volumes of requests.

The design of decentralized solutions can entail not only the Plan phase but also the whole MAPE-K loop. For example, in case of a single organization offering QoS-aware SOA applications, the self-adaptive SOS can be designed as a decentralized system consisting of a set of federated SOSs that can coordinate themselves according to a master-slave scheme. In case of multiple organizations, more complex solutions need to be devised: under the hypothesis of federated cooperating SOSs, distributing the whole MAPE-K loop among multiple SOSs requires to devise a distributed solution of the overall optimization problem. Analyzing the current literature, we noted that the case of several self-adaptive service-oriented systems under cooperating or non-cooperating scenarios is not yet satisfactorily covered and we believe that investigating how to cope with these issues is a timely and promising research indication.

CONCLUSION

The development of distributed applications has recently shifted from the classic in-house development to activities concerning the identification, selection, and composition of services offered by third party providers through a service marketplace and this shift is rapidly accelerating with the advent of Cloud computing. This new model, which is the basis of the SOA paradigm, increases the interoperability level of the applications, by forcing them to only use standard protocols for any activity. However, when QoS matters, SOA applications might suffer from their distributed nature because the QoS levels offered by service providers may quickly fluctuate over time, due to the highly varying execution environment. On the other hand, the dynamic composition of SOA applications can provide a solution to govern providers' QoS fluctuations by choosing at runtime which providers to use under certain conditions. Such a control process is often implemented by an external application governing the SOA application itself. There exist various approaches for realizing the control process, but their common reference model is the MAPE-K control loop.

In this chapter we analyzed how the MAPE-K reference model has been applied to design self-adaptive SOS; for every MAPE-K phase we presented a taxonomy organized according to the five Ws and one H concept that clarifies the many dimensions and options which are available when designing a self-adaptive SOS. Although every phase is required in the realization of a control process, we focused on the Plan phase because it

is the core of the adaptation process; specifically, we analyzed how the SOS can self-adapt in order to satisfy some non-functional requirements. Most approaches in this research line address the adaptation by selecting the appropriate services that can be exploited during the SOA application execution or by properly managing the resource provisioning in such a way to meet the target QoS levels.

As a case study, we presented the MOSES framework, a fully functional prototype that realizes every phase of the MAPE-K model relying on a modular system architecture. We demonstrated how it is possible to improve the QoS of a SOA application that operates in a highly varying execution environment, where component services continuously appear and disappear. The experimental results showed that the execution of a SOA application managed by MOSES allows us to achieve a reliability improvement of 20% with respect to a service broker that does not fully exploit the MAPE-K architecture.

REFERENCES

Agarwal, V., & Jalote, P. (2010). From specification to adaptation: An integrated QoS-driven approach for dynamic adaptation of web service compositions. *In Proceedings of 2010 IEEE International Conference on Web Services (ICWS '10)* (pp. 275-282).

Alrifai, M., & Risse, T. (2009). Combining global optimization with local selection for efficient QoS-aware service composition. *In Proceedings of 18th International Conference on World Wide Web (WWW '09)* (pp. 881-890). ACM.

Alrifai, M., Skoutas, D., & Risse, T. (2010). Selecting skyline services for QoS-based Web service composition. *In Proceedings of 19th International Conference on World Wide Web (WWW '10)* (pp. 11-20). ACM.

Anselmi, J., Ardagna, D., & Cremonesi, P. (2007). A QoS-based selection approach of autonomic grid services. *In Proceedings of 2007 Workshop on Service-oriented Computing Performance: Aspects, Issues, and Approaches (SOCP '07)* (pp 1-8). ACM.

Ardagna, D., Baresi, L., Comai, S., Comuzzi, M., & Pernici, B. (2011). A service-based framework for flexible business processes. *IEEE Software*, *28*(2), 61–67.

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

Ardagna, D., & Pernici, B. (2007). Adaptive service composition in flexible processes. *IEEE Transactions on Software Engineering*, *33*(6), 369–384.

Bellucci, A., Cardellini, V., Di Valerio, V., & Iannucci, S. (2010) A scalable and highly available brokering service for SLA-based composite services. In *Proceedings of 2010 International Conference on Service-Oriented Computing (ICSOC '10)* (pp. 527–541). Springer.

Berbner, R., Spahn, M., Repp, N., Heckmann, O., & Steinmetz, R. (2006). Heuristics for QoS-aware Web service composition. *In Proceedings of IEEE International Conference on Web Services (ICWS '06)* (pp. 72-82).

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

Canfora, G., Di Penta, M., Esposito, R., & Villani, M. L. (2008). A framework for QoS-aware binding and re-binding of composite Web services. *Journal of Systems and Software*, *81*, 1754–1769.

Cardellini, V., Casalicchio, E., Grassi, V., Iannucci, S., Lo Presti, F., & Mirandola, R. (in press). MOSES: A framework for QoS driven runtime adaptation of service-oriented systems. *IEEE Transactions on Software Engineering*, accepted for publication in June 2011.

Cardellini, V., Casalicchio, E., Grassi, V., & Lo Presti, F. (2007). Flow-based service selection for web service composition supporting multiple QoS classes. *In Proceedings of IEEE 2007 International Conference on Web Services* (pp. 743-750).

Cardellini, V., Di Valerio, V., Grassi, V., Iannucci, S., & Lo Presti, F. (2011). A performance comparison of QoS-driven service selection approaches. In *Proceedings of 4th European Conference Service Wave*. Springer.

Cardellini, V., Di Valerio, V., Grassi, V., Iannucci, S., & Lo Presti, F. (2011). A new approach to QoS driven service selection in service oriented architectures. *In Proceedings of IEEE 6th International Symposium on Service-Oriented System Engineering (SOSE '11)*.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to algorithms* (2nd ed.). MIT Press.

Dobson, S., Denazis, S., Fernandez, A., Gati, D., Gelenbe, E., Massacci, F., & Zambonelli, F. (2006). A survey of autonomic communications. *ACM Transactions in Autonomic and Adaptive Systems*, *1*(2), 223–259.

Ezenwoye, O., & Sadjadi, S. M. (2007). RobustBPEL2: Transparent autonomization in business processes through dynamic proxies. In *Proceedings of 8th International Symposium on Autonomous Decentralized Systems (ISADS '07)* (pp. 17-24). IEEE Computer Society.

Hart, G. (2011). The five Ws of online help for tech writers. *TechWhirl*. Retrieved January 24, 2012, from http://techwhirl.com/columns/the-five-ws-of-online-help/

Hwang, C., & Yoon, K. (1981). *Multiple criteria decision making. Lecture Notes in Economics and Mathematical Systems*. Springer.

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

Klein, A., Ishikawa, F., & Honiden, S. (2010). Efficient QoS-aware service composition with a probabilistic service selection policy. *In Proceedings of 2010 International Conference on Service-Oriented Computing (ICSOC 2010)* (pp. 182-196). Springer.

Leitner, P., Wetzstein, B., Karastoyanova, D., Hummer, W., Dustdar, S., & Leymann, F. (2010). Preventing SLA violations in service compositions using aspect-based fragment substitution. In *Proceedings of 2010 International Conference on Service-Oriented Computing (ICSOC 2010)* (pp. 365-380). Springer.

Martello, S., & Toth, P. (1987). Algorithms for knapsack problems. *Annals in Discrete Mathematics*, *31*, 70–79.

Menascè, D., Casalicchio, E., & Dubey, V. (2010). On optimal service selection in service oriented architectures. *Performance Evaluation*, *67*(8), 659–675.

Menasce, D., Gomaa, H., Malek, S., & Sousa, J. (2011). SASSY: A framework for self-architecting service-oriented systems. *IEEE Software*, *28*(6), 78–85.

Michlmayr, A., Rosenberg, F., Leitner, P., & Dustdar, S. (2010). End-to-end support for QoS-aware service selection, binding, and mediation in VRESCo. *IEEE Transactions in Service Computing*, *3*(3), 193–205.

Mirandola, R., & Potena, P. (2011). A QoS-based framework for the adaptation of service-based systems. *Scalable Computing: Practice and Experience*, *12*(1), 63–78.

Mosincat, A., Binder, W., & Jazayeri, M. (2010). Runtime adaptability through automated model evolution. *Proceedings of 14th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2010)*, (pp. 217–226).

OASIS. (2007). *Web services business process execution language* (WSBPEL). Retrieved from http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

Papazoglou, M. P., Traverso, P., Dustdar, S., & Leymann, F. (2007). Service-oriented computing: State of the art and research challenges. *IEEE Computer*, *40*(1), 38–45.

Rosario, S., Benveniste, A., Haar, S., & Jard, C. (2008). Probabilistic QoS and soft contracts for transaction-based Web services orchestrations. *IEEE Transactions in Service Computing*, *1*(4), 187–200.

Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S., & Lorenzo, J. (2009). MUSIC: Middleware support for self-adaptation in ubiquitous and service-oriented environments. In Cheng, B. H., Lemos, R., Giese, H., Inverardi, P., & Magee, J. (Eds.), *Software engineering for self-adaptive systems* (pp. 164–182). Springer-Verlag.

Salehie, M., & Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. *ACM Transactions in Autonomic and Adaptive Systems*, *4*(2), 1–42.

Wada, H., Suzuki, J., Yamano, Y., & Oba, K. (in press). E^3: Multi-objective genetic algorithms for SLA-aware service deployment optimization problem. *IEEE Transactions in Service Computing*.

Yu, T., Zhang, Y., & Lin, K. (2007). Efficient algorithms for Web services selection with end-to-end QoS constraints. *ACM Transactions in Web, 1*(1).

Zheng, Z., Ma, H., Lyu, M. R., & King, I. (2011). QoS-Aware Web service recommendation by collaborative filtering. *IEEE Transactions in Service Computing*, *4*(2), 140–152.

KEY TERMS AND DEFINITIONS

MAPE-K: A reference model for architecting self-adaptive systems.

Quality of Service (QoS): The property of a service to provide predictable performance despite the availability of a limited set of resources.

Self-Adaptive: The capability of a system to autonomously change its behavior with respect to changes in itself and/or its surrounding environment.

Service Broker: An intermediate entity between users of SOA applications and candidate service providers. It offers a value-added service, possibly satisfying some QoS constraints.

Service Oriented Architecture (SOA): An architectural paradigm for building loosely-coupled network applications based on black-box software components named services. Such services can be easily composed to support dynamic and flexible applications.

Service Selection: Given a workflow, the ability to choose for each task in the workflow one or more specific service among a set of functionally equivalent implementations offered by service providers. The selection goal is to optimize some objective function (e.g., global utility) possibly subject to some constraints.

Workflow: A sequence of connected tasks and the related data flows representing the application logic.