UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

**THÈSE**

*pour obtenir le titre de*

***Docteur en Sciences***

*de l'Université de Nice - Sophia Antipolis*

**Mention Informatique**

*présentée et soutenu par*

**Cristian RUZ**

# AUTONOMIC MONITORING AND MANAGEMENT OF COMPONENT-BASED SERVICES

*Thèse dirigée par* Françoise Baude,

*au sein de l'équipe* OASIS,
*équipe commune de l'*INRIA Sophia Antipolis*, du* CNRS *et du laboratoire* I3S

*soutenue le 23 juin 2011*

*Jury:*

| | | |
|---|---|---|
| *Président du Jury* | Pr. Mireille BLAY-FORNARINO | Université de Nice - Sophia Antipolis |
| *Rapporteurs* | Pr. Philippe LALANDA | Université Joseph Fourier, Grenoble |
| | Pr. Lionel SEINTURIER | Université Lille 1 |
| *Co-Rapporteur* | Dr. Romain ROUVOY | Université Lille 1 |
| *Examinateurs* | Dr. Luc BELLISSARD | TAGSYS |
| *Directeur de thèse* | Pr. Françoise BAUDE | Université de Nice - Sophia Antipolis |

*“À ma mère,*
*et toujours à Álvaro”*

# Acknowledgements

I would like to start by thanking the CONICYT-INRIA Scholarship program for funding my studies in France during these years and for giving me the possibility to live this experience abroad.

Undoubtedly my biggest gratitude goes to Françoise for her continuous support and guiding through my thesis, for asking always the right question about which I had not thought before, for always allocating some time in a busy schedule to discuss matters, and for having a very pleasant disposition at all times. I really learned a lot. Thanks to Tomás for making the link. I would also like to thank the members of my jury for taking the time to read, comment and improve my work.

Je voudrais remercier tous les membres de l'équipe OASIS, pour son aide et pour avoir eu le courage d'accueillir un 6e chilien dans l'équipe. J'espère ne pas avoir abîmé votre image des chiliens avec mon français pourris et j'aimerais bien vous montrer mon pays (mais pas tous en même temps !).

It is never easy to write a thesis far from home. You are far from your environment, far from the people you love, from the people you care about, and you miss many important events, things and persons. However, in every place I have lived, I have always been blessed of knowing wonderful people and very good friends. I should thank here lots of people. Merci Sylvie et Cathy pour toute l'aide bureaucratique (ce n'est pas une petite chose en France), merci Paul et Bastien pour l'aide technique, merci Mélaine et ses balades. Muito obrigado Guilherme por muitas horas "hardly working". Thanks Elton, Imen and Uzair, my fellow predecessors. Merci-ura, Sheheryar. Grazze mille a Franca e Elvio. Multumesc Ana, Raluca şi Emil, eu trebuie să înveţe româneşte. Gracias a Gaby y los padrísimos (o no tanto) partidos que vimos. Gracias a mis amigos argentinos (cordobeses o no). Gracias a Mario, Antonio, Betty, Ángela, Pato, Diego por recibirme en los primeros meses; a Paula y Mar por unos meses increíbles; a Marcela, Carlos, Juan Carlos, Mara y Vanessa por la permanente compañía; a Carolina y Sebastián por su invaluable amistad, ayuda y soporte en los últimos meses.

Por supuesto, a toda mi familia y amigos en Chile y Bélgica, del colegio a la universidad, del DCC-PUC a la UDP, de Antofagasta a Santiago, Curicó e intermedios, hay algo de cada uno de ustedes en este trabajo. Gracias por mantener el contacto a la distancia, por teléfono, por internet, por Skype, en cada etapa desde mi partida en Marzo 2008 hasta ahora, escuchando quejas, chistes, trivialidades, y todo lo demás. Gracias a mis ahora compadres, Federico y Claudia por la alegría de darme un puesto de padrino para Juan Pablo. Ya nos juntaremos para celebrar con cada uno de ustedes, y para recordar a quienes partieron en alguna etapa del camino (Don Manuel, tío Luchín, Silvana, Max, Silvestre, Tom, señor Patán, nos veremos más adelante).

Last, but not least, two very special ones. De no haber trabajado con Álvaro, talvez no habría seguido este camino. Te debo otra marca en el árbol. Sin mi madre y su infinita fortaleza de tenerme lejos, primero en Santiago, luego en Antibes, todo hubiese sido más difícil. Te amo todo el tiempo, hasta cuando no se nota.

And to you, who read all this page, if I did not mention you explicitly or implicitly, my apologies. You should also be here.

Cristian,
à Sophia Antipolis,
le 23 juin 2011

# Contents

# List of Figures

# List of Tables

# List of Listings

# 1 Introduction

## Contents



## 1.1 Motivation

In recent years, software applications have evolved from monolithic, stable, centralized and structured applications, to highly decentralized, distributed and dynamic software. This evolution has naturally enforced a change in the development process of such software.

The practice of building in-house software for specific purposes, turned to the development of small pieces of encapsulated code implementing specific tasks that can be reused in different applications, that were known as "off-the-shelf" components. Moreover, these software components, that gave base to the concept of component-based software development, could be independently developed and provided by third parties, decentralizing, at least partly, the development process. This practice reduced the necessity of rewriting commonly used tasks, helped to cope with changing requirements, and focused in an intelligent separation of tasks that are delegated to software components, that are glued together to create the new application [DNGM+08].

However, in that situation, the ownership and the management of the application are still in the hands of the entity that develops the application. The next step came when the focus turned to the provision of a *functionality* instead of just a piece of software, and this functionality begun to be provided "as a service". In this situation, independent providers offer a set of functionalities in the form of *services* that can be accessed and used in a standard way, facilitating the aggregation of such services to create new service compositions with added value, better suited to their current needs. Such applications are referred to as *service-based applications*.

The service-based approach to software development has, undoubtedly, many advantages. Code reutilisation, outsourcing, and modularity provide a more rapid development process. The possibility of having third parties providing functionalities that can be accessed in a standard way has given rise to an ever-growing number of loosely-coupled geographically dispersed services available on the Internet, that has naturally taken the role of being the delivery means for such services. Composition standards have been developed to facilitate the creation of aggregated services giving rise to a rapidly evolving service ecosystem.

Such dynamism has also been a response to evolving requirements in software development. In fact, the software development process used to consider a "closed-world" assumption in which the external world changes so slowly that software can remain stable for a long period before any major update needs to applied. However, each time more and more situations arise where this

assumption is not anymore valid and software applications must face an "open-world" in which the environment changes continuously and applications need to **adapt** and **react** to changes in a dynamic way, even if those changes are unanticipated [BDNG06].

However such dynamicity, loosely-coupling and heterogeneity of providers has introduced not trivial challenges in the management of service-based applications. As the maintenance of the software does not depend completely on the provider of a service composition, due to the fact that some pieces of them may be under the control of third parties, some characteristics that are usually manageable in controlled environments, like availability, response time, security, and others related to the quality of the delivered service, are not always ensured. Even if the provider is able to manage all the services in a composition, environmental factors like network latency, congestion, and hardware failures may affect the service in unforeseen ways. Traditional optimization cycles where the application is stopped, analyzed, modified and restarted does not fit well in a situation where not all the services are under the control of a single entity, and where there is a need to timely react to requirements and environmental changes, automating the analysis and decision taking phases as much as possible, and with a minimal perturbation in the availability of these services.

At the same time, service-based applications are defined in accordance with some goals related to the quality of the service provide (QoS goals), and they are expected to comply to these goals at runtime. It is thus, a necessary requirement that confronted to such dynamically evolving environments, service-based applications need also to dynamically evolve and adapt by having an appropriate managing mechanism. But not only that. Given that the requirements and conditions over the application may also evolve in unpredictable ways, the management mechanism may also need to dynamically evolve.

As an example, consider applications that aggregate content available from other sites like Google News or Drudge Report for news, Metacritic for movie reviews, or metasearch engines like Metacrawler. In all these cases the content is not provided by the site itself, but obtained from other providers and filtered according to user preferences. Consider also travel applications like TripAdvisor or Expedia which aggregates different services like hotel reservation, client reviews, geographic locations, car renting and recommendations into a single site where the user can make all the needed planning. Again, those sites also do not offer the final product themselves, but they provide a uniform and easier way to use different pre-existing services. However, if some of the services that these compositions use experience problems, these problems may propagate to the composed service if they are not appropriately handled by a manager.

As mentioned, evolution can be triggered by different sources. Services that become rapidly popular require better infrastructures to support their service, and several of them ultimately decide to migrate to highly available and elastic infrastructures (like *cloud* environments) that provide different levels of support to host their applications. However, the fact of depending on third-party provided services, whether it be for hosting the content offered by the composition, or for the infrastructure support requires appropriate management in case of unexpected situations, like a slow response from the content providers, or unavailabilities in the case of infrastructure providers. Any failure in some part of these "composed" services may reflect in failure in the final provided service.

## 1.2 Problem description

As expressed in the brief motivation presented above, service-based applications face a situation where "everything can change", and they need to be able to timely adapt to such situations with minimal perturbation to the functionality they provide.

There are several issues that can be pointed out when analyzing this situation:

- **Lack of uniformity and flexibility in different services**. Each service that is used as a composing part of another service, and that is developed and provided by a different party,

may provide a different management interface even if the functionality is equivalent, and the conditions under which it can be used (cost, number of requests, availability, response times), may also be different. When devising an action that involves the manipulation of different services, it is not trivial to consider all the set of different interfaces that can be available. Also the management features for each externally provided service may be different, limiting the overall flexibility.

- **Impossibility of foreseeing all the possible changes and conditions** that can influence the functioning of a service-based application. When considering services composed by other services provided from different sources, the range of unexpected failure possibilities becomes wider. However, from the point of view of the user of the final composition, any failure is a responsibility of the composition and not necessarily of the individual services, which are hidden by the composition. Service compositions, thus, need to be prepared to adapt to different situations that may arise, often, in unexpected ways, in some of the services they use.
- **Complexity of developing effective autonomic tasks**. An autonomic tasks is a task that can be decided and executed by the application automatically under certain conditions. This kind of tasks seems very convenient for programming adaptations to changing conditions into an application, eliminating the need of having human managers involved in certain tasks. However, when coordinating actions among multiple services with different interfaces and different managing capabilities, this task is even less trivial. Plus, to effectively attain a global goal in a composition, an autonomic task may need to be subdivided through the different services involved, and this subdivision is not easy.

Several solutions have been proposed for tackling the complexity and heterogeneity of service compositions. The *component-based* model for software engineering (CBSE), applied to the service-orientation area has provided a generic and structured model for designing service-based applications, called SCA (Service Component Architecture); however this model is focused at the design of a service-based application and does not consider runtime modifications. On another side, the *autonomic computing* initiative is an ambitious discipline that promotes the idea of having systems that can manage themselves given certain high-level objective to achieve, however the effective coordination required to provide a global autonomic behaviour to a composition of seemingly independent services is still a major issue.

One of the main ideas taken from the *autonomic computing* area is the construction of closed *autonomic control loops* that include several phases in which a situation is observed, then analyzed, and a decision taken to react to this situation. This idea has been successfully applied to specific applications to be able to react to uncertain situations, however their implementation under uncertain management requirements is not so direct in dynamically evolving environments as service-based applications.

Given this wide panorama of solutions, it is quite clear to see that autonomicity is a helpful discipline for making services more adaptable to evolving conditions. Nevertheless, a proper provisioning of autonomic behaviour should take into account that a composition may vary during its lifecycle, and also the requirements made over the autonomic behaviour must be handled.

We believe that an appropriate use of a component-based approach in a structured service-based application can facilitate the adaptation of a service to changing environmental conditions; and, by applying these adaptation features to several individual services, it is possible to reach a better adaptation capability for a service composition, and provide autonomic behaviour to services.

## 1.3 Goals

In this thesis we promote the idea that an appropriate means for monitoring and managing service-based applications must be attached as close as possible to the services they intend to

manage, and it must be flexible enough to modify its own behaviour in order to adapt to different management requirements.

This model of flexible management features that can be inserted and removed from different services in a composition, would allow to better adapt to changing conditions, and can facilitate the integration of autonomic tasks.

That said, we specify the goal of our research as to **improve the adaptability of service-based applications by providing a common and efficient means to monitor and manage services, while being low-intrusive and flexible enough to adapt to changing management requirement, ultimately facilitating the provisioning of autonomic tasks to services**.

## 1.4 Solution Overview

Our solution consists in a flexible monitoring and management framework that can be attached to individual services in a component oriented service-based application, and consider different levels of management, from the overall composition of the application, to the computational resources where these services run.

For being able to provide such a system, we propose a framework that presents the following features:

- Component-based approach, to enforce separation of concerns, by modeling different elements of the management system as components that can be plugged-in or plugged-out of the framework.
- Allow different protocols to monitor and manage services, leveraging them to a common ground, where decisions can be taken more easily.
- Take into account the infrastructure where the service runs, providing management capabilities that can include several levels of the composition, from higher level goals, to lower level resource management.

The solution takes the form of an *autonomic control loop* where its different phases are implemented as components and these components are attached to the services where some monitoring or management feature is needed. This intended flexibility implies that not all the services of a composition need to include a completed closed autonomic control loop, but rather they include only parts of it. We believe that this flexibility facilitates the construction of autonomic tasks better adapted to the service needs.

## 1.5 Thesis Structure

Along the presentation of this thesis, we separate the description of our solution from the support implementation that we provide to demonstrate the feasibility of our approach. We have structured the contents in the following way:

- Chapter 2 gives the **context** in which our work develops, describing the common terminology and the different areas that are related to our contribution.
- Chapter 3 presents the **state of the art** showing different existing works and tools that have been proposed to tackle some of the challenges that we aim to address. We present a comparison between the different features that they provide and classify them according to how their work compares or inspires our proposition.
- Chapter 4 presents our contribution and **positions** it with respect to the current state of the art. We describe the kind of problems that we aim to solve and how our solution helps in addressing those issues.

- Chapter 5 details the **design of the framework** that we propose and the consideration that we have taken into account in each phase of the autonomic control loop, the interfaces we have defined and how they can be used.
- Chapter 6 presents a **technological background** about GCM/ProActive, which is the middleware tool that we use to provide a concrete implementation of our framework, and also describes the technical contributions that we have made to this middleware in order to support our solution.
- Chapter 7 details the **implementation** that we have provided over the GCM/ProActive middleware, detailing each element of the implemented framework.
- Chapter 8 presents some experimental **evaluations** that we have conducted over our implementation, and describes an example use case about a component service-based application that is gradually improved with monitoring and management features by using our solution.
- Finally, Chapter 9 shows the final **conclusions** about our work and describes some perspectives for future research and improvement.

# 2

# Context: Services and Components

## Contents



This chapter provides the background and concepts required for a proper comprehension of the topics addressed during this thesis. Along with describing the concepts involved in this work, this chapter also intends to present the research challenges that have a relationship with the research questions that motivate this thesis.

The general area in which this thesis develops touches several fields. Our contribution looks to improve the implementation of service-based applications, so we introduce the *service-orientation* area to highlight the importance of service-oriented approaches in the current application development trends. At the same time we target a particular kind of service-oriented applications, those that have been built following a structured *component-oriented* model, which provides several advantages in terms of abstraction, structure and flexibility. As service implementations needs a reliable supporting infrastructure, current trends point in the direction of highly available infrastructure like *grids* and *clouds* as an appropriate solution; in our work we aim to consider the impact of that supporting infrastructure in the adaptation decisions that can be taken over the application. A major research area that looks how to improve the adaptation not only in service-orientation but in a broader range of applications is the *autonomic computing* area, whose principles we adopt in our proposition.

The chapter is organized as follows. Section 2.1 describes the paradigm of service-orientation and its relationship with the widely used Service Oriented Architecture concept, describing a model that will be used to position our contribution. Section 2.2 reviews the concept of Grid and Cloud computing, relating them to common appropriate infrastructures to support service-based applications. Section 2.3 reviews the paradigm of Component-based Software Engineering as a

means to support the construction of modular software that share several features with service-based applications. Section 2.4 describes the Service Component Architecture specification that promotes a component-based approach to develop and support Service Oriented Architectures. Section 2.5 mentions some of the challenges of the Autonomic Computing area and describes how they can support the construction of autonomic and adaptable service-based applications Finally, section 2.6 summarizes the main research challenges that will to be addressed during this thesis.

## 2.1 Service-orientation and SOA

After several years of advances and refinements in programming paradigms for easing the programming of complex software applications, the concept of *service-orientation* may seem like just another way of describing a much older concept.

A proper description about service-orientation requires a definition of the concept of *service*.

The concept of *service* is common in every day world. Indeed, each person who carries out a task providing a benefit to others, is performing a *service*. A taxi driver, a mailman, a salesperson, a cooker, a medic, a janitor, an accountant, all provide well defined concrete *services*, which may be called transportation, mail delivery, selling, cooking, medical attention, cleaning, accounting. Moreover, all these separate and well defined services may take part in a larger chain. For example, a person who is arranging a wedding may use a delivery service for sending invitations, a civil officer, a photography service, a catering service, a location renting service, and using each of those individual services, it composes a bigger and more complex wedding service. It is worth noting, that the composed wedding service is also a service that conceptually is not different from the individual services from which it is composed. Each of the services provides a specific functionality, which is accessed by the specific predefined means, while the way to carry them out is an internal concern that is not expressed in the description of the services. Also, each of the services that compose the bigger service continue existing by themselves, independently if they form part of another service or not.

A *software service* is not much different. One of the initials, and still common definitions for services was given by Mike Papazoglou:

> "Services are self-describing, platform-agnostic computational elements that support rapid, low-cost composition of distributed applications." [Pap03]

From this definition, one of the important notions is that of self-description. In fact, it is expected that a service provides all the information required to use it. The mention of rapid and low-cost composition implies also a notion of loosely coupling and the fact that a service can be used as part of a bigger distributed application. Finally the requirement of being platform-agnostic implies that a service must not be tied to any specific technology.

A complementary vision can be found in the definition given by the OASIS [1] Consortium:

> "A service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description" [OAS06]

This definition gives importance to *how* to access the capabilities, while describing nothing about the capabilities themselves. This is because in a *service* the logic is encapsulated and abstracted from the rest, as it is not a main concern. In practice, services form units of solution logic that provide a set of related capabilities that can be publicly invoked to achieve a well defined goal, usually expressed in a service contract [Erl].

Following these definitions, and the practical use of services, some important elements can be identified:

[1] Organization for the Advancement of Structured Information Standards

- **Service provider**, as an entity that offers a service and makes it available, including its description, and providing the related technical and business support.
- **Service consumer**, as an entity that uses a service by accessing to the capabilities that it offers. A service consumer can be another service, or external application, customer, user, or any entity capable to use the capabilities of the service according to certain rules.
- **Service description**, or service contract, is the element that contains, in a precise way, all the information required to access the capabilities offered by the service.

### 2.1.1 Service-Orientation

The concept of *software service* (in the following, called simply *service*) gives support for the paradigm known as *service-orientation*. In contrast, for example, to object-orientation, where the main entities are groups of methods with certain characteristics (and usually modelled from real world elements), or to component-orientation, where individual, modular pieces of software are composed through interfaces, *service-orientation* treats *services* as first class elements.

It must be noted, however, that *service-orientation* is not a completely different paradigm for building applications. Instead, service-orientation shares several design guides and concepts with other paradigms.

Applications built using the *service-orientation* paradigm, known as *service-based applications* (or *service-oriented applications*) use *services* as basic units of composition to create solutions. When designing a service-based application, the priority is the definition of separated functionalities that perform a specific task, and that are made available in a way that can be used and take part in other solutions.

The vision of service-orientation is that applications will be created by easily assembling small well-defined tasks, publicly available as services. [PTDL07]. This vision has an implicit requirement for service *interoperability*, which is reflected in the way that services communicate and publish their capabilities, and how they interact with other services. If services are developed in an interoperable way, then different providers can develop and offer their services in an independent way, helping to realize the vision of service-orientation.

*Service-Orientation* is guided by design considerations, referred by Thomas Erl [Erl] as the following *service-orientation principles*:

- **Standardized Service Contracts**. Regardless of the specific format used for describing the service capabilities (i.e. the service contract), it is important that this format be known and used by all the parties that access the service.
- **Service Loose Coupling**. The level of dependencies between services must be as low as possible. In fact, consumers should depend only on the capabilities expressed in the service contract, and not on the implementation of the capability itself. This allows independent evolution of services while still ensuring a base of interoperability.
- **Service Abstraction**. The service must hide as much of the underlying details of the internal logic as possible. By relying only in the information published in service contracts, and hiding the rest, the service avoids unnecessary dependencies and preserves loosely coupled relationships.
- **Service Reusability**. The capabilities of a service must be provided for a technologically-agnostic context. This ensures that the service can be reused in other contexts that were not initially considered.
- **Service Autonomy**. The autonomy of a service is increased as they have more control over their underlying runtime implementation. The ability of service to run autonomously [2], the more predictable its runtime behaviour will be. This is particularly important when

[2]The term "autonomous" must not be confused with "autonomic". In fact "autonomicity" is an important concept later in this thesis.

designing a service that is composed of other services, as the composed service is less autonomous.

- **Service Statelessness**. Although services are not forbidden to be stateful, it is recommended that services delegate the management of state information to specialized entities, and focus in functionality, in order to minimize shared resource consumption and increase scalability.
- **Service Discoverability**. Services must provide metadata that allows them to be effectively discovered and interpreted. An appropriate metadata description about a service increases the opportunities for reuse and composition. Note that this principle does not assume the existence of a service registry.
- **Service Composability**. Services should be designed in a way that they solve specific concerns, so to improve their ability to be part of a service composition. This is the service-oriented way to provide *separation of concerns*.

One of the objectives of these principles is to promote the existence of an interoperable set of services. In fact, interoperability can be identified in each of these principles.

### 2.1.2 Service Oriented Architecture (SOA)

A service-based application can be built in a number of ways. *Service Oriented Architecture* (SOA) emerged as an architectural model for realizing service-based applications. As defined by Papazoglou:

> "SOA is a logical way of designing a software system to provide services to either end-user applications or other services distributed in a network through published and discoverable interfaces."[Pap03]

SOA comes to bring a common way to organize a service-based solution promoting reuse growth and interoperability. As mentioned in the OASIS Consortium reference model for SOA: An SOA "is not itself a solution to domain problems, but rather an organizing and delivery paradigm that enables one to get more value from use both of capabilities which are locally 'owned' and those under the control of others. It also enables one to express solutions in a way that makes it easier to modify or evolve the identified solution or to try alternate solutions" [OAS06].

This last description mentions some important concepts:

1. an SOA is not a solution itself, but a way to organize and build (service-based) solutions;
2. there is a need for interoperability, as there can exist different and heterogeneous service providers; and
3. solutions should be able to evolve and be easily modifiable.

As for the participants, their interactions can be defined using the basic model for an SOA, shown in Figure 2.1.

In the basic relationship, (1) the entity that wants to offer a capability, the **service provider**, *publishes* a **service description** in an intermediate entity called **service registry**. The service registry stores a set of service descriptions and makes possible to perform queries over them. As second step, (2) an entity that wants to *use* a service, the **service consumer**, queries the service registry and *discovers* the available services based on their service descriptions. With that information, (3) the service consumers can *find* the owner of the service, the service provider, and *bind* to the provided service and initiate a (4) request/response communication. As seen, the service registry acts a **service broker** that allows service consumers and providers to find each other. In the following we will use the terms consumer and provider to refer respectively to service consumers and service providers.

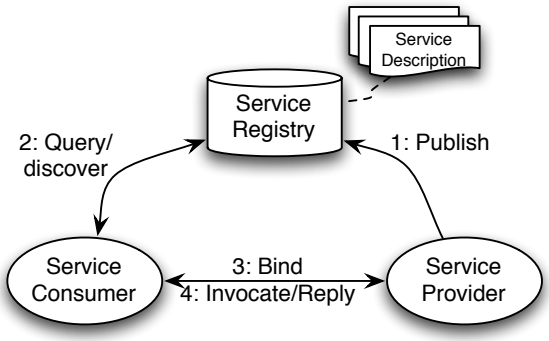


Figure 2.1: A basic Service Oriented Architecture

This basic model defines a technologically-agnostic construction where the elements of the service provisioning can be organized and interact. The model allows to show the basic interactions between consumers and providers. In practice, the service consumer/provider relationship can be more complex. When a consumer finds a provider, they must agree to initiate the relationship and this is usually established in a *service contract* after a *negotiation* process. Once the conditions are agreed, their compliance needs to be **monitored** during the service provisioning. When creating service compositions, it is necessary to **coordinate** the different services in order to provide an effective composed service. Papazoglou et al. [PTDL07] proposed a commonly referred model that takes into account several functionalities required by an SOA and separate them in three layers, as shown in Figure 2.2.

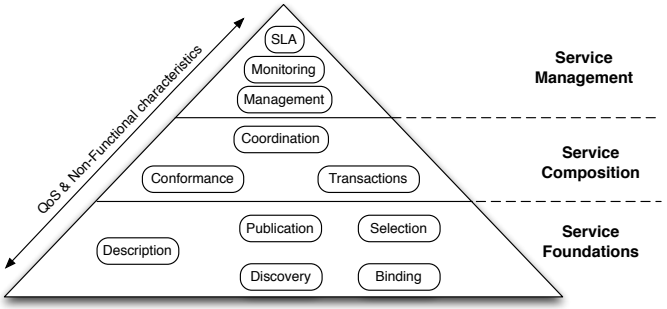


Figure 2.2: Extended Service Oriented Architecture functionalities [PTDL07]

In this model, the functionalities required by each layer are based on those provided by the lower layer. The lower layers cover the basic services relationship, the middle layer groups issues involved when composing services, and the highest layer tackles management of services. A brief description of these layer and functionalities follow: [PH07]:

**Service Foundations** Involves the middleware that serves as a platform for the service-oriented application. The middleware should allow to connect possibly heterogeneous providers and let the application developers to define the service functionality in terms of description, publishing, finding and binding of services. Service providers and consumers follow the roles described in Figure 2.1, and can include actors called **service aggregators**. A service aggregator groups services published by other providers and offers them as new value-added services, acting also as providers.

**Service Composition** Involves the tasks performed by **service aggregators** to take multiple services and form a new added value *composite service*. A composite service can be seen as any

other simple service and, as such, can take part in the creation of another composite, or form an end solution by itself.

In this context, terms like *orchestration* and *choreography* are used to describe a means to coordinate the interactions involved in the composition of services. Functionalities like transactions, and conformance are needed to ensure effective compositions.

**Service Management** This layer requires to realize monitoring and management activities over the services. This includes the collection of information about the SOA-based application during its lifecycle, and to be able to control the service in order to comply to certain conditions.

Management activities include monitoring the appropriate metrics to ensure that the service behaves according to certain conditions, provide information about the performance and other characteristics of the service, control the lifecycle or state of the service, and being able to modify the composition of the service in order to adapt to some specified conditions.

A very important element in Figure 2.2 is that "QoS and non-functional characteristics" are transversal to all layers of the model.

**Quality-of-Service** (QoS) is a term that was originally defined in the telecommunications area to describe "A set of quality requirements on the collective behavior of one or more objects", and that has been applied to the services terminology to indicate a set of properties that allow to characterize the non-functional aspects of service, i.e., aspects that are not related to how the functionality of the service is implemented, but with how this functionality is delivered to the consumer. These properties are referred to as *QoS characteristics* or *Qos attributes* and are measured by obtaining *Qos metrics*.

QoS characteristics can be measured through all the layers of the model. In the Service Foundations layer, QoS characteristics of individual services can include the response time of a request, the availability of a service, the cost of a service, the security model required to use the service, or others. The management of QoS characteristics at this layer may involve modifying some parameter of the platform where the service executes, change the resources available for the service, or tune some parameter of the service implementation, among others.

At the composition layer, QoS characteristics may involve several services and their computation involves aggregating the individual QoS characteristics of the composing services in a meaningful way.

At the management layer, QoS are important to decide global actions upon a composed service, for example, to decide to deploy/start/stop a service, or modify some parts of a composition. This level is related to the compliance with global goals about the composition.

In the end, a proper QoS management should be able to integrate the QoS concerns expressed through each layer and expose them in a meaningful way.

When establishing a consumer/provider relationship, goals about QoS are usually stated as conditions in a **Service Level Agreement** (SLA). An SLA is a contract between the consumer and the provider that may include, among other items, goals about the QoS of the service during the consumer/provider relationship, known as **Service Level Objectives** (SLO). In turn, SLOs are usually stated as *conditions* that must be verified during the provision of the service.

### 2.1.3 Implementing SOAs

Being an architectural model, SOA is not tied to any specific implementation technology. However, common approaches and tools have been developed that support the development of SOA-based solutions.

A commonly used technology for implementing SOAs is the Web Services architecture [W3Cb]. The Web Service stack provides protocols and standards that cover several aspects and that can be used to implement SOAs, and several providers rely on these technologies to support their SOA, although it is not the only way to implement SOAs.

Regarding the basic Service Foundations layer, services can be specified and described using the Web Services Description Language (WSDL) [W3Cc]; registry and discovery are supported by the Universal Description Discovery and Integration (UDDI) specification [OASa]; and the interaction between services uses the SOAP [W3Ca] communication protocol. Support to describe Service Compositions is commonly addressed by workflow languages like WS-BPEL [OASb]. Management and monitoring activities can be supported by numerous languages, depending on the kind of activities targeted, like WS-Policy [W3Cd], WS-Management [(DM], or WSDM [OASc].

However, as more heterogeneous technologies are available to address different SOAs, the need for an open and extensible solution has given raise to the concept of **Enterprise Service Bus** (ESB) [Cha04]. The ESB is an open standards-based message backbone that can support the implementation, deployment and management of SOA-based solutions. An ESB works as a message broker that can interact with different technologies by implementing the appropriate interfaces, and enables connectivity between services that use different formats. An abstract vision of an ESB is shown in Figure 2.3.

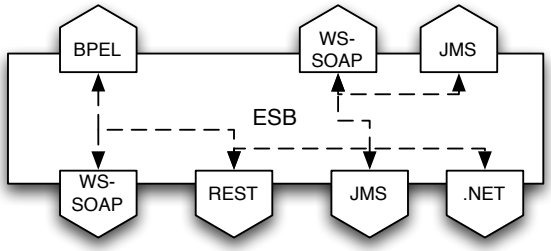


Figure 2.3: An Enterprise Service Bus (ESB) allows transparent connectivity between heterogeneous services.

In practice an ESB allows loosely coupled connectivity of services and, due to its middleware nature, can manage the bindings between services and support non-functional features as transactions, security, performance metrics, dynamic configuration and discovery.

### 2.1.4 Challenges in service orientation

Several challenges can be mentioned in the development and evolution of service-based applications. Instead of listing them exhaustively, we will concentrate on those that are related to the goals we want to address in this thesis.

- **Dynamically (re-)configurable runtime architectures**. The architecture of the service-based application should provide the means to be (re-)configured at runtime and to be optimized in accordance with the requirements of the application.
- **QoS-aware service compositions**. In a context where there may exist multiple providers for services that provide similar or equivalent functionalities, QoS attributes play an important role in helping to determine a selection of proper services to ensure some global QoS condition. This type of composition is called *QoS-aware composition*, and refers to taking into account the QoS attributes of a service previous to making it part of the composition, as the global QoS attributes of the composed service will probably depend on the individual QoS of the individual services selected.
- **Autonomic services**. Provide autonomicity to services can be viewed from several points of view. As one advantage, it can allow to manage service infrastructures with minimal human intervention. It also can be used to adapt different parameters of the service in order to comply with certain QoS-related goals.

- **Autonomic composition of services**. Autonomicity can also be provided at the level of the service composition. By giving autonomic capabilities to service composition, new services could be created giving some high-level goals and relying on a "service composer" element to automatically discover, select, bind and compose the new service. At the same time, the composed service is able to modify its own composition and adapt to changing conditions.

To properly support these challenges, some lower level support must exist. In particular, monitoring and management facilities must be available at each level of the service-based application in order to obtain the required data to perform QoS-aware compositions, to dynamically reconfigure services architectures, or to perform autonomic tasks.

SOAs have particular characteristics that make the monitoring and management of QoS a non-trivial and challenging task as we describe below.

The loosely coupling and multiplicity of providers in SOA make this a **dynamic** environment, and in such environment, the consumer/provider relationships may vary during the service execution. In particular, in SOAs not all the services are usually under the control of a single entity and there may be multiple sources for unpredictable behaviour during the service provisioning, ranging from internal characteristics of the service provider, to issues in the channel used to deliver the service, as usually the Internet is used for this matter.

Such dynamicity makes that the QoS attributes of a service composition be hardly predictable. Although, an initial composition may comply to some conditions at design time, and the initial deployment time, it becomes necessary to ensure that these conditions also hold at runtime, even after some reconfigurations may be applied. That requires continuous *runtime monitoring* during the lifecycle of the service.

Management activities for loosely coupled applications include installation, deployment, configuration, metrics collection and tunning to ensure a responsive service execution. The monitoring of information about the managed-service platform allows to diagnose performance problem via root-cause failure analysis, realize SLA Monitoring, and autonomically decide on actions.

## 2.2 Grid and Cloud Computing Resources

The growing offer of services has enforced the need for highly available computational resources where to host these services, as on-premises infrastructure support is each time less convenient in terms of cost and maintenance. The plain offer of specific-purpose computational power exists from a long time ago in terms of *Grid Computing* resources, whose research area has faced, among others, the goal of having a highly available and interoperable network of computing power. The service-oriented approach has somehow complemented this vision and enforced a delivery model of such resources "as a Service" pushing the term of *Cloud Computing* to make computational resources available at different levels of abstraction.

The Grid and Cloud computing terms are highly interrelated and share several common issues and objectives [FZRL08]. Their differences in several aspects is also a matter of discussion. Instead of comparing side-to-side both technologies, we present their main characteristics and usefulness as supporting infrastructure for service-based applications.

### 2.2.1 Grid computing

The idea of having a world wide interconnected computational infrastructure available for executing applications is not completely new. Already in 1999, the concept of a Grid computing infrastructure [FK99] had the vision of unifying multiple providers of computational resources into a highly distributed universal source of computational power, accessible from any place. In practice, several Grid initiatives emerged, each with its own characteristics and tools, but few interoperable features that allow to realize the original vision.

A refined definition of Grid states that the main objective is "to enable coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations" [FKT01]. The term of *Virtual Organizations* (VO) came to define a set of individuals and/or institutions that own computational resources and that can define sharing rules for their resources as if they were all from the same organization. A commonly accepted checklist about what is, and what is not a Grid was given by Ian Foster [Fos02], stating that a grid:

1. coordinates resources that are not subject to centralized control.
2. uses standard, open, general-purpose protocols and interfaces.
3. delivers non-trivial qualities of service.

These definitions highlight some features about Grid infrastructures that will be relevant in the context of this thesis:

- Grids are multi-institutional platforms, which has as a consequence that both the resources shared and the sharing rules may be heterogeneous.
- Grids are dynamic, which implies that the resources available in a Grid may be available or not at different times and possibly in a rather unpredictable way.

Several efforts have been brought to standardize and unify the interfaces used for accessing and exploiting Grid infrastructures, like the Open Grid Services Architecture (OGSA) [FKNT03], later superseded by the Web Services Resource Framework [OASd]. However, as Grid initiatives are built for different purposes, most of them remain still different in terms of resources, software and tools.

Grid infrastructures have remained a popular means to offer computational resources. Their goals go in line with the vision of having a widely-available source of computational resources even if their interoperability is somehow restricted to certain organizations. They remain, nevertheless as a powerful supporting infrastructure for hosting services.

### 2.2.2 Cloud computing

Many definitions for Cloud Computing have been proposed and little consensus is still achieved. We choose the following definition that includes important characteristics for the context of this thesis:

> A large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms and services are delivered on demand to external customers over the Internet [FZRL08].

From this definition it is possible to identify some features of Cloud Computing platforms:

- Clouds provide *virtualized* resources. Cloud providers usually rely on virtual machines to share physical resources, thus offering isolated view of their resources to the users. Virtualization also simplifies management activities for end users, like configuration, deployment and replication.
- Clouds are dynamically scalable. A concept often known as *elasticity*. As a consequence of virtualization, resources can be offered to the user in a dynamic way as they need them, allowing the user to increase their virtualized instances and perform load balance between them.
- Clouds offer their services in a *pay-per-use* model, allowing the user to have control and balance their expenses with respect to the computing power they receive.

Clouds can be classified according to their service delivery model:

- SaaS (Software as a Service) provides access to software applications as services over the Internet, eliminating the need to host, install, and run the application on the client computer, simplifying maintenance and support for the user. This kind of applications, also called Cloud Applications are quite appropriate for supporting services in an SOA context. Some examples are: Google Apps (Mail, Calendar, Reader, Docs), or Salesforce.com.
- PaaS (Platform as a Service) delivers access to a computing platform or solution stack, supported by a Cloud infrastructure, where the users can host their own applications. The user has control over the applications that run on the platform, but not over the platform itself. Some examples include: Google AppEngine, or Windows Azure.
- IaaS (Infrastructure as a Service) delivers access to the Cloud infrastructure, usually in the form of a virtualized environment, where users can deploy all the software they require. Users have access to the virtual machines where their applications run and storage facilities, giving them the highest degree of control. Nevertheless, the users cannot access details about the physical infrastructure behind. Common examples are: Amazon EC2, Eucalyptus, or OpenNebula.

The fast growing of Cloud Computing offerings has reinforced the ideal vision of a global Cloud (originally Grid) of computing power available from any place. In fact, some authors already talk of the emergence of a multi-cloud platform composed by a mixture of different Clouds, and motivated by the need to balance the existence of in-house and external resources to obtain a better trade-off between cost and performance. This multi-cloud environment will have to cope with changes in the offered resources and user requirements in an adaptive way. Some commercial solutions like RightScale and FlexiScale, or research initiatives like Eucalyptus work on the integration of different Cloud providers in order to provide effective multi-Cloud platforms.

Although it is always a contentious topic, it is easy to see that Grid Computing and Cloud Computing platforms share many things [FZRL08]. In fact, both visions share the goal of reducing the cost of computing, and increase reliability and flexibility by switching from locally owned and hosted applications and infrastructure, to an externalized service that can be accessed as it is needed providing a potentially higher computing power.

Yet, the means to achieve these visions may differ in some aspects. Grid Computing infrastructures have been motivated by the need to achieve highly scalable computing resources and storage to applications with specific purposes (scientific, business, industrial). Cloud Computing, on the other side, can be seen as a concept that has evolved from the Grid area, to deliver more abstract resources and services to the users, with a more clear service-oriented view. From this point of view it is hard to say that Cloud Computing is just another name for Grid Computing, nevertheless they share several common challenges.

The adoption of Cloud computing is growing, and evidence says that it will continue to be so. Cloud environments provide a strong base for developing externally hosted applications, available “as a Service”. The vision of Cloud Computing of having a large pool of computing power, storage, platforms and services available seems like an appealing support for developing service-oriented applications, and many authors talk already about a *Service Cloud* [CGM10] as federation of sites and services from various infrastructure providers. The virtualization and dynamic scalability features have an important role in supporting service that can adapt their resource utilization.

## 2.3 Component-based Software Engineering

Component-Based Software Engineering (CBSE) is a paradigm for software development that attempts to solve the problem of *separation of concerns* by implementing concerns as software *components*. Software components can be defined as follows [Szy02]:

> A software component is a unit of composition with contractually specified interfaces

and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Software components (in the following, referred to simply as 'components') present some important features for software development:

- **Encapsulation**. Components encapsulate their internal logic, allowing them to be treated as black-boxes, accessible only through their defined interfaces.
- **Composition**. Components are composable. Components can be assembled in order to work together, and the composition can be performed by a third party.
- **Description**. All that is needed to know about a component is given by its description. The advantage of a precise description is that it allows to develop tools for composition and formally reason about the components.

Components are expected to be *reusable*. This means that a component that has been used for a composition can take part also in other compositions. The fact that a component can be known from its interfaces gives ground for defining the composition of component-based application through an *Architecture Description Language* (ADL); this is a language that defines a component-based application from the components involved in a composition and their connections through interfaces. This means of creating an application by assembling preexisting components by connecting their interfaces in an appropriate way is known also as *component-oriented* programming.

Several component models have been proposed. A component model defines the kinds of interactions that are allowed between components and general rules for performing composition. Implementations of component models support the execution of components that conform to that model. Industrial component models include Microsoft .NET[net], Sun Enterprise Java Beans (EJB)[ejb], the Common Component Architecture (CCA)[cca], and the Corba Component Model (CCM) [ccm], while research initiatives include SOFA[sof], Fractal[BCL$^+$06] and GCM[BCD$^+$09]. In particular GCM shows very interesting properties like hierarchical composition, reconfiguration capabilities, both also present in Fractal, and distributed deployment which are useful in our work. A proper background about Fractal is given in Section 6.1, while GCM is described in Section 6.2.

The component-oriented paradigm is usually compared with object-orientation. In fact both approaches share several common goals as modularity, encapsulation and reuse. Their use however can be distinguished. Component-oriented architecture focus on the separation of concerns and enforces stronger encapsulation when developing and application; this allows a higher level of abstraction than what is usually achieved with objects which are usually more fine-grained.

A simple example of a component-based solution for designing a Vacation Planner application can be seen in Figure 2.4 using a common UML notation. The interfaces are indicated on the side of the components and use different shapes to indicate if the component "depends" on a connection to another component, or if it "offers" connectivity to another component. In practice, component models define custom names an notation for each case. For example, the "Hotel Reservation" component offers an interface called *IHotel* to allow access to its functionality, and it depends on two additional components "Room Management" and "Credit Card Billing" to accomplish its task, however the only way the "Hotel Reservation" component can interact with those components is through the *IRoom* and *ICCBill* interfaces. In fact, no implementation logic is mentioned in this example and, from the point of view of the composition, any component that offers a *ICCBill* interface could replace the "Credit Card Billing" component. This example illustrates an application composed of several components that could have been independently developed, where all the information they expose is about the interfaces they offer, and the interfaces they depend on.

In some aspects, components can be seen as a natural implementation for services, as they naturally address some of the principles mentioned in Section 2.1.1. In fact, the interfaces exposed by components can provide a ground for Service Contracts and support discoverability; the

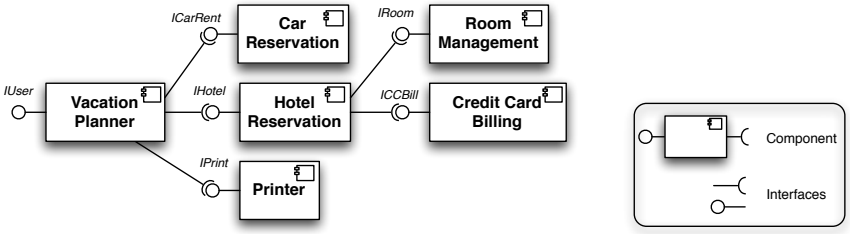


Figure 2.4: A component-based vacation planner application, showing multiple components connected through their interfaces.

capability of components to be composable is derived from the "separation of concerns" approach of both service and component-orientation; and the encapsulation characteristic of components promotes loosely coupled design, abstraction and reusability.

However, a proper component-based solution must provide additional functionalities in order to provide a complete SOA solution. For example, management capabilities are not uniform through the different component models; SOA environments are dynamic, meaning that the architectural composition of an SOA-based solution may evolve over time and this requires some level of runtime adaptation in the composition; the challenges of autonomic services in SOA requires that additional logic be inserted in a component without violating their encapsulation. Finally, concerns like monitoring and management identified through all the levels in an SOA implementation need to be addressed.

During this thesis we take profit of a component-based approach to support several concerns in a service-oriented application. In particular we base our proposition in a component model that provides a reconfigurable component architecture to deliver flexible monitoring and management features, supporting autonomic features in a component-based service.

## 2.4 Service Component Architecture: SOA+CBSE=SCA

The convergence of service-orientation needs and component-based approaches can be traced to the Gravity project [CH04], in which a component model is proposed to provide dynamic availability, and allowing autonomous reactions on services. The combination of component-based approaches to service-orientation is based in some principles highlighting the fact that services are characterized by a contract, and components implement a contract; services provided specific functionalities that can be reused, which is also the cases for components, that can be completely replaced by another that follows the same contract. This convergence between the requirements for SOA and the features provided by component-based solutions gave rise to the initiative called Service Component Architecture (SCA) [OSO07b, Cha07, MR09].

SCA is a set of specifications which describe a model for building applications and systems using a Service-Oriented Architecture. SCA follows the service-orientation idea that a business function is provided as a set of services, which are assembled together to create a service-based solution. Applications built as a set of services, called composite applications, can include both new services created specifically for the application, and also business functions from existing applications that are reused as part of the composition. SCA provides a model both for the composition of services and for the creation of service components, including the reuse of existing services.

SCA applies the ideas of Component-based Software Engineering to the design of SOAs, where, not surprisingly, components called *SCA Components* are the basic units of construction, and take advantage of encapsulation, composition and description. SCA defines those elements in a technologically-agnostic way, that allows to separate the implementation of individual services and communication protocols from the architectural description of the application, enforc-

ing the construction of service-based applications that can rely on heterogeneous technologies.

An *SCA Component* encapsulates the implementation of a service and makes it available through clearly specified interfaces called *SCA Service*s. An SCA Service is, thus, the access point to the functionality provided by the SCA Component. At the same time an SCA Component expresses the dependencies on other services as *SCA Reference*s. An SCA Reference acts as a service that an SCA Component may call. Both SCA Services and SCA References are defined in terms of groups of operations inside an *Interface*. SCA Components provide a mechanism to configure an implementation externally, through *SCA Properties*.

The implementation of an SCA Component can be done using any technology that allows to relate the service offered by the SCA Component to a concrete implementation. Some implementation technologies used by SCA Components are traditional programming languages like Java and C++, workflow languages like BPEL, or scripting languages like PHP and JavaScript.

Composition is realized by connecting SCA Components through their interfaces. SCA Services and SCA References are connected by *SCA Wires*. The mechanism by which clients can call and use services may be specified using *SCA Bindings*. Targets of SCA Bindings, called binding types, can be, for example, services exposed by Web Services or JMS, other SCA Services. SCA provides an extensibility mechanism where additional binding types can be defined.

Composition can also be realized in a *hierarchical* way. SCA defines a hierarchical model that includes *SCA Composites*. SCA Composites allow to assemble SCA elements in logical groupings containing a set of SCA Components, SCA Services, SCA References, and SCA Wires that interconnect them. SCA Composites can be treated as regular SCA Components, meaning that they can be wired to other SCA Components which can also be composites, and they can be included in other SCA Composites. The SCA Services and SCA References used by a composite are exposed by *promoting* the interfaces of their internal SCA Components.

A summary picture illustrating the definitions presented using the standard notation for SCA is shown in Figure 2.5

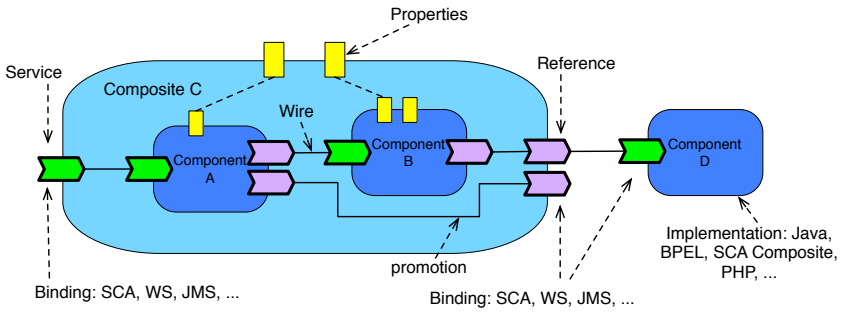


Figure 2.5: An SCA Composite including two SCA Components

The assembly of an SCA application can be described using an XML-based ADL file, sometimes referred to as Service Component Definition Language (SCDL).

Different SCA Runtime implementations have been developed. An SCA Runtime provides the support to create, deploy and execute application based on the SCA specifications. Some SCA runtime environments include IBM WebSphere Application Server [IBM], Fabric3 [fab], Apache Tuscany [tus], Paremus Service Fabric [par], and FraSCAti [SMF+09].

The SCA specifications allow the association of non-functional properties to SCA component using the notion of *SCA Policy* [OSO07a]. An SCA Policy is a statement that controls or constrains some non-functional property on the SCA applications and that must be enforced by the SCA Runtime. This capability has been added mainly to include properties related to authentication, confidentiality, integrity, message reliability and transaction. However, SOAs usually require additional non-functional properties like monitoring, reconfiguration and adaptability that may require more complex descriptions than those that are possible by relying solely on SCA Policies.

The structured assembly model of SCA allows to develop a service-based application following a component-oriented approach, easing reuse of services, with independence of their implementation technology or communication protocol. This separation makes the application more adaptable to heterogeneous service providers. At the same time, the specific interfaces, implementation and protocols are established in the SCDL composition description file. A modification in the architecture of the service-based application, or in the communication technology can be realized by modifying the SCDL file in an independent way from the service implementation.

The major drawback in SCA is the lack of support for dynamic evolution. The SCDL file is a static design-time construct, and cannot be modified during the execution of the application. Consequently any modification in the composition of the service-based application requires to stop the complete application, and restart it using the new composition. Dynamic reconfiguration and other non-functional aspects are not addressed by the SCA specifications, so it is up to the SCA Runtime implementations to provide such features if it is required.

In this thesis we promote the idea of a component-based design for providing monitoring and management features to a component-based service-oriented application. In order to provide a generic approach, we rely on the SCA specification to describe the architecture of our framework, however our approach also involves the runtime modification of the composition, and so it requires the support of an SCA platform that provides such dynamicity.

One example of such reconfigurable SCA platform is the FraSCAti platform, described in Section 3.1.4.2, which bases on the Fractal specification to add reconfiguration capabilities to an SCA-based application. In our work, however we rely on an alternative implementation of the SCA specification, based in the GCM model (Section 6.2) which enforces a more clean separation of concerns by leveraging a component-oriented approach, and which can also handle distributed deployment concerns.

## 2.5 Autonomic Computing

The *Autonomic Computing* initiative was pushed by IBM as a response to the ever more increasing complexity in the maintenance of computing systems [Hor01]. The motivation grounded in the difficulty of managing systems that span heterogeneous environments and extend beyond the boundaries of single companies. The abilities required for installing, configuring, optimizing such systems become too complex for system integrators and managers, and it becomes even more difficult to deliver decisive responses in a timely way. Let alone that all these activities are commonly not even part of the main functional objective of the system, but they are anyway crucial for obtaining an adequate respone.

The vision of autonomic computing based on the idea of self-governing systems. This is, systems that can manage themselves given high-level objectives from an administrator [KC03]. The inspiration is that of the autonomic nervous system of humans beings, which governs issues like heart rate, body temperature and respiration, freeing the brain from dealing with those low-level, yet vital, activities, and concentrate in other tasks. The situation is similar to complex computing systems. Management tasks like installation, configuration, protection, optimization, are not the main functional objective of most applications, but if they are not properly addressed, the application cannot accomplish its task.

The proposition, then, is to enable self-governing systems that take control of all these non-functional tasks, letting more space for the application and consequently for the developers, to concentrate on the main functional goals. However, the issue of providing self-governance to applications is not an easy task. In order to really free developers from the burden of programming self-governing features on their applications, there must exist a way to develop these concerns independently and to integrate them with the application at some stage.

### 2.5.1 Autonomic Control Loop

The characteristics of self-governing or, more commonly named, self-managing systems usually include [Hor01]:

- The system must be able to know itself and its surroundings. This means that the system must be aware and must be able to collect information about its own functioning and about the environment.
- The system must be able to modify itself. This implies that the system must be able to produce an effect in its own configuration.
- The system must be able to react to varying and unpredictable conditions. This means that the system must be able to analyze the information it has and decide upon some course of action in a timely way.

These activities are not static. They must continually repeat themselves and be re-evaluated during all the functional activity of the system. The implementation of this behaviour takes the form of *feedback control loops*, in which typically four canonical activities can be identified [MKS09]: *collect*, *analyse*, *decide*, and *act*, which are organized as shown in Figure 2.6

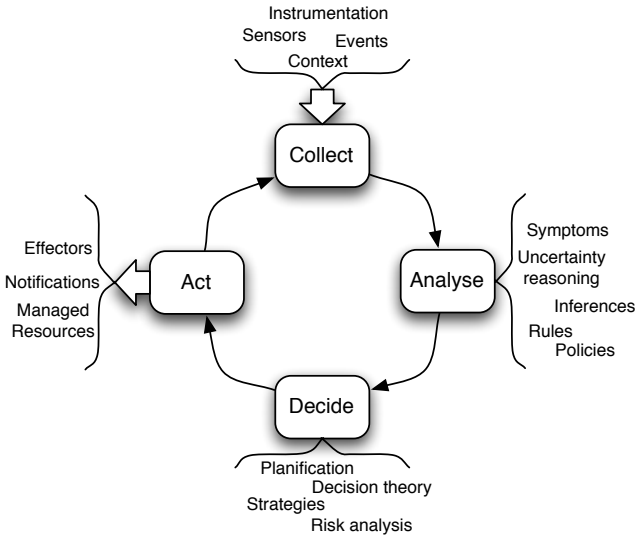


Figure 2.6: Main activities of the feedback control loop [DDF+06]

In the feedback control loop, a set of sensors or probes, *collect* data from the executing process and its context. The collected data is processed or filtered and possibly stored in order to reuse it later. A diagnosis engine *analyses* the collected data to infer trends or identify relevant symptoms. Next, a planning engine attempts to predict the future behaviour of the system and *decides* on how to *act* on the executing process and its context through effectors. The new configuration of the system produces new data that is *collected*, closing the feedback control loop.

The generic model for the feedback control loop involves several questions that must be solved when implementing such a system [MKS09]:

- **Collect**. Determine what kinds of data must be collected, and which are the sources of such data. In which format will the collected data will be represented, and at which rate it will be collected. The sources of data may be a fixed set, or they may vary at runtime.
- **Analyse**. Determine which algorithm or diagnosis technique will be used. How the state of the system is represented from the collected data, and how it relates to the objective state.

- **Decide**. Determine how far is the current state of the system from the desired objective state. What algorithms can be used for planning the transition to the objective state. How can multiple control loops be organized and avoid interfering.
- **Act**. Determine which are the managed elements, and how can they be manipulated. Which techniques are used, f.e., parameter tunning or architectural modifications. The changes to perform may be predefined or dynamically generated.

IBM proposed a reference architecture for autonomic computing [IBM06], in which the central element is called *autonomic manager*. An autonomic manager implements the autonomic behaviour by providing an *autonomic control loop* that implements the activities defined in the generic feedback control loop. The phases of the autonomic control loop are defined as: *Monitor*, *Analyse*, *Plan*, and *Execute*, and they are usually referred to as the *MAPE* autonomic control loop. Sometimes also a common element called *Knowledge Source* is highlighted, which represents the management data that can be shared for all the phases, and the loop is called *MAPE-K* autonomic control loop. The main activities of each phase are shown in Figure 2.7 and include:

- **Monitor**. Provides the mechanisms to collect, aggregate, filter and report monitoring data collected from a managed resource through *sensors*.
- **Analyse**. Provides the mechanisms that correlate and model complex situations and allow the autonomic manager to interpret the environment, predict future situations, and diagnose the current state of the system.
- **Plan**. Provides the mechanisms that construct the actions needed to achieve a certain goal, usually according to some guiding policy or strategy.
- **Execute**. Provides the mechanisms to control the execution of the plan over the managed resources by means of *effectors*.

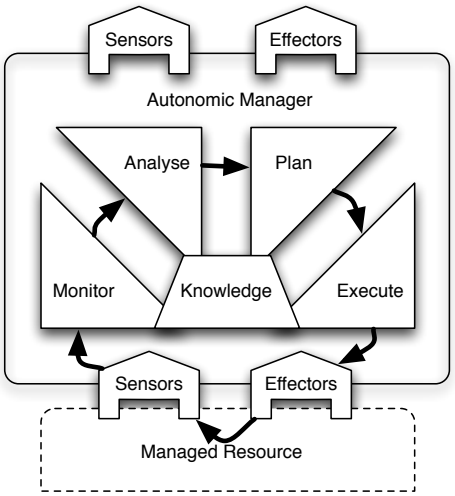


Figure 2.7: Autonomic Control Loop [IBM06]

The design of the autonomic manager considers the possibility of arranging several autonomic managers in a hierarchical architecture [MKS09]. This kind of architecture has the goal of making the autonomic behaviour more scalable and supporting different autonomic control loops. In such setting, some autonomic managers will be located in a low-level, near the managed resource and will control short term goals like limiting the concurrency, or tuning some parameters of the managed resource; while higher-level autonomic managers will have a global view of the system and handle long term goals like performing load balancing on multiple resources or execute modifications in the architecture of the managed system.

One possible setting for the reference architecture can be exemplified in Figure 2.8. In this setting the architecture is arranged in three layers where a high-level orchestrating manager controls a set of intermediate resource managers, where each one of them may control a different aspect of self-management (like optimization, security, healing) and may have at its command a set of managed resources. Each managed resource may implement a local autonomic control loop, or may just offer interaction through its sensors and effectors. At the same time a manual (human manager) may intervene at any level of the hierarchy through a management console, and all the layers share a common knowledge base upon which to take their decision. The interesting part of this hierarchical setting is that all the communication between the different autonomic managers is enacted only through their sensors and actuator interfaces.

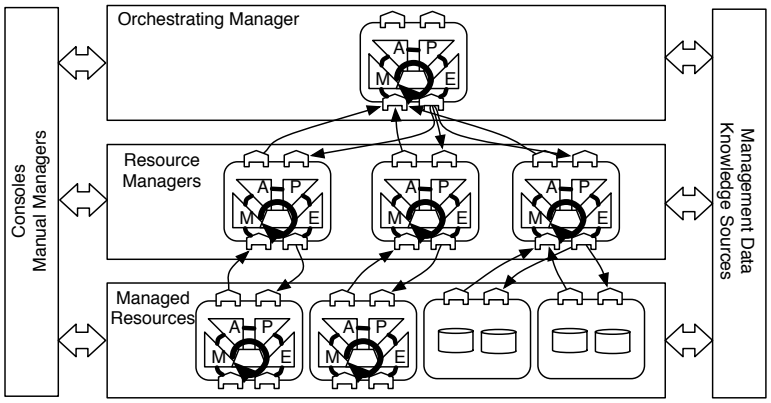


Figure 2.8: An example for Autonomic Computing Reference Architecture [MKS09]

## 2.5.2 Self-Adaptability and Self-* Properties

Self-management is tightly related to the term *self-adaptation*, and in many cases both terms can be used interchangeably [ST09]. Self-management refers to the capability of the system to know itself and its environment, and modify itself as a reaction to internal or external conditions. We will refer to *self-adaptation* as the more general behaviour of a system of modifying internal characteristics of itself in order to comply with certain goals. Of course, self-adaptation to varying conditions can only be obtained by properly integrating self-managing capabilities.

Self-management properties are usually expressed as a set of characteristics called *self-* properties*, depending on the adaptivity goal they target. Salehie and Tahvildari [ST09] describe some of this self-* properties in a hierarchical arrangement shown in Figure 2.9.

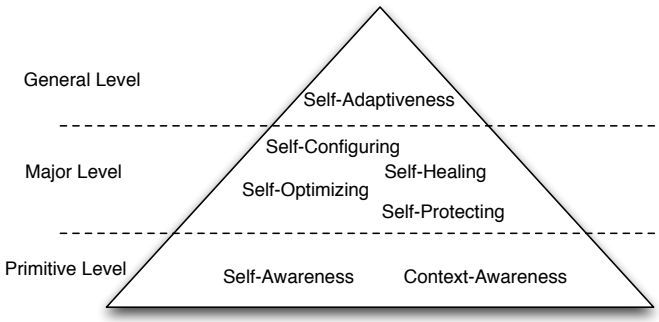


Figure 2.9: A hierarchical view of self-* properties [ST09].

In this hierarchical view, the *General Level* refers to the global objective of self-adaptivity, which can encompass several properties referred to as self-maintenance, self-control, self-evalua-

tion, or self-organization, depending on the major goals they try to achieve. The properties grouped in the *Major Level* are the main ones introduced by IBM [KC03] and form the basis for achieving more general goals:

- *Self-Configuring* refers to the capability of reconfiguring automatically and dynamically in response to internal or environmental changes by installing, updating, integrating, and composing or decomposing elements.
- *Self-Healing* is the capability of discovering, diagnosing and reacting to disruptions. It can also be tackled by anticipating potential problems in order to prevent failures. Self-Healing may be decomposed in tasks like self-diagnosing and self-repairing.
- *Self-Optimizing*, also referred to as self-tuning or self-adjusting, refers to the capability of managing the performance or resource utilization of the system in order to satisfy the requirements of different users. Internal goals may be related to end-to-end response time, throughput, utilization and workload.
- *Self-Protecting* is the capability of detecting potential threats to the system, f.e., by identifying hostile behaviours, and taking actions to avoid them or to mitigate their effects.

Finally, the properties mentioned in the *Primitive Level* are expected to be those that allow the ones of the level above to be effectively implemented. Concretely they refer to the capability of being aware of its own state (*self-awareness*), and to get information of the environments (*context-awareness*). Both properties are the basis for taking decisions with respect to any of the properties of the level above.

### 2.5.3 Autonomicity and Services

As mentioned in Section 2.1.4, one of the challenges in service orientation is to provide autonomicity to service-oriented applications in order to be able to adapt to changing conditions in such environments, and resolve problems with minimal human intervention.

Autonomic support for services can provide *self-configurable* services that can configure themselves and adjust to different environments; *self-healing* services that can react to disruptions in their own functioning, or in their dependencies, and can take corrective actions to avoid or alleviate major disruptions; *self-optimizing* services can tune themselves to better adjust to the end-user needs, or optimize their resource utilization; and *self-protecting* services can detect hostile activities (like access violations, or denial-of-service attacks) and take corrective actions to reduce their vulnerability.

In Section 2.1 we characterized service-based applications as being highly **dynamic** environments, and exhibiting **heterogeneity** in terms of implementing technologies, providers and management capabilities. Naturally in such environments it becomes hard to correctly manage all the possible situations that can arise: the sudden unavailability of a service, a change in its QoS characteristics, the availability of new services with other QoS characteristics, the influence of a modification in one service on the overall composition, the handling of potentially conflicting decisions. Autonomicity seems to be an appropriate approach to handling such tasks.

When introducing autonomic behaviours to service-based applications, it is also possible to use different levels of granularity (referring to the model of Figure 2.2). At the level of Service Foundations, autonomicity can be seen as the capability of a service to adjust to its individual requirements by modifying some parameters or adjusting its resource consumption on the infrastructure where it is hosted. When considering the level of Service Composition, adaptability can take care of the compliance of the whole composition with respect to some goals, and may dynamically trigger a reconfiguration of the composition in order to adapt to some condition. Finally, in the Service Management level, adaptability may be used to ensure SLA compliance according to some agreed goals.

## 2.6 Summary

In this chapter we have presented the several areas around which our contribution is related.

We have introduced the subject of service-orientation, which drives the development of many current applications, in a situation where the construction of new applications is often guided by the reutilization and intelligent composition of several existent services developed by independent parties, to provide added-value services.

In this context, the infrastructure support for running such applications has been changing from locally hosted "on-premises" infrastructure to highly available and dynamically scalable infrastructures as Clouds, whose capabilities are offered "as a Service". By relying in Cloud services, providers can offer their services, and free themselves of infrastructure maintenance costs. At the same time a proper management of the elasticity of Cloud infrastructures can help in the adaptation requirements of possibly autonomic services.

The development of SOAs has taken many approaches. Perhaps one of the most industrially supported of them has been the SCA model which uses properties of CBSE to drive the design of service-based applications and provide an structured way to compose them, independently from specific implementation technologies. We use this same support for describing the design of our framework in Chapter 5, while we also consider runtime modifications, which are not included in the main SCA specifications.

Finally, we have pointed out the usefulness of the ideas from Autonomic Computing to address the challenge of having more adaptable services that can dynamically modify themselves to the user needs and can target SLA compliance. In particular, the implementation of an autonomic feedback loop can help services to implement self-* properties.

Our work takes ideas from these areas to propose a component-based framework that can take in charge the adaptability task in services-based applications. We propose separate components for implementing each of the tasks involved in an autonomic control loop. The objective behind this separation of concerns is to better adapt to the management needs of a service-based application by attaching the autonomic control loops close to the services that needs to be managed. The component-based approach helps to structure the communication between the different phases, and allows to modify the logic of each phase to facilitate the adaptation of the autonomic behaviour as it is needed. The framework aims to consider all the levels of service-based application, allowing to introduce concerns related to the composition of the services and to lower level infrastructure details, by leveraging these concerns to a common level where they can be uniformly managed.

However, a broad body of work has been in the different areas that comprise an autonomic control loop, in particular by applying some of these tasks to service orientation and resource management. In the next chapter, we describe more deeply the distinct phases of the autonomic control loop and the current works that exist around them. Section 3.1 describes challenges specific to each phase, and present some works that address these challenges. Then, in Section 3.2.2 we describe a set of frameworks and tools that have been developed to provide autonomic behaviour to applications by implementing autonomic control loops, including those that have been applied to service-based applications, whose ideas have inspired our work.

# 3 State of the Art

## Contents



This chapter presents the state of the art with respect to the different phases of the autonomic control loop. The analysis is separated in two parts: section 3.1 describes advances in each of the involved phases: monitoring, analysis, planning, and execution, and mentions research works and tools whose main objective fits in one of the steps that we want to enable through our framework. Section 3.2 identifies works that integrate some or all of the concerns of the autonomic control loop and that does not fit in only one of them. Finally, in Section 3.3, we compare the analyzed frameworks and tools in terms of the features that we have identified as important.

## 3.1 Phases in the autonomic control loop

We present advances in the different steps that conform the autonomic control loop. Namely we separate the description through each one of the phases of the autonomic control loop: monitoring, analysis, planning, and execution. It is necessary to mention that, while we have established a separation between the phases of the autonomic control loop, several of the works analyzed have not been developed with that approach in mind and consequently can not be fitted only in one category and some of them present overlapping features with other phases. However, we have focused the analysis in the most prominent features of each work, and in those that are more relevant for our work.

### 3.1.1 Monitoring

Monitoring is a very broad subject, which has received a lot of attention from the community. Gathering data from a running program provides valuable feedback about the behaviour of the program and may help to find points for introducing improvements.

However, monitoring does not come for free. A monitoring system requires some degree of intrusiveness on the application that is going to be monitored. Monitoring tools usually must make a trade-off between the intrusiveness they have on the application, and the precision of the data they gather. A highly intrusive approach may provide precise data about the program, but may introduce undesired side effects that distort the results. On the other hand, a monitoring approach that is too separated from the application may provide only vague information about the program execution.

Monitoring techniques and tools can be analyzed from several points of view: intrusiveness, dynamicity, flexibility, scalability, or the types of applications they target. In the following we describe some tools and monitoring techniques from the point of view of a running application.

#### 3.1.1.1 Profiling tools

Data gathering techniques were initially designed for single monolithic applications, and have the objective of *profiling* an application, this is, obtaining runtime information that characterizes the behaviour of the application, and help to introduce optimizations.

Profiling may use several techniques for gathering data:

- **Manual Instrumentation**. Manually introducing monitoring code into a program is one of the most simple and direct ways to obtain information about the runtime behaviour of a program. However, manually introducing monitoring code inside an application is highly error-prone, intrusive, and impractical in complex applications as it requires a deep knowledge about the functionality of the application.
- **Compiler-based Instrumentation**. Several compilers allow to introduce profiling code during compilation. While this task is less error-prone, it is limited by the profiling features provided by the compiler. Once the program has been compiled with profiling features enabled, it can be executed using an external tool that captures the profiled information and displays it. This is the approach used by GNU gprof [Fen].
- **Interception-based Instrumentation**. Languages that can be interpreted or executed in virtual environments, like Java, .NET or Python, provide instrumentation at the virtual environment level, and allow to use hooks. Hooks are special points in the execution where an event is generated or some monitoring code can be attached. This technique is more dynamic and less intrusive as it does not require changes on the functional code, while it also allows to program separately the monitoring code. In some cases the hooks are predefined by the virtual environment. This approach is used by tools like JVMTI (JVM Tools Interface) [Ora], .NET, and the Python profiling module.
  A more flexible approach is that used by aspect-orientation [KLM+97]. The programmer can define *pointcuts* as points where the execution flow can be intercepted by an *aspect* code. The *aspect* is programmed separately and weaved into the application code at compile time. This approach allows to introduce custom monitoring code in an independent way, although this flexibility is only limited at compile time.
- **Statistical Profiling**. Statistical profilers use regular sampling at the hardware and/or operating system level, while leaving the binary code untouched. By probing the state of the hardware or kernel indicators, they can provide an approximate view of the behaviour of the application. The precision of the data gathered may depend on the frequency of the sampling, though intrusiveness is still very low. The flexibility of this technique is limited as the information obtained is mostly general and hardly associable with specific parts of the application. Examples of this tools are Intel VTune [Int], AMD CodeAnalyst [AMD] and Apple Shark [App].

Depending on the kind of the data gathered, information may be displayed in several forms: a call graph, detailing dependencies between method calls; statistical summaries, detailing resource consumption per method or unit of computation; or an execution trace, detailing sequence of method calls and performance related information.

#### 3.1.1.2 Monitoring Distributed Applications

As the complexity of applications increased and distributed applications became common, more extensible, flexible, efficient and less intrusive approaches were required. The monitoring task became more complex as the execution is not limited only to a single executing entity.

In distributed applications, dependencies between sub-parts of the application spread over possible large geographical areas, introducing sources of bottlenecks in network interfaces and communication issues. The complexity of these systems, which may be composed of heterogeneous technologies, required higher-level tools (that is, at the level of the business operations) to analyze the performance of such environments, and to check that the behaviour of the application is the intended one.

**Autopilot** Autopilot [RVSR98] defines a model where sensors can be attached to an application and registered in an external registry. Using the registry, sensors can be contacted by a client, and provide data to them via events. Autopilot is one of the first works to separate the monitoring task from the rest of a more global adaptation process, which is better described in Section 3.2.2.1. The separation of sensors, that work as data providers, and clients, that work as data consumers was used as inspiration for later grid-oriented monitoring models.

**ARM** ARM (Application Response Measurement) [Joh04] is an open standard published by The Open Group[1] for monitoring and diagnosing performance bottlenecks within complex enterprise applications that use loosely-coupled designs. ARM defines a simple API that applications, or the middleware where they run, can use to transfer information about their transactions to an external management software. Implementations of the API have been provided for Java and C/C++.

ARM works by manually introducing API calls in the target application. This step defines what transactions will be monitored. The API calls are implemented by a local ARM agent, which can be accessed from a management application. The management application connects to the agents, and can query information from them to perform analysis, correlation of calls, or provide graphical views. This allows to build traces and end-to-end call graphs that help to locate failures or bottlenecks in the application. The architecture of client/server application instrumented with ARM is shown in Figure 3.3.

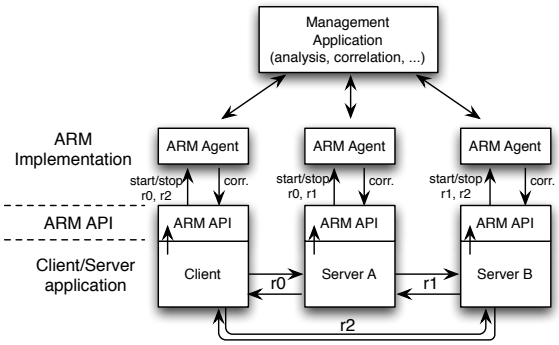


Figure 3.1: ARM monitoring architecture in a client/server application [Joh04]. Clients and server are instrumented with ARM API calls, and their associated *ARM Agents* perform the monitoring task, which can be used by management application.

Applications like the Apache HTTP Server and IBM Websphere Application Server are instrumented with ARM standard calls, meaning that it is possible to monitor them through ARM agents.

[1] http://www.opengroup.org/

**Magpie** Magpie [BDIM04] is a toolchain for automatically obtaining the workload of a system under realistic operating conditions. Magpie does not introduce additional instrumentation in the application, but relies on events generated by the Event Tracing for Windows logging infrastructure that works at the kernel level. During the service of a request, events generated by this infrastructure are detected and correlated to trace the path of requests and obtain performance measures. This way, a request is described according to its path taken through all the system components, and their resource consumption.

**Pinpoint** Pinpoint [CKF+02] is a tool for problem determination in distributed applications. Pinpoint performs middleware instrumentation to introduce and propagate unique identifiers through the path of a request, and packet sniffing to detect and correlate network communication. Using the collected information, a data clustering analysis engine deduces faulty or poor-performing components. Pinpoint has been implemented by instrumenting the J2EE platform.

#### 3.1.1.3 Monitoring in Grid Platforms

Distributed applications with high performance needs require powerful computing infrastructures which are made available through computational Grids. Grid infrastructures comprise hardware, software, and knowledge resources, and aims to "enable resource sharing and coordinated problem solving in dynamic, multi-institutional virtual organizations" [FKT01]. As a consequence, one of the characteristics mentioned of Grid platforms, is that they "deliver nontrivial qualities of service" [Fos02], referring to the fact that the constituent resources may be managed to satisfy complex user demands and several aspects of quality of service like performance, availability, security, and co-allocation of different resources.

Execution on a computational Grid requires to monitor not only the performance of the distributed application, but also the involved grid resources as they influence the overall performance. Moreover, as Grid technologies span through heterogeneous Virtual Organizations (VO), it becomes common that the grid resources are not completely under the control of the application programmer, so it is not always possible for the programmer to instrument the platform over which the application runs by using the tools described in the previous section. Instead, grid users are commonly tied to specific grid software or tools.

In this context, Grid providers usually provide some level of resource monitoring that is made available for Grid users. Grid monitoring tools must be able to handle scalability and efficiency issues, as the amount of sources of monitoring data and the monitoring data itself increases rapidly. Several grid monitoring tools have been proposed [ZS05]. Here we mention some of them where the underlying ideas, in particular about scalability, are relevant for our work.

**Globus MDS** Globus MDS (Monitoring & Discovery System) [All, CFFK01] is part of the Globus Toolkit for grids. Globus considers elements that acts as sensors, called *Information Providers*; elements that aggregate information from sensors and produce composed data, called *Grid Resource Information Services* (GRIS); elements that aggregate different GRIS and allow to form a hierarchy, called *Grid Index Information Services* (GIIS). The GRIS elements, by aggregating different monitoring information providers, are able to federate multiple overlapping Virtual Organizations (VOs), which are uniformly managed by GIIS elements.

The hierarchy of sensors, GRIS and GIIS communicates through two defined protocols for transmitting and discovering monitoring information. The *Grid Information Protocol* (GRIP) is used to discover (search) for monitoring data on some part of the hierarchy, and to recover (lookup) data according to some criteria on a specific provider. The *Grid Registration Protocol* (GRRP) is a notification mechanism through the different elements become aware of each other. The communication protocols GRIP and GRRP, and the information model used to query and retrieve monitoring data have been implemented using OpenLDAP.

Through this hierarchical approach, Globus MDS allows to collect monitoring data from multiple VOs in an uniform and scalable way. We remark that MDS does not define the metrics and sensors that are going to be available, but provides an architecture for discovering and collecting them efficiently.

MDS has been superseded in the last version of the Globus Toolkit by MDS4 which recreates the hierarchical approach using Web Services technologies and interoperable standards for propagating monitored information, and it is better described in Section 3.2.1.8. Nevertheless, the monitoring architecture originally presented by MDS has served as a basis for later developments in scalable grid monitoring architectures.

**SCALEA-G** SCALEA-G [TF04, TSF06] is a unified performance monitoring framework for grids resoures and applications. SCALEA-G provides a peer-to-peer Grid infrastructure of *Sensor Manager Service* that collect and store monitoring data from sensors. Sensors can be system or application level, and can interface with existing monitoring services via plugins, like Globus MDS [All], Ganglia [MCC04], or Nagios [nag]. The Sensor Manager Service can be used by clients to subscribe and query for monitoring data, which is represented using an XML language. Sensors Manager Services are able to locate each using a Registry Service. SCALEA-G is based on the Web Service Resource Framework (WSRF). SCALEA-G attempts to solve the scalability problems by a plain peer-to-peer approach instead of hierarchy.

**Ganglia** Ganglia [MCC04] uses a hierarchical design targeted at federations of clusters to achieve scalability. Ganglia defines two daemons: `gmond`, targeted to single clusters, and `gmetad` used to federate different clusters and forming a hierarchy. Monitored data is transmitted using XDR (eXternal Data Representation), an IEFT standard to transmit data through heterogeneous architectures. Clients use a command line program to access the monitoring data and a set of front-ends are available to provide visualization. Their scalable approach has been widely used in several high performance environments [Gan].

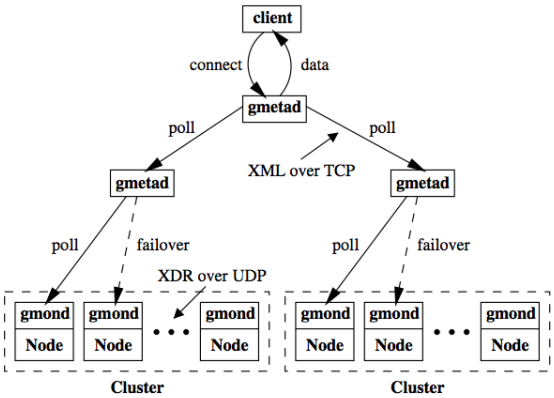


Figure 3.2: Ganglia architecture [MCC04]

**GMA** The Global Grid Forum formalised the ideas behind several grid monitoring tools in the Grid Monitoring Architecture (GMA) [TAG+02]. GMA identifies three kind of components: producers or event sources, which produce monitoring data and makes them available; consumers, or event sinks, which requires and consumes monitoring data; and directory services, which register producers and allows to discover them, working as a lookup service. The model is shown in Figure 3.3. Moreover, the model allows that some components act as producers and consumers by implementing the appropriate interface, and taking the role of intermediaries. These intermediaries can be used to filter monitoring data and lower the load on producers.

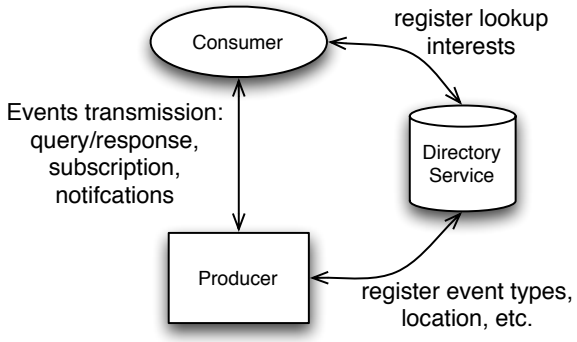


Figure 3.3: Components of the Grid Monitoring Architecture

Implementations of monitoring architectures for grids like GridRM [BS03], uses the GMA to provide an extensible tool for managing grid resources. GridRM uses a set of Java-based gateways that collects and normalises events in order to provide a uniform access to different data sources.

In general, grid monitoring systems are oriented to monitor resource consumption and focus on scalability and efficient communication and aggregation of monitoring data. These concerns are addressed mostly by using intermediate republishers of monitoring data that act as filters and reduce communication, following the approach of the GMA model [ZS05].

There is a high diversity of APIs and it is hard to find points for interoperability between the different platforms, as the grid platforms are usually hosted by institutions for specific purposes and provide different tools for using them. These characteristics make difficult to concretize the idea of a unique global Grid. Consequently, monitoring approaches are specialized to the targeted Grid infrastructure.

#### 3.1.1.4 Monitoring of Cloud Platforms

On recent times, *Cloud Computing* has emerged as a common term that seemingly, and not completely erroneously, shares several visions and challenges with Grid platforms. The evolution of Cloud platforms has come from a shift in focus from a platform that provides an infrastructure for storage and computational resources, to one that aims to deliver more abstract resources and services [FZRL08].

Cloud platforms include features like *virtualization* and *dynamic elasticity* that introduces challenges in the monitoring of the resources provided by the Cloud platform.

Virtualization introduces more granularity at the level of the computing resources available. Cloud platforms offer different levels of services to the end user, which is exposed only to a predefined API, and normally the underlying lower level resources are opaque to the users, so that tracking a problem down through this software and hardware stack might be more difficult.

Monitoring approaches for Cloud platforms that delivers services, that we refer in the following as Service Clouds, must be aware of virtualization of resources. In fact, monitoring is required as part of the Cloud platform to be able to provide dynamic elasticity and perform load balancing between instances. Moreover, as Cloud resources are commonly delivered in a pay-as-you-go model, end users will be willing to have detailed monitoring information about their resource usage.

With the advent of multi-clouds hosting of services, commercial monitoring tools have begun to target federated cloud environments, where the tools must aggregate metrics from different cloud providers, and be able to normalize values and provide an abstract view of the Cloud environment.

**CloudStatus** CloudStatus [Hyp] is a Cloud monitoring tool provided by Hyperic, that is able to collect and aggregate monitoring metrics from services hosted in different cloud environments. By normalizing and aggregating metrics, CloudStatus is able to provide a unified view of the health of the Cloud hosted service, and helps to determine causes for performance problems. The set of monitoring metrics can span from overall cloud performance to individual service-specific metrics. Currently, CloudStatus is able to monitor Amazon Web Services and Google App Engine cloud providers.

**LogicMonitor** LogicMonitor [Log] is a monitoring tool for highly distributed applications that, among others, is able to handle monitoring of virtualized resources and custom metrics. LogicMonitor uses a set of lightweight agents deployed in each service that must be monitored. They have developed an extensive set of agents for several common cloud platforms and hosting servers. Agents communicates their collection results to a central LogicMonitor server, communicating by the 443 SSL port which is usually available through most firewalls. The LogicMonitor servers are hosted in their own highly available platform, where they are analyzed an exposed to the end-user.

**Lattice** Lattice [CGM10, CGC+10] is a framework to monitor virtualized resources on Cloud environments. Lattice is defined around the concept of producers and consumers, so that data sources (probes) act as producers of data, and monitoring elements act as consumers. Moreover, a third element called communicator is introduced, which works as a monitoring information delivery service, so that the communication channel between probes and monitors can be changed without affecting them. The ability to monitor several levels of the Service Cloud environment is also considered. Data sources are implemented to provide information from the infrastructure level (the physical hosts), the Virtual Execution Environments deployed on each host, and the services that are available on the Service Cloud. Lattice is able to gather data from federated service clouds by using monitoring brokers on each federated cloud.

#### 3.1.1.5 Monitoring of service-based applications

Service-based applications introduce several concerns for monitoring. Services are developed to accomplish specific tasks, and are expected to satisfy contractual requirements related to Quality of Service (QoS) properties. Monitoring them in an effective way is a key requirement for checking that these contractual goals are accomplished.

Service properties may be identified as *functional* properties, where the objective is to verify if the service delivers the functionality that is expected from him; or *non-functional* properties, related to the QoS of the application, where concerns like availability, security, performance, reliability, cost, energy consumption, and others are of interest [GG07].

Consequently, monitoring concerns for service-based application can be manifold. Concerns brought by service-orientation like dynamicity, loosely coupling and heterogeneity may be present at the service application level, at the service platform or middleware, or at the physical infrastructure level, making the monitoring task more complex.

The need for monitoring service-based applications at runtime has brought a large number of research projects, both from the academic and from the industrial world. Tools like those mentioned for distributed environments usually focus on measuring the performance of resources at an infrastructure level, but seldom at an application level as this is a concern of the programmer of the application. They also are developed for specific implementing technologies, which is hardly extensible or cannot interoperate with heterogeneous providers.

In general, it is not possible to introduce instrumentation directly on services that are provided by external parties. Monitoring tools must rely on features provided by the middleware where the services run, and usually resort to interception of services communication, or detection of events [CS10]. Monitoring tools must also be flexible enough to allow to define metrics

that were not originally considered in the design of the application, or that can be defined in terms or existing metrics.

Service orchestration engines are a suitable place to introduce interception, as they control all the steps of service interaction [MRD08]. Approaches like the proposed by Baresi et al. [BGG04] describe a method to automatically add monitoring components to a BPEL orchestration, using annotations expressed in comments inside the BPEL description. The annotations are pre-processed and transformed into additional components that are included as part of the orchestration. An evolution of the idea of introducing modifications into a BPEL description is taken by the ORQOS platform [BRL07]. ORQOS relies on policies described using the declarative language QoSL4BP [BRL08] and introduces monitoring activities into the BPEL description. Policies can be used also for describing QoS requirements and adaptation actions.

Aspect-oriented approaches have also been proposed; Bianculli and Ghezzi [BG07] describe a solution that uses AspectJ to introduce pointcuts into the open source ActiveBPEL[2] engine, that interact with a conformance Monitor element. This approach avoids manipulating the BPEL description for each monitored process, but harms portability.

As monitoring becomes an orthogonal requirement in service-based applications, dedicated QoS Monitoring infrastructures have been proposed. Zeng et. al [ZLC07] describe such an infrastructure, where most of the QoS metrics are associated to evaluation formulas that can be computed and updated efficiently based on events detected by a BPEL engine.

Given the wide range of properties that can be measured in a service, monitoring frameworks and tools usually are tied to a language that specifies which properties must be monitored. These descriptions are, in some cases, devoted to explicitly specify monitoring requirements, or in other case are the result of a previous negotiation between consumer and providers. In the following we mention some frameworks used to gather properties of running services, focusing on the monitoring aspect.

**Dynamo** Dynamo (Dynamic Monitoring) [GG07] is a toolset that instruments a WS-BPEL process to monitor services interaction. The weaving of monitoring calls takes place at deployment time and is based on a description given in WSCol, an specification language to describe pre- and post-conditions that are expected for each process. Dynamo can collect monitoring data by analyzing the variables mentioned in the services description, or by using external analyzers. After each invocation call, the execution is stopped and the monitoring and analysis take place.

Dynamo enforces separation of concerns by defining separately the conditions that are of interest for the service execution. As the instrumentation takes place at deployment time, it lacks the flexibility required in case that some conditions may change or new monitoring requirements are introduced.

**Cremona** Cremona (Creation and Monitoring of WS-Agreements)[LDK04] is a framework proposed by IBM to monitor functional and non-functional properties of services interactions. The properties to monitor are expressed using the WS-Agreement language [ACD$^+$07]. Cremona provides an "Status Monitor" component that is specific to a service, and it is used to monitor the metrics needed to create an agreement. After an agreement is provided, Cremona uses a "Compliance Monitor" to connect interfaces of a service that provide the monitoring data.

One of the advantages of Cremona is to allow the connection to the monitoring interfaces provided by the target service, by defining plugins, and without the need to instrument the service implementation. This approach allows a finer grained and flexible monitoring approach than what can be obtained by intercepting service interactions. However, the set of measurable properties can only be defined at design time.

**Cilia** Cilia [GPD$^+$10] is a mediation framework designed to facilitate systems integration, by the development of decoupled mediation components that implement the mediation tasks.

[2] http://www.activebpel.org

Cilia follows a component-based approach in which components implement simple tasks for data processing like transformation and filtering, or communication tasks like data gathering and data delivery. The conceptual model presents two kind of components: *Mediator* components that handle the data manipulation tasks, and *Binding* components that provides the communication protocols. Cilia components can be assembled into *mediation chains*, in order to provide a complete mediation between services and data.

Cilia features runtime evolution of mediation architectures. Runtime evolution is supported by a runtime model that is maintained using a technology independent representation. Modifications over the runtime model like the addition/removal and binding/unbinding of mediators are reflected in the real instances. The implementation provided, which supports this dynamic reconfigurations, is supported by iPOJO [EH07, EHL07] components (which are mentioned among the adaptation support technologies in Section 3.1.4.2).

Although the overall objective is to facilitate the integration of service-oriented systems, instead of making the managed system itself more adaptable, Cilia shares our goal of providing a flexible architecture designed for supporting non-functional tasks.

### 3.1.2 SLA Analysis

The Analysis step in the autonomic control loop implies the checking of conditions to verify if the application is complying with certain goals. In this work, we target component-based service-oriented applications, where such goals are stated as conditions expressed inside SLAs. Hence, in this section we mention challenges and advances in automatic analysis of goals in SLAs.

The SLA Analysis involves several challenges. An automated SLA Analysis process requires a **precise** and unambiguous definition of the SLA, that can be checked by an engine. The engine must be able to understand the specification and must be customizable enough to collect the required data, model it in a logical manner and efficiently evaluate the SLA conditions at runtime [SDM01]. Also, the SLA representation must be **generic** enough to represent new conditions, as it is generally not possible to know or anticipate all the possible SLA goals that can be required to enforce, nor it is possible to foresee all the different service providers that may need to be contacted to obtain the monitoring information needed for SLA checking. Also important in our context is to have **flexibility**, so that the set of SLA goals can be modified at runtime; in fact, as we target evolving service-oriented applications, it is natural that the SLA goals can also evolve. The most relevant works we have found about SLA representation are presented in section 3.1.2.1.

Given a proper description of an SLA, the objective is to ensure that the service complies with these goals at runtime. This objective can be addressed in a static way by, f.e., choosing a set of services in a composition whose characteristics comply with the goals stated in the SLA. However, at runtime, environmental conditions can change, and it is necessary to perform runtime *SLA monitoring*, and ensure runtime *SLA compliance*. In this context, it is useful to distinguish the scope of actions taken by the existing works:

- Those approaches that perform *SLA monitoring* attempt to, given an SLA description, build or use an existing runtime monitoring infrastructure to obtain the metrics needed to check the compliance to the SLA and, in case the service does not comply with some goal at runtime, the system triggers a notification indicating that an *SLA violation* has occured. These works are mentioned in section 3.1.2.1.
- Other works attempt to *ensure SLA compliance* at runtime. In this situation runtime SLA monitoring is required but detecting that an SLA violation has happened is not enough. Instead, the objective is to detect the possibility of an SLA violation before it actually happens, and trigger preventive actions in order to avoid the occurrence of the SLA violation. This way the service compliance to the SLA is not disrupted. These kind of efforts, that mostly use statistical predictors and stored history, are presented in section 3.1.2.2.

In the following we mention some works related to the challenges we have mentioned for SLA Analysis.

#### 3.1.2.1 Languages for SLA description and SLA monitoring

In general, SLA languages refer to similar concepts. As they are contracts between service consumers and services providers, they usually contain: (1) a description of the *parties* involved in the contracts, (2) a list of objectives, which we refer to as *Service Level Objectives* (SLO), (3) and a set of *conditions* or constraints that concretely express the SLOs. The actual scope of each concept may vary in each language, but whenever possible, we intend to relate the concepts inside each language to this vocabulary.

Most of the SLA languages proposed are defined in terms of XML schemas, in order to be extensible and support a generic set of constraints. SLA languages were initially strictly oriented to non-functional QoS related capabilities whose compliance was of importance for systems administrators and developers in order to optimize the application. Later approaches allow to describe also conditions related to the functional aspect of the application that may be of interest in consumer/provider relationships. Most of them are defined along with a framework to monitor the required data needed to check the compliance to the SLA.

**Sahai et al.** One of the first SLA languages that can be automatically processed was proposed by Sahai et al. [SMS+02], as an XML schema that takes into account a date constraint (a time period where the constraint must be valid), and a set of SLOs. An SLO is defined as a set of clauses that include the data item to analyze (*measuredItem*); the moment when the condition is analyzed (*evalWhen*); a range of measures to consider (*evalOn*); a function to apply to the set obtained (*evalFunc*); and an action to take when the evaluation has been done (*evalAction*). Some of these elements can also be empty. Although simple, this approach allows to express quite complex SLAs regarding QoS constraints [SDM01]. The *evalAction* element allows to plug reactor elements to clauses and provide some post-processing.

**SLAng** SLAng [SLE04, Ske07] is a domain-specific language for describing SLAs in terms of QoS characteristics, in loosely-coupled application services provisioning scenarios which communicate through a network service. The objective is to obtain a description of the QoS characteristics that is precise enough to compare and reason about them.

The abstract syntax of SLAng is described using the EMOF meta-model standard, an extension of UML. The constraints are defined formally using the Object Constraint Language (OCL) to provide the semantics. The SLAng meta-model describe broad elements like *PartyDefinition*, *ServiceDefinition*, *PenaltyDefinition* and *AdministrationClause*, where each one of them can be extended to define an SLA language for each concrete scenario. Actually, SLAng can be better defined as an abstract language that describes general elements that take part in the description of QoS characteristics, and it must be extended to specify concrete SLAs, serving as a base to creating SLA description languages.

One consequence of the abstract definition of SLAng is the possibility to reason about the model. The authors present the concept of *monitorability*. Monitorability is defined as the capability of other parties to oversee the compliance to the SLA [SSCE07], so that any decision about SLA violation cannot be contested by the client or by the provider, and eventual penalties can be applied in a trustworthy way. The assessment of monitorability is obtained by placing SLA conditions only on events that are observable by all the involved parties, and a method is provided to determine monitorability of SLAs described with SLAng.

SLAng does not define a monitoring infrastructure to assess the compliance to the SLA, as it intends to be as independent as possible of concrete implementations.Later works have used SLAng [MPMJS05] to automatically build monitoring elements from an SLAng based description, as part of a monitoring middleware for SLAs. This monitoring middleware uses SLAng to

identify the metrics that need to be monitored and enforces the connection to the sources of that metrics to feed an SLAng analysis engine.

**WSLA** WSLA (Web Service Level Agreement) [KL03, LKD+03] is a framework developed by IBM [3] for specifying SLAs between a service consumer and a service provider and the obligations of both parties, expressed in an XML language.

The WSLA language comprises three sections: (1) a description of the parties involved in the contract including their technical properties and their available interfaces; (2) the service description including the SLA parameters observable in the service and a definition of the SLA parameters in terms of metrics; (3) the obligations of each party including action guarantees and constraints over the described SLA parameters, in terms of SLOs expressed as conditions.

In addition to the language, the WSLA framework defines the existence of a *measurement service* that is able to obtain the values for the specified metrics, and a *condition evaluation service* which is able to check automatically the conditions that define an SLO.

WSLA does not include a protocol for SLA negotiation, assuming that this step has been previously made. The obtained SLA description may complement existent WSDL descriptions of services. The WSLA specification and an *SLA compliance monitor* that includes both the *measurement services* and the *condition evaluation service* have been implemented as part of the IBM Web Services Toolkit.

**WSOL** WSOL (Web Services Offering Language) [TPP+03] is an extension to WSDL for specifying constraints in web services. It introduces the notion of *classes of services*, to refer to a concrete service along with its associated QoS conditions, so that one service can have many classes offering different QoS levels.

The WSOL specification defines: *service offerings* as a representation of a single class of service; *constraint expressions*, where functional or non-functional constraints can be specified, relying on external ontologies, and access rights; *management statements* to describe the conditions under which the service is offered; and *reusability constructs* to facilitate reutilisation of previously specified constraints.

WSOL also gives support to specify some simple management actions. WSOL allows to specify what they call *service offerings dynamic relationships*, which can express predefined service offering alternatives in case one particular service offering cannot meet the constraints. Although these relationships must be predefined and they are static, it allows to guide adaptations in the composition of the services at runtime in case an SLA violation is detected.

**WS-Agreement** WS-Agreement [ACD+07] is a standard developed by the Global Grid Forum (Open Grid Forum) for specifying agreements between service providers and service consumers, and a protocol for creating (as a result of a negotiation) and monitoring such agreements at runtime.

The WS-Agreement language is defined as an XML schema containing two parts: *Context* and *Terms*. The *Context* describes the partners: service consumer and service provider that take part in the contract. The *Terms* describe the objective terms of the service provision. The *Terms* section includes the *service description terms* that describe the functionalities that are going to be delivered, and a set of *service guarantee terms* describing assurances on service quality that need to be enforced during the provision of the service.

WS-Agreement also defines a single round negotiation process where the initiator sends an agreement template to the consumer, which fills the template according to certain constraints specified in the template, and sends it back to the initiator as an offer. The initiator then decides the acceptance or rejection.

[3] http://www.research.ibm.com/wsla/

WSAG4J[4] is an open source implementation of WS-Agreement for Java based environments, that has been used for implementing SLA Monitoring policies [CS10], and dynamic SLA negotiation [PWZW09] by developing extensions to specify policies and service management.

CREMONA (Creation and Monitoring of Agreements) [LDK04], (whose monitoring framework was mentioned in Section 3.1.1.5) includes a Java library that provides an architecture for implementing the WS-Agreement negotiation process. The agreement processes is encapsulated in an "Agreement Provider" that drives the negotiation and delivers an accepted agreement to the monitoring framework. CREMONA works as an agreement middleware for binding service providers and consumers and monitoring the compliance to the agreements.

**WS-Policy** The Web Services Policy Framework (WS-Policy) [W3Cd] is a W3C recommendation to express the capabilities, requirements and general characteristics of entities in a service-based system, expressed in an XML schema. According to the specification, a *policy* is a collection of *policy alternatives*, where each policy alternative is described as a set of *policy assertions*. A policy assertion is a requirement on a service like transport protocol, security or QoS characteristics. WS-Policy defines a set of operators to combine policy assertions.

WS-Policy is used to express conditions in the interaction of consumers and providers. By expressing their capabilities using WS-Policy, consumers and providers can be aware of the expectations of each other and decide if they enter into an agreement or not. Once the agreement is reached, policies can be related to specific technologies to enforce the agreed behaviour.

Being a general specification, WS-Policy does not provide specific constructs for each kind of policy (security, reliability, monitoring), but it can be extended as required in a domain specific way. WS-Col (Web Service Constraint Language) [BGP06] is an example of such an extension targeted at specifying monitoring constraints on services executed by a WS-BPEL composition.

**MoDe4SLA** MoDe4SLA [BWRJ08] is an approach for diagnosing the cause of SLA violations in addition to only detect them. MoDe4SLA introduces a dependency model constructed from the SLO specifications (in an SLA language independent way) to determine the set of services on which a specific service depends, and to calculate an impact factor for each one of them. This way, upon an SLA violation, it is possible to drive the responsibility on the called services and determine which of them are having an acceptable behaviour and which are likely candidates for blaming or replacement.

The main contribution of MoDe4SLA is to consider explicitly the composing services on the diagnosis on an SLA violation, and to introduce an impact factor per service, that can help to drive a subsequent planning process. However, the dependency and impact analysis are rather static and cannot evolve if the composition changes at runtime.

#### 3.1.2.2 Prevention of SLA Violations

In order to ensure runtime SLA compliance, tools can not limit to only monitor a service and detect when a violation has occurred. Instead, the approaches to ensure SLA compliance require to execute an action to ensure that SLA compliance be restored. In our work, we have separated the planning and execution of actions to restore SLA compliance from the proper detection of an SLA violation. However, other approaches to deal with ensuring SLA compliance take a more proactive approach and attempt to use prediction of SLA violations.

Prevention of SLA violations attempts to predict the occurrence of SLA violations before they actually happen, allowing to take actions to avoid their occurrence. In order to prevent an SLA violation, some kind of *prediction* must be made. Approaches for prediction can be based on historical monitoring data collected from a service, and may involve the use of statistical estimators and probability fitting algorithms. In general, a dynamic prediction takes into account the elapsed part and the remaining part of the service, and must be able to take a decision in a restricted time lapse.

[4] http://packcs-e0.scai.fraunhofer.de/wsag4j/

**Prediction in workflow compositions** Canfora et al. [CDPEV05, CDPEV08] present a dynamic approach for workflow compositions where the analysis of SLA compliance is performed after the execution of each step of the workflow, as new information about the execution is available. Given an objective QoS level $Q_{TH}$ expressed in the SLA, the method updates two values: the QoS level $Q_{ACT}$ achieved until the current step, and the estimated QoS level $Q_{EST}$ after the execution of the current step. The $Q_{EST}$ is computed for each step during the QoS-aware composition of the workflow according to predefined estimator formulas for each workflow construction: loops, switches and sequences. Loops are estimated by annotating an estimated number of iterations; switches are estimated by assigning probabilities to each branch; and sequences are estimated by simple addition. When the values for $Q_{ACT}$ and $Q_{EST}$ differ in more than a given percentage, a replanning is preventively triggered.

**EVEREST+** EVEREST+ [LS10] is an extension to the EVEREST framework, that can perform prediction of QoS violations. EVEREST [MS07] is a generic monitoring engine for checking violations of software system properties. Conditions in EVEREST are expressed using *EC-Assertion*, a language based on Event Calculus (EC) [Sha99], a first order temporal logic language. EVEREST has been used for monitoring SLOs in service-based systems.

Monitoring rules in EC-Assertion are expressed in the form $body \Rightarrow head$, meaning that when *body* is true, then *head* must also be true. Both sides are expressed in terms of standard EC predicates like *Happens()*, *HoldsAt()*, *Initiates()*, *Terminates()*, and *Initially()*.

The monitoring framework extends WS-Agreement to include SLA description, including conditions expressed using EC-Assertion. The set of events that can be used is restricted to those that can be observed during the execution of the service-based application, which is expressed as a BPEL composition. A monitoring manager connects to the event sources of the service-based application, and stores the events in an event database. A component called *monitor* filters the events that are relevant for the specified rules in EC-Assertion and checks their compliance.

EVEREST+ allows to attach predictors to the EVEREST framework. EVEREST+ predictors can use historical event data stored in the EVEREST database to determine the probability that a given condition will be fulfilled or not, by fitting statistical distribution functions to them. EVEREST provides an API to query the historical event data, so that custom EVEREST+ predictors can be attached to it.

This approach highlights the separation between the event storage and collection, and the prediction function, which makes use of the stored events.

### 3.1.3 Planning

The planning phase in an adaptation loop corresponds to the construction of a plan to take a system to a predefined objective state. In the case of adaptation loops for service-based applications, this step takes place after a condition, expressed within an SLO has been broken, in systems that aim to ensure runtime SLA compliance.

The most simple approaches for determining actions, take the form of tables of rules that include conditions and actions, where the relative truth value of the conditions determine the action to take (fuzzy logic approach) like those presented in AutoPilot [RSR01]; the Event Calculus expressions of EVEREST [MS07], where certain conditions may trigger a set of actions; or the actions that may be embedded in some SLA descriptions [SMS+02]. All these approaches, however are static, as the set of actions that can be taken is predefined at runtime.

Other works [LLJ+05, CDPEV08] involve approaches from the artificial intelligence area, where the service has an initial state and, by means of heuristics, the service must be taken to an objective state. Depending on the criteria that wants to be optimized, a cost function is defined that can involve multiple QoS parameters. In these cases the set of actions is mostly abstract and the planning algorithm must use the available information to create a concrete plan.

The set of actions that can be decided may also range from very simple actions like tunning some parameters of a service (number of connections, resource usage), to execute replacement of services or more complex reconfigurations of a composition. In any case, for effects or our work, the objective of the planning phase is the generation of this set of actions, as we delegate the execution to specialized tools like those mentioned in section 3.1.4.

In the general case, generating a reconfiguration plan for a service composition may involve several steps: perform service discovery to find a set of available services; select a subset of candidate services that can be used as replacement, which involves to check interface matching (or adaptation); collect information about the candidate services to feed a decision algorithm; and finally to select an appropriate set of services to bind.

The problem of selecting an appropriate subset of services from a set of services that implement the same functionality, subject to certain QoS constraints, is known as *QoS-aware composition*, and it is an NP-Hard optimization problem [BF05]. The QoS criteria under which a composition is considered "better" than other may include multiple factors like price, performance, energy consumption, security, reliability, or availability. Section 3.1.3.1 mentions some approaches that have been used to perform an efficient QoS-aware selection and composition of services.

Due to the dynamic nature of service environments, in most cases even an optimal composition may not guarantee a required QoS level at runtime, because the QoS characteristics of the service may change at runtime and differ from the values used for computing the optimal composition. Section 3.1.3.2 presents some works that tackle the dynamic recomposition of services at runtime.

Another topic in the computing of a QoS-based composition is the source of QoS data. It can be obtained from QoS values exposed or published by the candidate service; it can be estimated from a set of historical monitoring data; or it can be monitored dynamically from the client side (as for example in [RPD06]).

#### 3.1.3.1 QoS-aware Service Selection

The problem of QoS-aware selection can be seen from the point of view of substituting a single service for another that provides the same functionality. Ghezzi et al. [GMPLMT10] compared several strategies proposed for performing the selection of a single service based on QoS information, along with an identification of situations where some of them present advantages over the others. The strategies include local decisions, estimators of QoS parameters, proxy-based selection, and a collaborative strategy, and they were analyzed in terms of how well balanced is the assignation from a set of clients who wants to bind a service to a set of providers that implement the same functionality. The strategies do not include the discovery step, and assume that a previous filtering process has been performed to provide a set of concrete services, so the decision step reduces to QoS based decision. In general there is no one-size-fits-all strategy and some of them are better suited to certain conditions than others.

A more complex problem involves building a composition of services in order to achieve an end-to-end QoS goal. The analyzed approaches consider workflow compositions that express abstract services, each of them having a set of candidate concrete services that must be selected. Several heuristics and nearly-optimal algorithms have been proposed for performing this task efficiently [BSR+06] considering BPEL compositions [MCD10]. The approaches may use techniques from genetic algorithms [LLJ+05, CDPEV08], linear and integer programming [ZBHN+04]. A common way to separate the concerns of the workflow composition from the selection of services is to design the composition as a set of *abstract services* with some optionally defined QoS constraints, and bind them to proxies or *brokers* who are in charge of collecting the required information from the candidate services and performing the selection [YL05, DAT08, SDHS05] to bind *concrete services* to them. This approach has the advantage that it separates the decision taking from the design of the workflow, which is viewed as a skeleton where concrete services can be bound and unbound guided by QoS constraints. Yu et al.

[YZL07] evaluate a series of heuristics for sequential service flows, and more general and complex flows by mapping to $0-1$ integer programming, or to a graph model where appropriate algorithms can be applied.

The mentioned works assume as input a set of services that can be effectively bound (or that can be bound by performing a previous adaptation), as a result from a previous discovery step. They also assume that the information about the QoS characteristics of the candidate services is already available, or can be obtained by independent monitoring data, or by some available estimator.

#### 3.1.3.2 QoS-aware Dynamic Rebinding of Services

The works mentioned in Section 3.1.3.1 perform a static selection and composition of services in a workflow, which is carried on before the execution of the workflow. In that situation the time taken to perform the selection and composition may not be a main concern.

However, in dynamic environments the selection and composition process may be triggered by a failed QoS condition that was detected at runtime (a service that is not available anymore, or a violated SLO), and a decision or reconfiguration plan may need to be taken in a very short time. The following approaches execute dynamic rebinding of services, and provide algorithms that attempt to deliver a near-optimal response during the execution of a workflow composition.

Canfora et al. [CDPEV05, CDPEV08] describe a dynamic runtime approach where replanning can be triggered preventively during the execution of a workflow. In this solution the binding between the abstract and concrete services is realized by means of a proxy service which contains the partner link to each concrete service. The proxy service is in charge of retrieving the set of candidate services and choosing the one that best contributes to fulfill the QoS requirements of the composition. This approach gives more control to the workflow engine.

Mabrouk et al. [MBK+09] presents a solution for efficient QoS-aware composition for dynamic service environments, in the context of the SemEUsE project [5]. As in previous works, the approach uses a guided heuristic based on K-means clustering algorithms to find a set of near-optimal service compositions that respect the imposed QoS constraints, and maximize a defined QoS utility function. The approach considers two steps: a local phase performed for each activity in the composition where clusters of candidate services are selected, and a global phase where candidates for each activities are composed together to obtain the near-optimal composition.

Alrifai et al. [AR09] proposes a solution that combines global and local optimization techniques. The approach decomposes the QoS global constraints into a set of local constraints that are used as conservative upper and lower bounds, and this separated problems are solved by local service brokers, providing more scalability. However, the greedy approach taken at the local level may not guarantee that the global QoS constraint be respected. Later approaches [ASR10] perform a previous filtering of services based on their level of accomplishment to some of the multiple QoS criteria to have a more appropriate filtering.

### 3.1.4 Execution

The execution phase is generally highly attached to the planning phase and most of the times is completely part of it. In our work we envision that the planning phase is a separate process from the actual execution of the plan, so to allow the generations of plans that are independent of the technology used for hosting the services. In this section we mention the existing support for reconfiguration in applications based on services and/or components, and existing adaptation frameworks.

#### 3.1.4.1 Reconfiguration Support

Reconfiguration of software architectures is a topic that emerged as software architectures became more complex and the need for supporting dynamically evolving architectures came out.

[5] http://www.semeuse.org

In fact, the increasing demand for continuously available software services, where downtimes are highly risky and inconvenient, requires to have architectures that can be safely modified at runtime without disturbing the QoS of the service.

Most of the proposed architectural modeling notations use ADLs that focus in static architectures. Oreizy [Ore96] highlighted the need of having an operational Architecture Modification Language (AML) and a declarative Architecture Constraint Language (ACL) that should cooperate to be able to describe and support architectural reconfigurations, and provide a formal framework where reconfigurations can be analyzed.

Adaptation requires support for modificating the architecture. Oreizy [OGT+99] identifies the convenience of relying on an architectural model of the system to drive reconfigurations, and software connectors to aid in runtime change. A recent survey [OMT08] highlights features that have been proved convenient in providing dynamic adaptability like clear identification of adaptation points, introduction of adaptation features into the targeted elements, and late binding. Most of these recommendations coincide with features provided by component models.

Another important concern targets the safety of reconfiguration actions. When executing modifications on a composition, these change can potentially render the composition of the application in an inconsistent state, f.e., if a consumer has sent a request to a service and this service is dynamically removed from the composition, the consumer may not receive an answer and block; or in a more general case, one action of the list may fail by external reasons or violate some restriction of the environment, in which case the reconfiguration may be incomplete.

**FScript/FPath** FScript [DLLC09] is a scripting language designed to support navigation and reliable reconfiguration of architectures based on the Fractal component model. FScript embeds the FPath language, which provides a notation to navigate inside and query Fractal-based architectures.

FPath is a domain-specific language which models a Fractal-based architecture as a directed labeled graph, where components, interfaces and attributes are modeled as nodes of the graph; their edges define relationships; and the label of the edges define the type of the relationship, like client/server bindings, child/parent relationships, and interface and attribute ownership. The syntax of an FPath expression is inspired in that of the XPath [6] language. Using FPath expressions it is possible to navigate inside a Fractal-based architecture in a precise way and query their components.

The FScript language uses FPath expressions to navigate through a Fractal-based application, select their components, and manipulate them at the architectural level. FScript also focuses in providing reliable reconfigurations, which is guaranteed by providing transactional semantics to the reconfigurations, so that they comply with the standard ACID properties and do not result in an inconsistent state of the system. FScript is also designed in an extensible way, so that it may be extended to support any extension to the Fractal model, a fact that we have used in Section 7.5.1 for the development of the PAGCMScript language.

#### 3.1.4.2 Adaptive architectures

The execution of an adaptation on an application requires support from the running environment to perform the changes. In a distributed environment like the dynamic, service-based applications we target, this task brings challenges. Actions must be executed in the appropriate services, and must be described in a precise way.

**SAFRAN** SAFRAN (Self-Adaptive Fractal Components) [DL06] is an extension to the Fractal component model to provide self-adaptation to components via an aspect-oriented approach. SAFRAN components include a controller called *adaptation-controller* which defines an interface that allows to attach or detach *adaptation policies* to the component. A component with such

[6] http://www.w3.org/TR/xpath/

controllers turns into an *adaptable component*. An *adaptation policy* in the context of SAFRAN is a set of ECA rules. This, rules described in the form: "when [event] if [condition] do [action]".

In SAFRAN, the aspect-orientation is realized as follows: a base program corresponds to the Fractal component; *point-cuts* correspond to notifications of events, because this is the moment where an evaluation is performed (and the control is taken off the main program); *advices* correspond to architectural reconfigurations expressed as *actions*; and the *aspect* implementation corresponds to the *adaptation policies* that maybe dynamically attached or detached via the *adaptation-controller*. SAFRAN provides a dynamic aspect-oriented approach where aspects can be dynamically weaved or unweaved from the base program.

SAFRAN may detect *internal events* related to the execution points in the base program, obtained by instrumenting the Fractal controllers, like the start or stop of a component, or the reception of a message; or *external events* related to the application context, and which are detected by using the WildCAT system [DL05]. Finally, *conditions* in the adaptation policies are represented as FPath expressions, and *actions* are describe using FScript [DLLC09].

SAFRAN uses a model based on a Fractal extension to provide adaptation, which is similar to our approach. By using a dynamic aspect-oriented approach, SAFRAN separates the component functional execution from the adaptive behaviour which can be dynamically added or removed from the component. Our approach also allows to dynamically attach or remove adaptive actions, but we rely on a component approach to incorporate other aspects from the control loop, for example separating the analysis of conditions and the set of events to detect, that we believe provide an even more clear separation of concerns, and allows more complex implementations.

**Frascati** FraSCAti [SMF+09] is a platform for supporting SCA-based applications. FraSCAti is an implementation of the SCA specification and is based on the Fractal component model. FraSCAti extends the SCA specification to allow dynamic introspection and reconfiguration of SCA applications. FraSCAti exploits the SCA Policy Framework specification to associate non-functional services to components. Non-functional services are implemented as regular SCA components and are attached to FraSCAti component via interception of requests.

FraSCAti has been used as support in frameworks that implement closed adaptation loops [RRS+10]. Being based in Fractal components, reconfiguration capabilities are naturally expressed using FScript/FPath, though other languages could be used thanks to the component-based approach.

**iPOJO** iPOJO [EH07, EHL07] is a service-oriented component runtime, that runs POJO applications on top of OSGi. The approach of iPOJO consists in wrapping POJO, non service-oriented applications that implements a specific business logic, with *handlers* that provide the service-related functionality like discovery, binding, lifecycle, and provisioning.

iPOJO uses offline byte-code injection to add the required handlers to a regular POJO and transform it into an OSGi bundle that can be dynamically deployed. iPOJO can handle dynamic adaptation and reconfigurations of the composition and the lifecycle of the application, by managing the fields values and bindings, due to the injected bytecode. However the composition of the handlers cannot be modified at runtime, unless a copy of the service with the modified handlers injected be provided and replaced.

**GrADS** The GrADS (Grid Application Development Software) framework [BCC+01] aims to develop adaptive applications for Grid environments. Adaptivity is implemented inside the middleware. A common application built using the GrADS library is compiled and converted into a *configurable object program*, which can be managed by the GrADS middleware by mapping them to appropriate Grid resources, and dynamically adapt them at runtime according to previously defined performance contracts.

Being specifically developed for grid environments, actions mainly include rescheduling and migration of applications. In this case, the adaptation control loop is tightly coupled to the runtime environments, with the intention to separate it from the functional code design. However,

both the adaptation control loop, or the criteria to trigger adaptations cannot be modified at runtime.

## 3.2 Frameworks and Tools

There is an extensive set of frameworks and tools that have been proposed, from the industrial and from the academic world, which address partially the phases involved in the control loop that we aim to integrate. This section presents a subset of them that we have identified as the most representative, along with their main features.

### 3.2.1 Frameworks for Monitoring and SLA analysis

As mentioned through sections 3.1.1 and 3.1.2, tools for SLA Analysis in service-based applications require a monitoring infrastructure to provide the data needed to perform the analysis. In some cases, this monitoring infrastructure is assumed to exist at the moment of defining the SLA, but in several occasions the monitoring infrastructure is defined along with the SLA, as they are a natural complement.

The following works focus on integrated approaches for both Monitoring and SLA Analysis in component or service-based applications.

#### 3.2.1.1 WildCat

WildCat[7] [DL05] is a generic monitoring framework for building context-aware applications. WildCat uses event detection to capture information from the environment, and provides a toolkit to define custom sensors. The events collected are processed through the Esper Complex Event Processing engine [Esp]. WidlCat uses EQL (Esper Query Language) queries to detect complex conditions from several distributed sensors arranged in a hierarchical organization.

WildCat defines a *context* as a tree structure, where *attributes* are the leaves of the tree and contain values. Values can be static values, active probes (associated to operating system values, or custom sensors), or query attributes computed from complex EQL expressions. Intermediate nodes are called *resources* and contains other resources and attributes. An example WildCat hierarchy is shown in Figure 3.4.

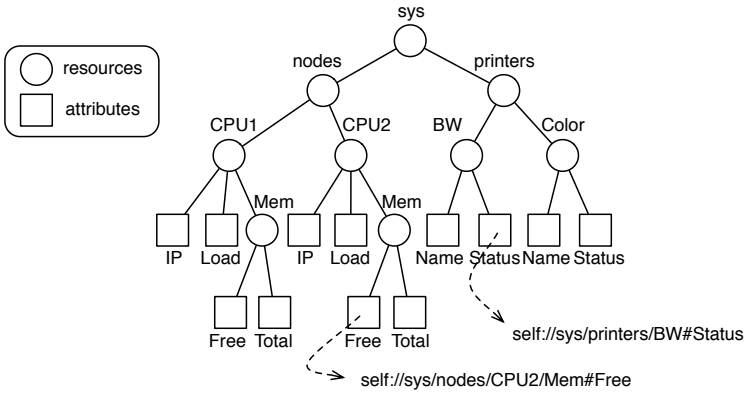


Figure 3.4: Example WildCat hierarchy, along with examples for addressing attributes.

WildCat has been used for implementing context-aware applications due to its generic approach for defining custom sensors [MLBH+09]. The hierarchical organization allows to aggregate and propagate monitored information in a scalable way. The CEP engine allows to detect

[7] http://wildcat.ow2.org/

complex conditions that, in our case, can be mapped to SLOs, and can be used as an implementation in our framework for the monitoring and analysis phases. Both the monitoring and the analysis in WildCat are, however, tightly integrated which may pose difficulties if only one of them needs to be modified.

#### 3.2.1.2 Kieker

Kieker [vHRH+09] is an object-based framework for monitoring the runtime behaviour of software services. Kieker provides a set of basic tools to add custom monitoring artifacts which, by implementing a given interface, can take the role of monitoring or analysis. Kieker is able to measure the performance of the monitored services and obtain distributed trace data. Kieker uses an interception mechanism based on aspect-oriented programming, where the code of the custom monitoring artifacts is woven into the application code.

The flexibility of Kieker is given by the fact that custom monitoring probes can be programmed and attached in an independent way from the application code. The injection of probes can be done in two ways: by explicitly annotating the methods of the classes that needs to be monitored (f.e. using AspectJ); or by intercepting services requests using Spring or Apache CXF SOAP interceptors to have a less intrusive approach.

The output of the monitoring is generated as text files which describes the monitoring records obtained. These files can be used as output for the analyzers objects. The analyzer objects can also be customized according to the kind of analysis required and plugins are provided to generate graphic results in the form of sequence diagrams, dependency graphs and call trees, using Graphviz and GNU PlotUtils.

Though Kieker allows to create custom monitoring probes for the monitored services, it does not allow to add or remove the monitoring probes at runtime, as the scheme can only be used at compile time or load time, and once the probes are woven into the application code, they cannot be removed.

#### 3.2.1.3 VRESCo

The Vienna Runtime Environment for Service-Oriented Computation (VRESCo) [MRLD10] is an SOA runtime environment which provides several features often ignored in SOA environments, like explicit distributed registries for service discovery [MRP+07]. VRESCo provides a service metadata model to represent registry information which allows to guide dynamic binding and service mediation. The core services provided by VRESCo are accessible through SOAP interfaces. VRESCo uses a framework called DAIOS [LRD09] to provide dynamic and asynchronous invocation of services.

VRESCo integrates a *QoS Monitor* [RPD06] element, which monitors services using aspect-orientation techniques and low-level TCP analysis. The *QoS Monitor* uses a client-side approach, which is very useful in cases that there is no access to implementation details on the server-side. The *QoS Monitor* parses the WSDL description of the services and introduces monitoring code in the automatically generated stubs, so to introduce cutpoints before and after the invocation methods and obtain the response time from the client side. At the same time, the *QoS Monitor* uses a TCP packet sniffer to capture the TCP traffic generated by the invocation and, by detecting handshakes and response transfers, it intends to associate TCP events to method calls, and measure the latency and execution time on the server side. The QoS model used allows to measure performance metrics like latency, response time, availability, accuracy and throughput. Other metrics may be introduced in the model, however this is possible only at design time.

SLA violation detection in VRESCo [MRLD09] uses the information obtained from the *QoS Monitor* and an eventing architecture [MRLD08] that relies on the Esper CEP engine [Esp]. SLA obligations are defined using objects that include attributes like: property to monitor, period of validity of the obligation, operation to which the obligation is applied to, comparison operator, threshold, and notification mechanism. More complex obligations with additional attributes can also be used. The SLA obligations are attached to the service via the Publish Interface of

VRESCo, and transformed to EPL (Esper Processing Language) subscriptions. The subscriptions are processed by the event engine, which establishes a match between subscriptions and events. After this step, the event engine notifies the subscribes whenever such a match occurs using, for example, E-Mail or WS-Eventing.

The PREvent framework is an extension to VRESCo [LWR+10, LMRD10] aimed to detect an SLA violation before it happens, and modify the service composition in order to avoid the SLA violation. The approach defines *checkpoints* as points in the execution where a prediction can be performed. Checkpoints are manually described using an XML-based language that includes the class that implements the predictor logic, the location of the checkpoint with respect to the activities of the service, and a list of facts (real metrics) and estimators (estimated metrics) to consider. Prediction is based on monitored QoS data and historical metrics. These data are used to train a *prediction model*, which can be a regression classifier (f.e. the WEKA [8] machine learning toolkit). The accuracy of the prediction is estimated from the historical data. The *predictor* compares the predicted SLO value with a given *adaptation threshold* and trigger a *Composition Adaptor* in case it determines that a violation is likely to occur. The *Composition Adaptor* chooses, from a set of predefined adaptation actions described in an XML-based language, the subset of actions that must be applied according to a strategy. Two strategies are available: a *safe strategy* which finds the composition that is most likely to prevent a violation (the predicted value is as far as possible from the threshold), or a *minimal strategy* which finds the composition that provides a predicted value as close as possible to the threshold and avoiding the violation.

VRESCo is one of the few environments that explicitly include preventive detection of SLA violations. Monitoring relies generally on interception of requests, as the environment has control over all the interactions. Adaptation is limited to predefined strategies. Although it is possible to include additional strategies it does not seem to take advantage or interact with current monitored information. Flexibility is restricted to design-time decisions.

#### 3.2.1.4 PADRES

PADRES [JCL+10] is a middleware for distributed publish/subscribe event management, implemented in Java. PADRES provides content-based routing by implementing an overlay of publish/subscribe brokers, where consumers declare their interests as subscriptions, and providers publish their content. PADRES has been applied in the development of an SOA runtime environment by serving as a distributed ESB that provides routing and mediation between loosely coupled consumers and providers.

The eQoSystem project aims to simplify the development and management of business processes automating tasks like monitoring, deployment and resource provisioning. In this context, PADRES has been integrated with SLA concerns through the development cycle, from the business modeling steps into a machine-understandable form. SLA rules are expressed using an SLA language that extends WSLA, where complex SLOs can be defined by composing pre-existing simpler SLOs.

From the business model and the defined SLA rules, the individual SLOs are associated to metrics which can also be compositions of simpler metrics. Individual metrics are associated to events that must be generated at runtime. The monitoring is realized by subscribing to the appropriate events through PADRES. This way, subscriptions are only enacted for the metrics associated to the required SLOs. Actions can be associated to metrics as Event Handlers, and Action Handlers can be associated to SLOs to take customized actions upon a violation event. The SLA is finally seen as a set of SLOs and Action Handlers [CMJ+08].

From the description of the SLOs and metrics, a monitoring client is automatically generated for each metric, SLO and action handler, which registers to the appropriate events through PADRES. The conditions under which the SLO rules can be applied are indicated using objects called *scope*s.

[8] http://www.cs.waikato.ac.nz/ml/weka/

The eQoSystem/PADRES approach is highly attached to a business process description and relies on a business process engine like BPEL to drive the execution of the business process and associate events, metrics and actions to the execution. eQoSystem allows to define new metrics and rules that can be applied at deployment time, and adaptation actions can be attached to them to provide runtime adaptability [MJC+09]. All the adaptation features, however, must be specified at deployment time and cannot be modified at runtime. The adaptation actions, also, cannot interact with the monitored information at the level of individual services, which impends to take more pertinent actions. Further work aims to apply this approach to manage business processes in Cloud environments [MJ10] by using a distributed set of BPEL engines, and mapping the BPEL process to events in PADRES in a distributed orchestration architecture called NIÑOS [LMJ10].

#### 3.2.1.5 SERVME

The SLA-based SERViceable Metacomputing Environment (SERVME) [RS09] is an environment for federated metacomputing that aims at matching providers based on QoS requirements, and performing autonomic provisioning and deprovisioning of services according to dynamic needs.

SERVME presents an object-based SLA model designed to meet the requirements of metacomputing environments. SERVME defines a *QosContext* object that represents the QoS requirements of the requester and includes functional requirements, system requirements (for resource matching purposes), organizational requirements, metrics, and SLA parameters representing the ranges of accepted values expected from the provider. During the negotiation, an object called *SlaContext* represents the offer from the provider and includes the SLA parameters offered, the *QosContext* that the provider intends to satisfy with this offer, the access point to the service that guarantees the SLA compliance, and a state of the negotiation. This model allows to drive an SLA negotiation process and perform matching of requirements.

At runtime, the compliance to the SLA is monitored by an *SLA Monitor* that is built based on the negotiated SLA, and a *QoS Monitor* able to monitor the QoS parameters of the provider. SERVME also includes an On-demand Provisioner, able to obtain service providers via the Rio Provisioner [9].

The focus of SERVME is to provide runtime compliance to the defined QoS requirements on the provided services. Adaptation is provided through the autonomic provisioning and deprovisioning of services. Monitored data about the services is assumed to be available at runtime.

#### 3.2.1.6 SLAM4M

The SLA@SOI project aims to provide an SLA-aware infrastructure to service oriented applications. One of the important topics that have been identified is the support for dynamic evolving SLAs and the consequences they have on the SLA monitoring infrastructure. The authors define the issue of *SLA Monitorability* as the capability to monitor a required set of SLA terms. The authors separate the SLA monitoring task in two interacting layers: the *SLA Management* layer that includes the mechanisms for performing *SLA Monitorability* checks and to dynamically setup a monitoring infrastructure to enable the SLA monitoring; and the *Service Management* layer that includes the event captors and monitors that perform the SLA checks.

The authors present the SLAM4M (SLA Management for Monitoring) framework [CS10, CS09b], which belongs to the *SLA Management* layer and performs a coordination process of available events and SLA monitors. The architecture is event-based, and considers a set of event captors associated to the composed services, and a set of monitors which are capable of receiving events and SLA rules, and performs SLA checks using that input, and detecting violations.

The SLA rules are introduced to the framework using the WS-Agreement language that has been extended to specify SLA Guarantee Terms. The event captors and monitors expose their

[9] http://www.rio-project.org/

monitoring capabilities, i.e., the set of events they produce and their SLA checking capability, using an XML-based language defined for this purpose. SLAM4M uses this information to check if the set of required SLA terms is monitorable using the available capabilities (event captors and monitors) and associates them as required, delegating the SLA checks to the available SLA monitors.

The authors assume the existence of a predefined interface that is exposed by event captors and SLA monitors in order to communicate their monitoring capabilities, to connect event producers with event-receiving SLA monitors through an event bus, and submitting SLA terms to the SLA monitors. Using this interface it is possible to perform the reasoning and associate the appropriate event sources and receivers.

The approach can be applied in a hierarchical way [CS09a, CS09b], in order to analyze SLAs at different levels of the composition of a service based system. This way, only the events that are useful for the checking of each SLA term are stored in a database.

The main advantages of this work are the separation of tasks between the SLA checking and the available monitoring sources. We extend the idea of separating these concerns, including in the layer related to the monitoring tasks the possibility of having specialized event captors for each different monitoring interface available, possibly including the interfaces defined by SLAM4M.

#### 3.2.1.7 DIPAS

DIPAS [TBNF09] is a distributed performance analysis service for web service-based workflows in grids, developed for the K-WfGrid system. Interfaces are based on WSRF to allow other services and clients to use the perfomance analysis features and facilitate integration. Communicated monitoring data is transmitted using XML-based representations. The performance evaluation is carried out by a component called *DIPAS Gateway* that analyzes the monitoring data, workflow representation and analysis requests from clients. The DIPAS Gateways publish their information into a registry, so that clients can locate them.

The DIPAS Gateways obtain the monitored information by connecting to grid monitoring services like GEMINI (Grid Monitoring and Instrumentation Service) from K-WfGrid. Although they can connect to multiple GEMINI services, the analysis is performed in a single DIPAS Gateway. The available information computed through DIPAS includes execution tracing, performance overhead analysis, and detection of specified performance problems. DIPAS defines a Workflow Analysis Request Language (WARL), which is used to specify analysis requests, and allows to define constraints (WARLConstraint), performance metrics to be analyzed (WARLAnalyze) and performance conditions (WARLPerfProblemSpecs). These elements are sent to the DIPAS Gateway and can be used to specify performance conditions to be checked during the workflow execution. There is no action associated to these conditions, as the objective is to specify what the client wants to be displayed.

DIPAS is an example of a broker based architecture for monitoring and analysis, where the DIPAS Gateways interact with the monitoring tools and make the monitored information available through a registry. DIPAS is also very interesting for the distributed analysis, in which the tracking of interactions between services is performed through all the DIPAS Gateways involved.

The final purpose of DIPAS is oriented to analyse workflow interactions in service grid environments, so the available information and analysis tools are specific for these infrastructures. It is not possible to add new monitoring concerns at runtime. However it is possible to specify runtime conditions to be analyzed through the external DIPAS Client.

#### 3.2.1.8 GT4

The most recent version of the Globus Toolkit, version 4 (GT4), [Fos06] is an evolution of the Globus Toolkit oriented to the development of services and applications over high performance environments. The architecture of GT4 promotes interoperability by defining interfaces based

on Web Services standards, a major change respect to previous versions. In GT4, most functionalities are provided as services. A set of *infrastructure services* allow to manage computational resources, addressing execution management, data access and movement, replica management, credential management, monitoring and discovery.

GT4 includes an *aggregator framework*, a software framework that can be used to build services that collect and aggregate data from other services. The monitoring solution of GT4, called Monitoring and Discovery System (MDS4) [SPM+06, SRP+06] is a realization of the aggregator framework, but other collecting and aggregating service can be built.

MDS4 implements the Web Services Resource Framework (WSRF) [OASd] and Web Service Notification (WS-Notifcation) specifications to provide query and subscription interfaces to detailed resource data. WSRF is a collection of specifications that aims to model and access stateful resources using Web Services (which are, by definition, stateless). WS-Notification is a specification to push information to other web services, enabling event-driven programming.

In MDS4, the main element is an *Index Service* that collects information about resources from several *information providers*, or from other Index Services. Information providers are already defined for common interfaces from existing monitoring tools like Nagios, Ganglia and Hawkeye, among others, and custom ones may be defined. An *Index Service*, built into each GT4 container, collects data using WSRF Resource Properties and WS-Notification subscription/notification interfaces. *Index Service*s can be arranged forming hierarchies (or other topologies), and become aware of each other by maintaining soft-state registration between themselves. This way, the set of Index Services form an effective distributed registry of monitoring information, that is shared as WSRF resource properties. The collected information can be queried and accessed through separate browser-based interfaces, command line tools, or Web Service interfaces provided by GT4, like WebMDS. An example of a hierarchical deployment of MDS4 Index Services is shown in Figure 3.5.

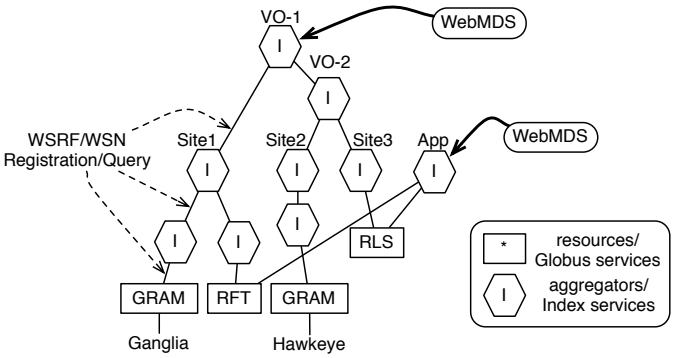


Figure 3.5: A hierarchy of Index Services in MDS4. Globus services act as information providers for monitoring information; among them, the GRAM service is able to obtain monitoring information from external sources. By contacting the Index Services, monitoring clients like WebMDS can obtain information about a set of resources

Overall, MDS4 provides a flexible monitoring framework that can use complex configurations and interoperate with existing tools. The information that can be obtained is mostly related to grid resources, instead of properties of the services. However, the flexibility provided by the WSRF Resource Properties and the possibility to add custom information providers does not forbid to propagate service related information given an appropriate provider and use MDS4 as efficient provider of monitoring data.

#### 3.2.1.9 RESERVOIR

RESERVOIR (Resources and Services Virtualization without Barriers) [RBL+09] is a EU-FP7 project that aims to provide an open architecture for federated cloud computing. The RESER-

VOIR framework focus on IaaS clouds and follows a service-oriented approach by addressing the needs of *Service Providers* that want to offer *Service* applications, and *Infrastructure Providers* who makes computational resources available in the form of a Cloud computing infrastructure.

The RESERVOIR architecture [CEGM+10] builds upon available virtualisation products and *hypervisors*, which are managers provided by concrete virtualisation infrastructures (f.e. Xen, VMWare), and that can execute actions on the infrastructure like replication and migration of virtual machines. RESERVOIR considers three layers to manage virtualised environments. The *Virtual Execution Environment Host* (VEEH), at the lowest level, uses plugins to communicate with *hypervisors* of concrete virtualised infrastructures. The *Virtual Execution Environment Manager* (VEEM) controls the activation/deactivation, migration and replication of the virtualised resources by communicating in an uniform way with multiple VEEHs, and can interact with the VEEM from other cloud providers. The *Service Manager* is the highest layer and communicates with the *Service Provider* to ensure that the requirements are enforced in the infrastructure. A summary description of RESERVOIR architecture is shown in Figure 3.6.

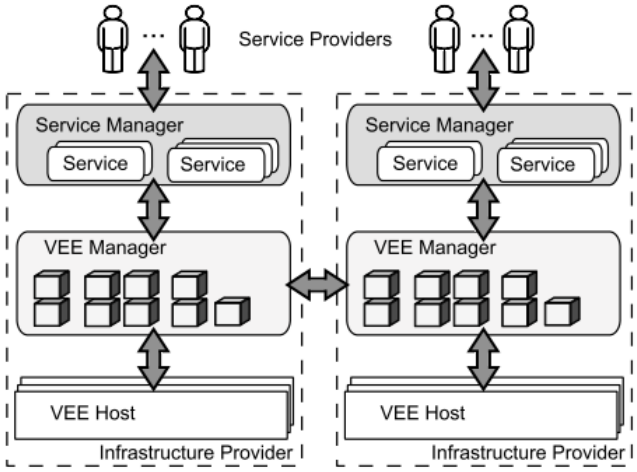


Figure 3.6: RESERVOIR Architecture [CEGM+10]. VEEMs and VEEHs can manage the virtualised resources according to the rules in the *manifest language*.

The requirements that the service provider imposes on the infrastructure are specified using a *manifest language* which is a form of an SLA specification, and is inspired by the model of SLAng [SLE04] mentioned in 3.1.2.1. The manifest language allows to specify the KPI (Key Performance Indicators) from the service provider; the association of the requirements to monitoring parameters; and a set of elasticity rules that allows to define adaptation actions based on an Event-Condition-Action description.

The RESERVOIR architecture shows an integrated approach for monitoring and SLA description, while keeping modular separated concerns. The main advantage is the separation between the infrastructure related deployment constraints, from the service related constraints. This separation allows to adapt one or the other according to the environmental needs. However, by relating QoS requirements to infrastructure concerns, the resource provisioning can be better guided and allows the providers to optimize resource usage while keeping a certain QoS at the level of the delivered service.

### 3.2.2 Frameworks that provide generic autonomic loops

This section present works that attempt to implement autonomic behaviour through autonomic control loops. Initial approaches comprised a restricted set of rules and the different phases of the autonomic control loop were usually tightly coupled. Along with the need for more flexible

architectures, these constraints have been softened. The tools presented here attempt to provide autonomic loops for general purpose applications.

#### 3.2.2.1 Autopilot

The Autopilot [RVSR98, RSR01] framework attempts to allow a distributed application to dynamically adapt to an evolving environment. Autopilot defines a set of elements that take a specific role in the adaptation process: sensors, actuators, clients, and distributed name servers. In the Autopilot model, an application is instrumented with sensors that read values from the application and that compute other values from them; and with actuators that are able to modify the value of a parameter. Both sensors and actuators register themselves with an Autopilot manager. A set of Autopilot managers work as distributed name servers. Autopilot clients are able to connect to these distributed name servers and find sensors and actuators. Decision making is performed through predefined decision tables: when a predefined condition is met from the values read by the set of sensors, a set of parameter modifications is executed through the actuators.

Autopilot puts a great deal of emphasis on the separation of tasks of each entity, and in the heterogeneity of the targeted application. Autopilot does not consider the addition or removal of sensors and actuators at runtime, and it allows only simple predefined decisions and basic actions as the modification of parameters. Nevertheless, this approach is one of the first to consider a clear separation of tasks for implementing a complete adaptive control loop with an implementation-independent approach.

#### 3.2.2.2 Rainbow

The Rainbow framework [GCH+04] attempts to provide a complete control loop for self-adaptation by using an architecture-based approach. Rainbow uses an abstract architectural model that represents the architecture of the application as a graph where nodes are called *components* and represent elements of the system (clients, servers, storage elements, interfaces, etc.), and the arcs are called *connectors* that represent interactions between the components. Components may be hierarchical, and may be annotated with required properties like QoS.

The model is maintained by a *model manager* that connects to *gauges* that collect monitoring information to update the model. A *constraint evaluator* element checks the model periodically and triggers adaptations if a constraint violation is detected. An *adaptation engine* determines the appropriate actions and execute the adaptation through an *adaptation executor* element. A *translation infrastructure* layer maps the *gauges* used by the model manager, and the actions required by the adaptation executor are mapped to concrete *probes* and *effectors* at the target system level. The elements of the Rainbow framework completing the control loop are shown in figure 3.7.

Rainbow shares some characteristics with the framework we propose in this thesis. The elements of Rainbow are directly mappable to the steps of the MAPE autonomic control loop we also target. Namely, *model manager*, *constraint evaluator*, *adaptation engine* and *adaption executor* correspond to the Monitor, Analyze, Plan and Execute steps respectively. By defining the elements in a language independent way, Rainbow allows to provide self-adaptation to different kinds of systems. By relying on the *translation layer* to connect to the concrete API or system elements that are required, the implementation of the autonomic control loop is kept separated from the target system.

However, Rainbow is oriented to provide self-adaption via an architectural model built and updated from the collection of monitored data, integrating all the phase of the autonomic control loop. In our vision, we design the autonomic loop following a component-oriented approach where any monitoring and adapting strategy may be implemented. Moreover, we attempt to attach the autonomic control loop to a set of loosely coupled services which may be available from different providers, whereas Rainbow uses a centralized autonomic control loop that controls all the architectural elements of the target system, which facilitate some design decisions. As a

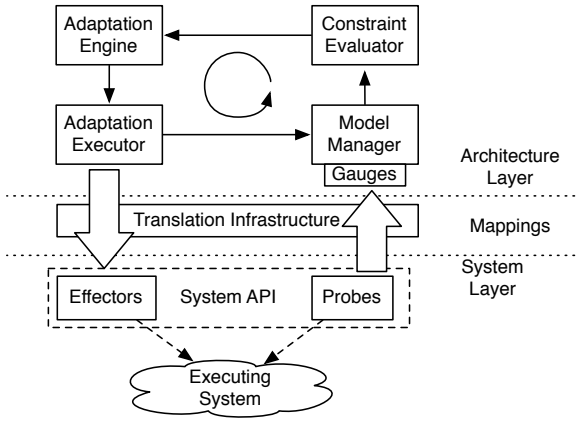


Figure 3.7: The Rainbow framework [GCH+04] uses a model to monitor, analyze and plan adaptations. The translation infrastructure keeps the separation between the model and the concrete probes and effectors.

final differentiation, Rainbow does not support at runtime evolution of constraints, monitoring requirements, or adaptation strategies.

#### 3.2.2.3 StarMX

StarMX [AST09] is an open source generic configurable framework for building self-managed Java-based applications using JMX. StarMX allows to create closed autonomic loops using various mechanisms like action policies.

The framework defines a set of entities called *processes*, which implement the adaptive behaviour and can include Java code or use a policy language that defines behaviours in the style condition/action. Processes can be bound in *execution chains*, which are associated to *activation mechanisms*. Upon calling an activation mechanism, all the processes in the associated execution chain are executed sequentially. Processes talk to *anchor objects* to perform their actions. Anchor objects are sensor, effectors, or other helper objects, and mostly MBeans are used in the implementation.

The communication with the framework is performed through a set of *services* that allow to query the anchor objects, initiate the activation mechanisms using timers or events, define the scope of the actions, data sharing among processes, and obtain logging information, among others. At startup, the framework uses a configuration file to deploy the processes, create the execution chains, and define the activation mechanisms on them.

Though it is oriented to Java-based applications, this autonomic approach attempts to provide autonomicity by following the MAPE control loop and separating the tasks of activation (in activation mechanisms), composable execution of actions (through processes and execution chains), and collection of data and execution (through MBeans that can work as sensors and effectors). The management actions are defined at startup time through a configuration file that cannot be modified at runtime, though it can be extended to interact with other autonomic environments or policies through configuration of MBeans.

#### 3.2.2.4 Entropy

Entropy [HLM+09] is a resource manager for homogeneous clusters. Entropy implements an explicit control loop to analyze the current allocation of Virtual Machines to nodes, and performs dynamic consolidation to reduce the number of used nodes, and reconfiguration plans using constraint programming.

Entropy implements a reconfiguration loop that is deployed in a node dedicated to the cluster resource management. In each one of the managed nodes there is a sensor with privileges (f.e.

running in Xen's Domain-0), and the resource manager node contains the Entropy Reconfiguration Engine. The sensors collect information about the usage of their corresponding VM and informs the Reconfiguration Engine of the state of the VM changes. The Reconfiguration Engine collects the state of the VMs and tries to compute a reconfiguration plan to consolidate the active VMs into an optimal configuration, and devises a plan to achieve this configuration minimizing the required migrations. When the plan is obtained the Reconfiguration Engine sends the corresponding orders to the VM Sensor which can also act as actuators and trigger migration requests.

The interest of describing Entropy is that it shows an application of an autonomic control loop to resource management. Although it is described for a specific kind of homogeneous clusters (provided through Xen VMs), Entropy presents an architecture for the control loop that can be implemented with our approach. In this situation, the sensors acts as collectors and actuators for executing reconfigurations, while the centralized reconfiguration engine involves both the analysis (when to act) and the planning step. A component-based approach, as the one we propose, applied to the Entropy Reconfiguration Engine would allow to experiment with different consolidation strategies that could be, f.e., performance oriented or cost-saving oriented. Finally, the strategies used by Entropy are not dynamically modifiable at runtime, though it is not a main concern in this work.

#### 3.2.2.5 Dynaco

Dynaco [BAP05, BAP06] is a framework oriented to design and develop adaptive component-based applications. Dynaco is described as an assembly of Fractal components that can be specialized for each application according to specific sub-functionalities: decide, plan and execute. The framework provides generic interfaces and defines general guidelines for implementing each functionality. Also, individual components can be dynamically modified fostering self-adaptation of the adaptation framework itself.

The structure of a Dynaco adaptable component is shown in Figure 3.8. The components that implement the adaptation activities reside on the *membrane* of the Fractal component, and are kept separated from the functional content. An special controller, called *modification controller* can be used to access and modify directly to the functional code.

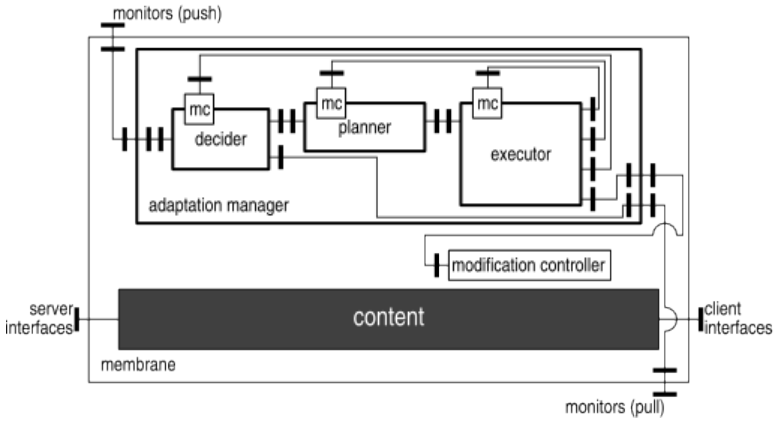


Figure 3.8: Architecture of a Dynaco adaptable component [BAP06].

In our work, we provide a similar decomposition of an autonomic control loop that can be attached and detached to components at runtime, also including the monitoring components as an integral part of the control loop. In addition, we exploit the composability of each autonomic control loop step allowing to have multiple sensors, rules, or adaptation strategies, and composed metrics and actions. In that sense, our work extends the approach taken in Dynaco further

promoting composability, and supporting a service-oriented approach both for the application functional code, and as support for implementing the autonomic management code.

Using Dynaco, other frameworks have been devised. Notably, QU4DS (Quality Assurance for Distributed Services) is a generic framework that covers the SLA life-cycle, including support for SLA description, negotiation, mapping to infrastructure resources, and assurance of SLA conditions.

### 3.2.3 Frameworks that provide autonomic loops for services

This section presents works that attempt to implement autonomic behaviour in service-based applications, or that follow a service-oriented approach for composition. The main contribution of these tools is in the separation of the managment layer, this is, the one that provides the autonomic behaviour, from the functional work of the targetted application, and in the introduction of SLA concerns in the definition of goals for guiding autonomicity.

#### 3.2.3.1 Cappucino

CAPPUCINO [RRS+10] is a platform for executing web services in ubiquitous environments, with support for context-aware adaptation of services. CAPPUCINO uses SCA to specify both the applications and their associated autonomic control loops, where each control loop is in charge of monitoring the execution of one SCA application and of adapting it with regards to the evolutions of the environment.

CAPPUCINO has been implemented using FraSCAti as the SCA runtime support, and using an FScript engine embedded as an SCA component to provide reconfiguration capabilities. Context-awareness is provided by COSMOS [CRS07], a framework for processing and representing context information, and SPACES, a mediator tool for distributing this context policies using REST. The autonomic control loop is designed as an SCA application that collects contextual information from the managed applications. The managed application uses SPACES to connect the remotely deployed CAPPUCINO elements through SCA RESTful bindings.

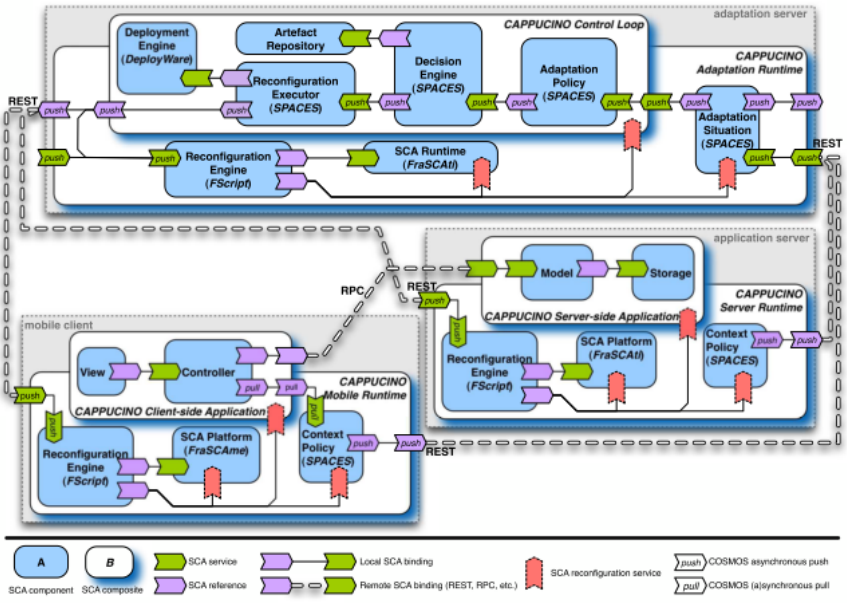


Figure 3.9: CAPPUCINO elements [RRS+10]. The control loop is hosted in the adaptation server. SPACES components are used to obtain monitoring information from the remote applications.

Context information collectors and reconfiguration engines are deployed as SCA components in each distributed location. SPACES context collectors works as agents sending their context information to a CAPPUCINO Adaptation Runtime, another SCA application, where the context is processed and a Decision Engine, and Reconfiguration Engine components create the reconfiguration plan and send it to the reconfiguration engines located in each distributed location, completing the autonomic control loop.

One of the benefits of CAPPUCINO is the explicit depiction of the autonomic control loop separated in SCA components. Being oriented to ubiquitous environments, one control loop is implemented for all the application, managed in a central Adaptation Runtime, where the context information is received, and actions are planned and sent to the respective distributed agents. Our proposition takes a similar component-oriented separation of the autonomic control loop, where we envision flexible elements in each step of the control loop, and allows to implement local control loops for each application component and a general control loop for the complete composition.

#### 3.2.3.2 Ceylon

Ceylon [MDL10] is a service-oriented framework for integrating autonomic services to build more complex applications that implement autonomic management. Ceylon considers a set of components that provide simple tasks related to monitoring and management like: monitoring a single parameter, detect a certain condition, planning a specific solution, or modifying a managed resource, f.e. replacing a storage unit for another. These components provide their functionality in a service-oriented approach and are available from a repository. The Ceylon framework aims to select those components and compose them at runtime in a loosely-coupled way, using topic-based asynchronous communication, in order to form the complex autonomic management application. Evolving conditions and requirements are supported as the composed management application can also be modified at runtime.

Ceylon considers a *Task Manager* layer that administers the management components providing them a common communication channel. Another layer, the *Strategy Manager*, supervises and reconfigures the components through the Task Manager. The Strategy Manager uses a runtime model of the autonomic management process that includes the management strategies and their associated tasks along with the management components that must be activated.

Management activities are encapsulated in Ceylon containers in order to separate the administration of the management activities from the implementation of the management task itself. Ceylon containers provide interfaces for specifying goals, event publish/subscription, lifecycle, human-machine interaction, and configuration.

Ceylon is oriented to construction of applications that perform autonomically. That means, autonomicity is a main *functional* objective in the development of the application. In our framework, we aim to provide autonomic QoS-related capabilities to already existing service based applications. Also, we take benefit of the business-level components intrinsic distribution and hierarchy to split the implementation of monitoring and management requirements across different levels, and taking autonomic activities near to the targeted services, thus enforcing scalability, which is not so clear to obtain through the Ceylon description.

#### 3.2.3.3 SAFDIS

SAFDIS (Self Adaptation For DIstributed Services) [GDA10, ADG10] is a context-aware framework for self-adaptation of distributed service-oriented applications. SAFDIS also takes the MAPE autonomic control loop as model, and implements all the phases as services. The focus of SAFDIS is in the distributed collaboration to create an adaptation plan. The analysis phase considers a short-term reasoner that can trigger simple, direct strategies, while a long-term reasoner can execute complex strategies. SAFDIS considers the utilization of different algorithms provided as services to obtain the best strategy to reach an objective state. As a result of the

planning phase, a set of abstract actions are produce, which must be translated in the execution phase to concrete platform-dependent actions.

SAFDIS collects the context-aware monitoring information from the WildCat engine. SAFDIS provides migration of services as adaptive actions, which are executed using OSGi. For this matter, services are remotely deployed and started using OSGi. Whenever a service needs to be migrated, their state is retrieved, then the service is redeployed on the new destination and the old state is copied.

The main contributions of SAFDIS related to our work are the separation between abstract actions generated by the planning phase, from the concrete actions that depend on the targeted platform, which makes possible to address heterogeneous service platforms; the other interesting concept is the design of an analysis phase that can react at different depth levels, a feature that is certainly possible within our framework. SAFDIS does not, however, considers the runtime evolution of the adaptation strategies themselves.

## 3.3 Comparison

Although the frameworks and tools presented in Section 3.2.2 support most or all of the phases of the autonomic control loop, few of them allow to modify the composition of the autonomic behaviour at runtime.

As a synthesis of the frameworks and tools presented, we have categorized the existent works according to the following features that we consider important for a complete adaptation framework, and illustrate this comparison in Table 3.1.

- Monitoring: the framework or tool considers, as part of itself, a monitoring architecture that may include probes, sensors, request interceptors, or that can perform subscription to event sources.
- Analysis: the framework or tool is able to continuously check for conditions using the monitored data, and it can detect or predict deviations from the expect behaviour.
- Planning: the framework or tool is able to decide some action, plan a recomposition, or choose some strategy to take the service to a certain state.
- Execution: the framework or tool has itself the capability to execute concrete actions to adapt or modify the target system by means of its own actuators, or it allows to describe the actions that must be executed on a targeted infrastructure.
- Extensibility: the framework or tool allows to incorporate or design new elements in some of the phases, in addition to those already provided, by providing a toolkit or API to provide custom elements.
- Flexibility: the framework or tool can include new elements either statically at design time, or dynamically at runtime in order to change or adapt the behaviour of the control loop.
- Scope: kind of applications that the framework or tool is applied to, or if it is a generic approach.
- Communication Type: technology used to transmit information through the control loop.

The support given to each phase varies. In Table 3.1, we utilize '+' to indicate that the support is restricted to a specific type of applications or that it may be constrained to modifying the interactions between the elements provided by the framework. On the other side, we use '++' to indicate that the support of the framework allows to have different implementations or develop custom elements to accomplish a task.

With respect to extensibility, we use '+' to indicate that the custom elements added are restricted to different combinations of the existing elements, and '++' to indicate that custom elements may be introduced.

In addition, the analyzed works vary not only respect to the support they give for the phases of the autonomic control loop, but also in the level of coupling between each phase. In our work, we promote that separating the implementations of each phase helps in creating more flexible control loops, so we also analyze how modular are the implemented phases. The Table 3.1 includes a closed line enclosing the phases whose support is tightly coupled.

Existing works are usually targeted to a specific kind of environment, or application, and profit of the respective feature of those domains. The works can be analyzed from several points of view.

Respect to the support for Monitoring and Analysis, older tools tend to tightly associate the monitored characteristics to the requirements expressed in SLAs, so that the monitoring framework is built around these requirements. According to their goal, WildCat, Kieker and RESERVOIR does not target specifically services, however WildCat and Kieker are generic enough to be applied in that context too. RESERVOIR, on the other side is a more modern project targeted at virtualised resources which are, however, used to host services. In fact, RESERVOIR is a more integrated approach for the service provisioning infrastructure.

The support for planning strategies is low or inexistent, and is mostly limited to execute predefined modifications. Regarding the flexibility aspect, only WildCat and GT4 were identified as capable of modifying their monitoring and analysis activities at runtime. Although their focus is different, both rely on hierarchies where nodes can be connected or disconnected at runtime. GT4 is, however, part of a complete and scalable environment targeted at heterogeneous infrastructures, while WildCat is intended as a support for building more complex tools.

Regarding the works that aim to implement the complete adaptation loop, Autopilot is one of the first to integrate all phases, however all the logic is comprised in its core and hardly modifiable. Rainbow provided a generic model for separating the managed application from the control loop with still a rather centralized approach. StarMX and Ceylon share that they intend to compose an autonomic control loop by composing smaller units that implement specific tasks, however, as told, Ceylon is oriented to create autonomic applications, while StarMX intends to provide self-management capabilities to existing Java-based applications. Ceylon provides also a runtime adaptation of this autonomic composition, while StarMX provides only the technical support for gluing the management tasks. Cappucino and Entropy can be seen as an application of an autonomic architecture for constrained application domains, as ubiquitous services, and virtualised resources, consequently they do not provide much flexibility for adapting the autonomic behaviour to other applications.

The most similar in terms of capabilities to our approach are Ceylon and Dynaco. As said before, Ceylon targets the development of autonomous application instead of providing autonomic capabilities to existing applications. Dynaco was developed to support adaptable components computing and is a flexible framework that allows their internal components to be also adapted. In our work we extend this adaptable approach to a service-based environment allowing multiple elements inside the phases of the autonomic control loop in a coordinated way according to the adaptation needs of the component.

| | WildCat | Kieker | VRESCo | PADRES | SERVME | SLAM4M | DIPAS | GT4 | RESERVOIR | Autopilot | Rainbow | StarMX | Entropy | Dynaco | Cappucino | Ceylon | SAFDIS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Monitoring | ++ | ++ | + | ++ | - | + | + | ++ | + | ++ | ++ | ++ | + | + | + | ++ | + |
| Analysis | ++ | ++ | ++ | ++ | + | ++ | ++ | ++ | + | + | + | + | + | ++ | + | ++ | ++ |
| Planning | - | - | + | - | - | - | - | - | + | + | + | ++ | + | ++ | + | ++ | ++ |
| Execution | - | - | + | + | + | - | - | - | + | ++ | + | + | + | ++ | + | ++ | + |
| Scope | generic | java / web services | web services | services (BPEL) | service provision | services | service workflows | grid services | virtualised resources | grids | generic | java apps. | virtualised resources | components | ubiquitous services | development of autonomous applications | services |
| Extensibility | ++ | ++ | + | + | ? | + | - | ++ | + | - | ++ | ++ | - | ++ | - | ++ | + |
| Flexibility | runtime | - | - | design | ? | design | - | runtime | design | runtime on/off sensors | design | design | - | design | design | runtime | design |
| Comm. | events | AOP +intercep. | SOAP intercep. + TCP | event bus | ? | events | WSRF / brokers | WSRF / WSN | ? | mediation middleware | ? | JMX | ? | ? | SCA REST | middleware for events | ? |

++ : strong support / extensible
+ : basic support / restricted
- : no support
? : not specified

Table 3.1: Comparison of analyzed works. The closed lines encircling marks indicate tightly coupled concerns.

## 3.4 Summary

In this chapter we have presented a set of relevant solutions for tackling the phases of the MAPE autonomic control loop, first in a separate way, and later presenting solutions that attempt to integrate them in a coordinated way to provide complete adaptation loops.

There is a vast set of solutions, and the set presented does not intend to be exhaustive. Nevertheless, it is possible to see that the different approaches must tackle several common problems like scalability, efficiency, low intrusiveness, extensibility and flexibility.

When targeting service based applications, loosely coupling, dynamicity and autonomic adaptation become important matters. The solutions presented tackle these concerns, but usually lack and integrated approach that considers the different levels of a service-based application.

In the next chapter, we present a background on the current support for integrated support for adaptation on a service oriented applications, and position our work respect to this state of the art.

# 4 Positioning

## Contents



This chapter presents the details of our proposition in the light of the context and the state of the art that have been presented in Chapters 2 and 3.

Section 4.1 presents the requirements that we have identified as important for a proper solution to the problematics that we found in the current solutions, and details the scientific contribution of our proposal. Section 4.2 describe how our solution helps in solving the problematics presented and how service-based applications can benefit from it. Finally, we summarize in Section 4.3.

## 4.1 Requirements and Proposed Solution

After having presented the context in which our contribution develops, and the related state-of-the-art, we present our scientific contributions with respect to that context. We start by analyzing the current situation, then we point out a list of requirements that we consider that should be included in a proper solution, and then we describe the scientific contributions of our solution and how it addresses the presented requirements.

### 4.1.1 Analysis of Current Situation

Chapter 2 introduced the service-orientation area, characterizing service-based applications as dynamic, loosely coupled relationships of heterogeneous services. Such applications are subject to evolution. Evolution may be triggered by different reasons, and may happen at different levels in a service-based application:

- Infrastructure level. An increase in network latency, a disruption in availability, a performance decrease of a service, may require to change the utilization of computational resources by, f.e., moving services to other supporting infrastructures, or increasing the number of resources used in the current infrastructure.
- Composition level. A modification in a composition, f.e., by changing a binding, may require an adjustment in other parts of the composition, as the new binding may influence the overall QoS of the composition.

- Management level. A modification in the parameters of an SLA, or the introduction or modification of new SLOs may require to modify parts of the application to comply with the new conditions.

All these evolutions may happen, often in ways that cannot be foreseen when designing the application, f.e., due to the introduction of new requirements once the service-based application is running, or by unexpected runtime conditions, a possibility that is increased by the fact that not all the services are usually under the control of a single entity.

Moreover, these sources of evolutions are not independent. For example, a composition may have some services deployed in an external infrastructure. If that infrastructure is suddenly unavailable then the services must be redeployed in another infrastructure. The new infrastructure may have another characteristics, for example, a higher cost, and then the total cost of the composition changes. If an SLO has a restriction about the cost of the composition, then the composition may need to be reconfigured, for example, switching other services for some alternative cheaper versions. In any case, a modification in some part of the service composition, may trigger additional changes in other levels.

To appropriately support these evolutions, it is necessary to efficiently perform monitoring and management tasks over the services, and ensure that their runtime behaviour complies with their contractually defined QoS requirements. However, the monitoring and management of a service-based application is not a simple task, as it crosscuts all the levels of an SOA, as mentioned in Section 2.1.2. Consequently, the dynamic evolutions that we have mentioned also have consequences in the execution of the transversal monitoring and management tasks:

- Infrastructure level. If new resources for hosting services are added, these resources need to be monitored. However, different providers may offer different technologies and limited capabilities to do monitoring and management on them.
- Composition level. A modification in the composition of a service-based application must be considered in the monitoring and management of the complete composition, in particular in the way to collect and compute the required information, and in the possible targets for adaptive actions.
- Management level. The introduction of new SLOs or the modification of existing ones may require to collect information that was not being collected before. At the same time, the introduction or removal of services may impact the place where an adaptive action may be taken.

The situation is thus, an evolving environment for services where changes can occur at different levels, and the services need to adapt to them. As most of this changes can occur at unpredictable moments and comprise unforeseen conditions, autonomicity seems a promising approach to address these adaptation needs.

The vision we aim to contribute to its concretisation is thus: having service-based applications that can be adapted, preferably autonomously, to a constantly evolving world, with none or minimal disruption of their functionality and complying with a set of required QoS characteristics, considering that these QoS characteristics can also evolve.

### 4.1.2 Requirements

We mentioned in Section 2.1.4 a set of challenges in service-orientation that have the objective of providing adaptability at several levels. Section 2.5 presented the topic of autonomic computing as a discipline that can help in the development of self-adaptable services by means of autonomic control loops. Section 3.2 presented some works that have advanced the research in the attachment of autonomic control loops in complex systems.

We have identified the following aspects that we consider key for a monitoring and management system that can make the above vision effective:

- **Extensibility**. Given that the monitoring and management conditions are manifold and highly dependent on the specific characteristics of the service-based application, the monitoring and management system must allow the definition of custom elements, instead of (or in addition to) fixed predefined libraries.
- **Flexibility**. Given that not all the monitoring and management requirements, nor the conditions under which the service-based application will execute can be completely foreseen at design time, the monitoring and management system must allow the introduction of new elements at runtime, possibly modifying the monitoring and management system itself.
- **Heterogeneity/openness**. As the landscape of services and providers is technologically heterogeneous, the monitoring and management system must be able to incorporate information from and execute actions on services with different monitoring and management capabilities and/or access protocols.
- **Efficiency**. In a context where the actual services that form part of a composition may be highly geographically distributed, possibly including many of them, and where, nevertheless short response times are usually expected, the collection of information and decision making must be done in a timely manner.
- **Autonomicity**. Given the complex nature of service-based applications, the monitoring and management system must be able to take autonomic decisions as much as possible in order to minimize the requirement of human intervention.

In Section 3.3 we identified that most of the approaches presented in Section 3.2.2, though providing autonomicity to applications, address most of the time specific kinds of applications, and few of them provide the extensibility and flexibility needs over the autonomic control loop itself.

The situation is, thus, an extensive set of tools and frameworks for monitoring and management, sometimes in an autonomic manner, that address specific technologies, with a general lack of uniformity; or that cover only part of the levels required in an SOA; and with rather strong support for triggering adaptations in the target applications, but with few space for adapting the monitoring and management behaviour itself.

### 4.1.3 Solution and Scientific Contribution

Said that, we state as our research objective to **improve the adaptability feature implementation in service-based applications by providing a generic framework** that

- gives a common and efficient means to monitor and manage service-based applications,
- allowing to introduce autonomic behaviour at runtime and
- allowing to modify their adaptability features if needed, in order to support evolving management requirements.

In order to provide such a solution, we must target the requirements previously stated. We aim to integrate the different steps of an autonomic control loop in a service-based application, in order to provide autonomic behaviour to services and to be able to take timely adaptation decisions. In order to tackle the complexity of designing such a framework, we choose to adopt a component-based design to present our proposition. Section 2.4 presented the SCA model where CBSE properties are used to capture the composability, encapsulation and reuse needs of service-based applications, while also providing abstraction from specific implementing technologies. We believe that a component-based approach like that used in the SCA design model can appropriately model the introduction of autonomic support in service-based applications.

However, as mentioned before, the SCA approach does not include concerns about runtime evolution, as SCA is a design-oriented specification. We need, thus, a proper implementation

support capable of dynamic evolution. We support our solution with an implementation over the GCM/ProActive middleware, which provides support for dynamic reconfigurations, and is capable of handling large scale deployment concerns, allowing to take on the monitoring and management concerns at all the levels of an SOA.

We present our solution as a **generic component-based framework that allows to introduce evolvable monitoring and management features into component-based service-oriented applications, and which can be used to support efficient and autonomic behaviours**.

Our proposition separates the MAPE autonomic control loop in its different phases, and models them as separate components that are attached as close as possible to the target services they aim to manage. Consciously following the MAPE control loop implied architecture allows us to support **autonomicity** in the adaptation if required. Nevertheless, if autonomicity is not required, our solution can support downsizing, keeping only the Monitoring and Execution components. These components interact with the target service by means of sensors and actuators included respectively in the monitoring and in the execution phases, leveraging the information available through the specific technologies made available by the service provider to a common ground where they can be used by the other phases. This way we tackle **heterogeneity/openness**.

The different components that implement the MAPE loop can collaborate in order to decentralize the monitoring and management tasks, and propagate the monitoring information they collect to the level where it is needed to take appropriate decisions. By keeping this information close to the services where it is obtained, and making it available through their defined interfaces, we aim to provide **efficient** propagation.

The component-based approach allows us to provide different implementations (**extensibility**) for each phase, which can be developed in rather independent ways and interact through defined interfaces. As we have presented through Section 3.1, there exist many different tools and technologies that can be used to address each phase, and our aim is to make possible for them to collaborate in an easier way.

We organize the collaboration between the components implementing the MAPE phases for each service, according to the architecture of the service-based application. As we rely in an SCA design, we profit of the explicit offerings and dependencies stated by the SCA Components, as well as the hierarchical composition in order to establish the collaboration links between our components.

Finally, we propose our framework to be reconfigurable, so that we consider the dynamic attachment and removal of the MAPE components to the target service, with the aim to support different management requirements (**flexibility**). In fact, it is not required that all the components of the MAPE loop be available in all services, but only those that are needed for providing the autonomic behaviour over the composition.

By relying in a component-based approach that can be added or removed at runtime, we also release the programmer of the service-based application from the burden of manually including monitoring and management concerns when developing the application.

## 4.2 Benefits of the solution for supporting autonomicity

Besides our solution offering each individual service the support for plugging its own autonomic behaviour, it also supports a distributed interaction of control loops.

One of the motivations to have a distributed interaction of control loops is the possibility of making them collaborate to implement some global autonomic management task. Featuring autonomic management when considering an application that is not standalone but built out of independent and loosely coupled services is not simple as it raises several issues. One of them is the appropriate distribution of an autonomic task designed from a global point of view, into

several smaller tasks that can collaborate to achieve a common goal whenever scalability is a concern, and the distributed nature of the application to manage wants to be leveraged. In fact, due to the wide distribution that can be achieved by a service-based application, it is not in general reasonable to maintain at all times a complete and updated model of the situation, upon which to base the decision taking. It is also not possible, due to the evolving characteristic of a composed service to foresee all the modifications that can be needed, as the composition of the service may change and one action predefined for a particular situation may be not applicable once a service has been replaced because the target service does not belong to the composition anymore, or because the new service has a different management interface.

It is, in general, more common to devise solutions to individual, simple and autonomic management tasks like the replacement of a single service, the modification of a parameter of a service, or the addition of a new element in a composition. However, for implementing effective autonomic tasks that take into account, for example, the complete situation of a composed service-based application before taking a decision, an appropriate interaction between several simple tasks must be possible. In this sense, our framework does not enforce a particular kind of solution, nor does it define or promote a method for dividing a autonomic global task into smaller ones. Nevertheless, our framework intends to facilitate the implementation of such tasks by providing an architecture where the different elements that are necessary to implement the autonomic task can be inserted and can further interact. Concretely, through our solution, we support the insertion of simple management execution tasks over a running service-based application. Simple execution tasks refers to specific tasks for a service according to the nature of the service and its implementation. These tasks may include incrementing the number of threads to run an application, replace a given required or client service by an equivalent one from another provider, add or remove bound services, migrate a service from one hosting environment to another. Besides including individual execution tasks, making them available through defined interfaces aims to facilitate the interaction of these different tasks, and allow the composer of the application to provide some *self-** features to the resulting compound service.

To better illustrate the potential benefits, we briefly highlight below, and in more details in Chapter 8, how the framework can be used to implement an autonomic solution featuring self-optimizing, self-healing behaviours guided by SLOs inserted in each component, and propagated only until the required level, in a “bottom-up” way.

### 4.2.1 A *self-** scenario

In Section 2.5.3 we mentioned some advantages that *self-** features can provide to service-based applications, by automating common management tasks that may be required at situations that are, in general, unforeseeable.

Consider, for example, a service-based application for image storing that comprises many replicas of its storage components in order to reduce the access time to its contents. We illustrate how self-optimized and self-healing behaviours can be obtained still within the constraints given by the associated SLOs.

A condition over this service may require that the average delivery time of an image be less than 5 seconds. If the application becomes popular, the number of concurrent accesses may provoke an increase of the response time. The manager of the service may decide to create a storage replica in another environment with better performance so to restore the average response time to the initial requirement. This is a simple autonomic **self-healing** response to an increase in the reading of a metric about response time. However, if the new hosting environment becomes suddenly unavailable, the response time will naturally increase, and the manager will need to deploy new replicas in a third environment to support the incoming requests. Figure 4.1 shows an example of this situation, where the square boxes represent the elements that are dynamically attached by our framework.

In any case, the use of new replicas in externally hosted environments is not free for the manager, so he may need to introduce an additional condition limiting the maximum cost of having

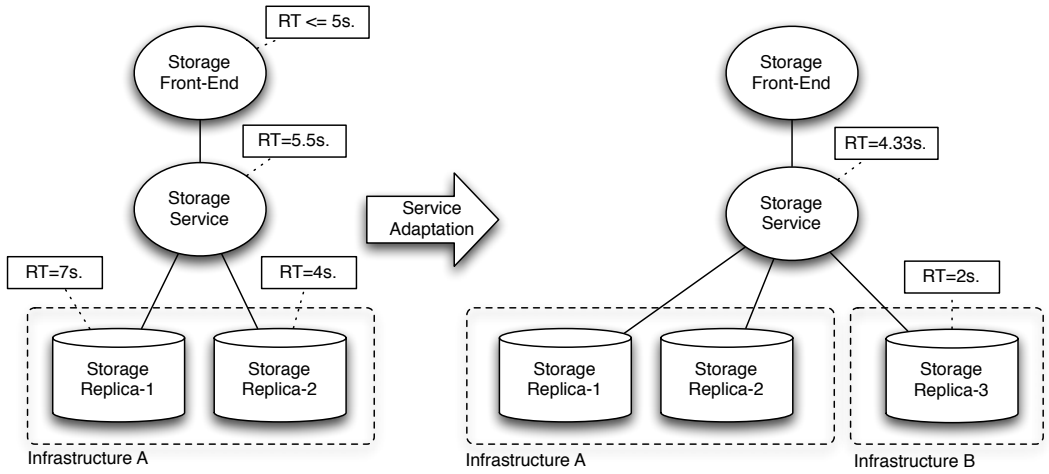


Figure 4.1: Self-Healing adaptation by deploying a new replica in another infrastructure

distributed replicas in different environments. For each of these environments the pricing model may be different and require the reading of different metrics in order to compute it. By inserting the appropriate logic in each replica, the manager can achieve a common view of the cost and insert a different autonomic behaviour that takes into account both the average response time and the total cost of the replicas in order to add or remove these replicas to/from the composition, achieving a **self-optimizing** behaviour, illustrated in Figure 4.2.

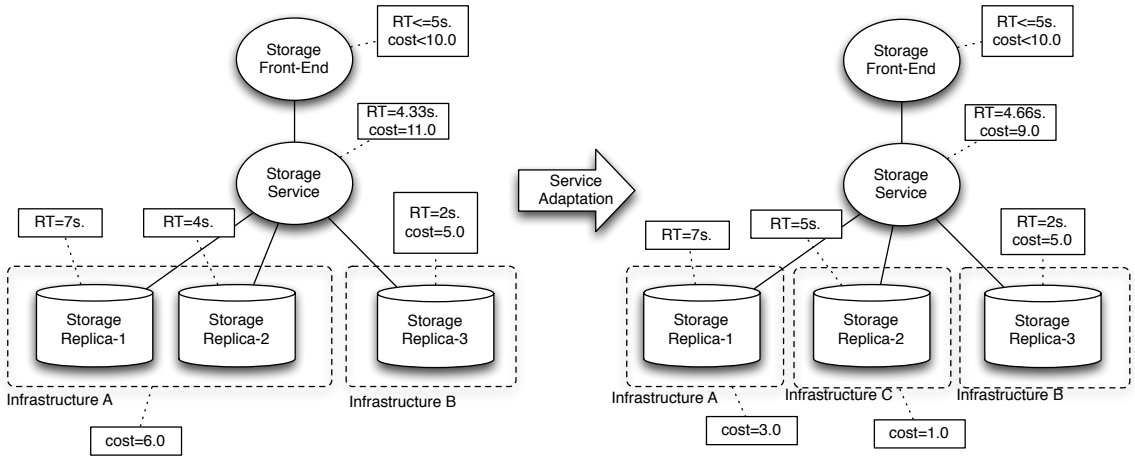


Figure 4.2: Self-Optimizing adaptation distributing the existing resources in available infrastructure, decreasing cost and maintaining response time.

### 4.2.2 Action propagation

Other kinds of action that we intend to facilitate with our framework is the implementation of local autonomic control loops. By this, we mean autonomic control loop that control a specific section of a service composition, and whose action needs not to collect a global state of the application, allowing to apply local optimizations.

A simple example, based on the same scenario of the previous section, involves attaching an autonomic action to each storage space, that considers the remaining free storage space. If a certain level of free space is reached, then a local autonomic action may compress some elements in order to made up more free space. In case no more compression is possible a notification is

sent to the *Storage Service*, which could decide to take a further action and, f.e., provide another storage unit, as shown in Figure 4.3

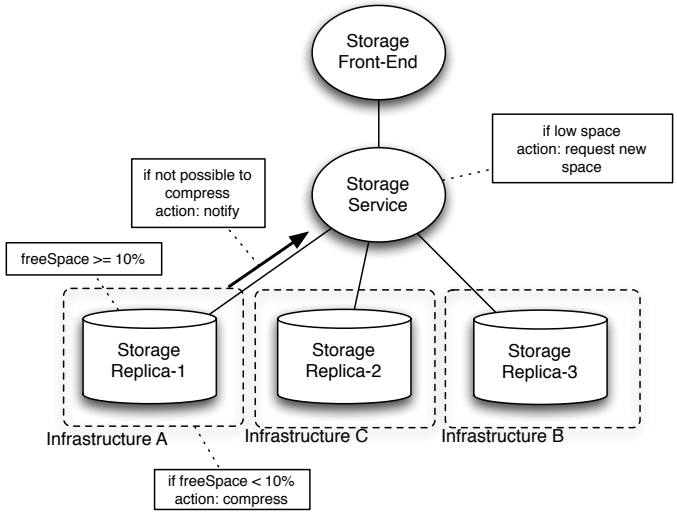


Figure 4.3: Action generated in a particular element, that is propagated to a higher level manager, which can decide upon other action.

Overall, we can reinforce that the framework allows the runtime insertion of different phases of the MAPE autonomic control loop in order to support the management needs of the application. We aim to facilitate the communication between the different capabilities that each service may include, which may comprise all or part of the MAPE loop, providing an environment where autonomic behaviours can be inserted into the services.

## 4.3 Summary

We have presented our solution in the light of the requirements that we have identified from the context of service-based applications, and the state-of-the-art. Once describe our scientific contribution, we have presented the technological background we use to support our solution and that will be used to provide a concrete implementation, namely the GCM model and its reference implementation GCM/ProActive. We describe how the features found in GCM/ProActive allow us to support the requirements for our thesis.

In the next chapter we present the design of our solution from a technologically independent point of view, and then we describe the details of our concrete implementation in the GCM/ProActive middleware, using GCM/SCA as a suitable composition framework for SCA-based applications that can be turned autonomic.

# 5 Framework Design

## Contents



In the previous chapter we have presented the background over which we build our contribution. This chapter presents the generic design of a reconfigurable component-based support for providing flexible monitoring and management capabilities to component-based service-oriented applications.

In order to not lose genericness in our description we refer to the notation of the SCA specifications to present our design. Although we are aware that SCA by itself does not provide reconfiguration features, we explain when needed the reconfigurability capabilities that are needed.

Section 5.1 presents a general view of our approach integrating all the phases of the autonomic control loop. Sections 5.2 to 5.5 describe the requirements for each phase and how we address them, presenting the interfaces design for each element. Finally Section 5.6 summarizes this chapter.

## 5.1 Overview

Our solution relies on the separation of the steps of the classical MAPE autonomic control loop. Namely, we envision separate components for monitoring, analysis, planning, and execution. These components, in the following called *MAPE Components* communicate through predefined interfaces, and are attached to each component that needs management, and possibly autonomic capabilities.

A service that has been augmented with monitoring and management capabilities becomes a *managed service*. The aim is that the attachment of these capabilities be as less intrusive as possible, and independent of the functional design of the service. This is why the proposition, from an external point of view, considers the addition at design time of a set of interfaces to a regular SCA service, to transform it into a *managed SCA service* as shown in Figure 5.1.

The additional interfaces are inserted at design time by augmenting the SCA ADL description. However, we expect that the MAPE components can be dynamically added or removed at runtime. In fact, this specific requirement goes in line with the goal of avoiding unnecessary intrusiveness in the functional objective of the managed service, by attaching to it only the MAPE

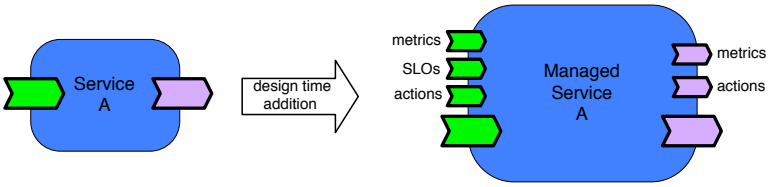


Figure 5.1: SCA component *Service A* is seen as a *Managed Service A* with additional interfaces

components that are needed. This also implies that at some moments the additional interfaces may be available but without any component attached. Depending on the component implementation, this may require the existence of dummy or empty components to those interfaces.

The general structure of a service component that follows our design is shown for an individual *Service A* in Figure 5.2. *Service A* is augmented with one component for each phase of the autonomic control loop: *Monitoring*, *Analysis*, *Planning*, and *Execution*. This way, *Service A* is converted into a composite *Managed Service A*, indicated by dashed lines. The original "service" and "reference" interfaces of the service component are promoted to the corresponding interfaces of the composite, so that from a functional point of view, the composite *Managed Service A* can be used in the same way as the original *Service A*. In both Figures 5.1 and 5.2, the interfaces that are added by our framework are shown as smaller interfaces, only in order to differentiate them from the original interfaces of the service. The objective of these additional interfaces is to be able to interact with the corresponding interfaces of other managed services.

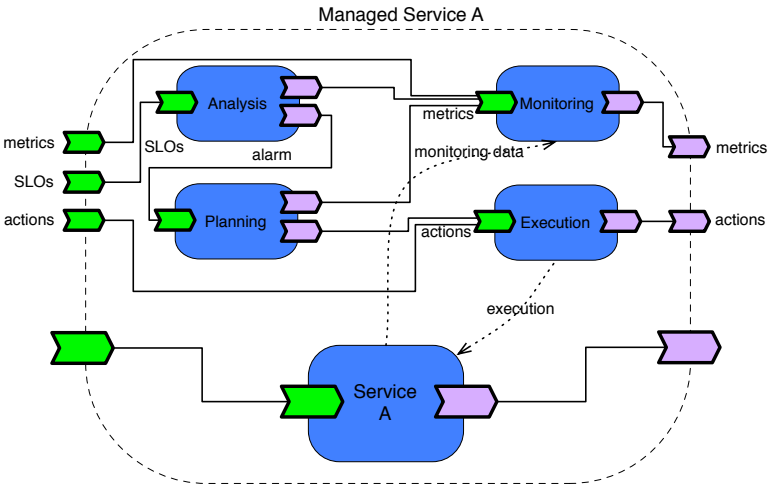


Figure 5.2: SCA component *Service A* with all its attached monitoring and management components

Figure 5.2 also shows dashed arrows connecting the *Monitoring* and *Executing* components with *Service A*. These arrows indicate that the *Monitoring* component must be able to collect monitoring data from the running service according to the specific ways that this target service allows. On the other side, the *Executing* component must be able to execute actions over the target service using the specific means for that service. These two steps are the only phases that enforce communication of the elements of the framework with the managed service.

The intended functioning of the framework is as follows. The *Monitoring* component collects monitoring data from *Service A* using the specific means that *A* may provide, f.e. using sensors for reading parameters of the service, or intercepting requests on the interfaces of the service. Using the collected monitoring data, the *Monitoring* component computes a set of metrics and makes them available through a *metrics* interface. The computation of metrics may include collaboration with the *Monitoring* component of other services. The *Analysis* component provides

an interface for receiving and storing SLOs. At runtime, the *Analysis* component checks the SLOs using the appropriate metrics that it obtains from the *Monitoring* component. Whenever a condition is not fulfilled, the *Analysis* component sends an alarm notification to the *Planning* component. The *Planning* component uses a strategy and creates an adaptation plan as a sequence of actions that can modify the state of the service and take it to a desired objective state. For taking decisions and feed the required parameters of the strategy, the planning component can obtain information from the *Monitoring* component through its *metrics* interface. The sequence of actions created by the *Planning* component are sent to the *Executing* component. The *Executing* component executes the actions on the service using the specific means that the service allows. As we envisage the possibility of having actions that include other services, the *Executing* component is able to delegate some actions to the *Executing* component of other services through an *actions* interface. This way, the loop is completed and the new information collected by the *Monitoring* component will reflect the new state of the *service*.

Although simple, this component-oriented view of the autonomic control loop has several advantages:

- First, by separating the control loop from the component implementation, we obtain a clear **separation of concerns** between functional content and non-functional activities. In fact, the only point where the framework must be connected with the target service are the *Monitoring* and *Executing* phases, in which the specific means to connect to the target service must be used.
- Second, the component-based approach allows to have separate implementations of each phase of the loop. As each phase may require complex tasks, we abstract from the specific implementation that each service may require and let the phases to communicate through predefined interfaces, so that each phase can be implemented by different experts.
- Third, as each phase can be implemented independently, we allow to compose each phase to have possibly multiple components, for example, multiple sensors, condition evaluators, planning strategies, and connections to concrete effectors as it is required, so that the implementation of each phase can be improved at runtime by applying dynamic reconfigurations if needed.

As we have mentioned before, a managed component comprises the interfaces that we have designed for communicating with other components. These interfaces are also intended to be used by external applications to be able to modify or adjust the composition of the control loop. At the same time we do not impose as a requirement that all the elements of the MAPE loop must be present in all the managed services. In fact, one of the advantage of using a reconfigurable component-based approach is that it is possible to introduce only the phases that are required in each managed service. However, a design decision is needed when specifying the interfaces that a component will have available for the monitoring and management tasks.

As a simple example, consider a component that represents a storage service, and provides some basic operations to read, write, search and delete files. In order to get information about the performance of the storage service, a *Monitoring* component can be added and expose metrics about the average response time for each operation, and the amount of free space. As an evolution, some non-functional maintenance actions can be exposed to compress, index, or tune the periodicity of backups. These actions can be exposed by adding an *Executing* component that can execute them over the storage service. Now the managed storage service exposes some metrics, and exposes an interface for executing maintenance actions. However, the storage service is still not autonomic and the reading of metrics and execution of maintenance actions are invoked by external entities. A next evolution can consider adding an autonomic behaviour to avoid filling the capacity of the storage service. An *Analysis* component can be added and include a condition that checks the amount of free space, and in case it is less than, f.e., 2%, it triggers an action oriented to increase the amount of free space. The decision about what action to take can be

delegated to a *Planning* component, which may decide to call the *Executing* component to carry on one of the available maintenance actions.

Depending on the management needs, any evolution of the storage service can be used. If the autonomic behaviour described is not needed anymore, then the *Analysis* and *Planning* components can be removed and return to the simple version of the storage service. The three versions mentioned of the storage service are shown in Figure 5.3.

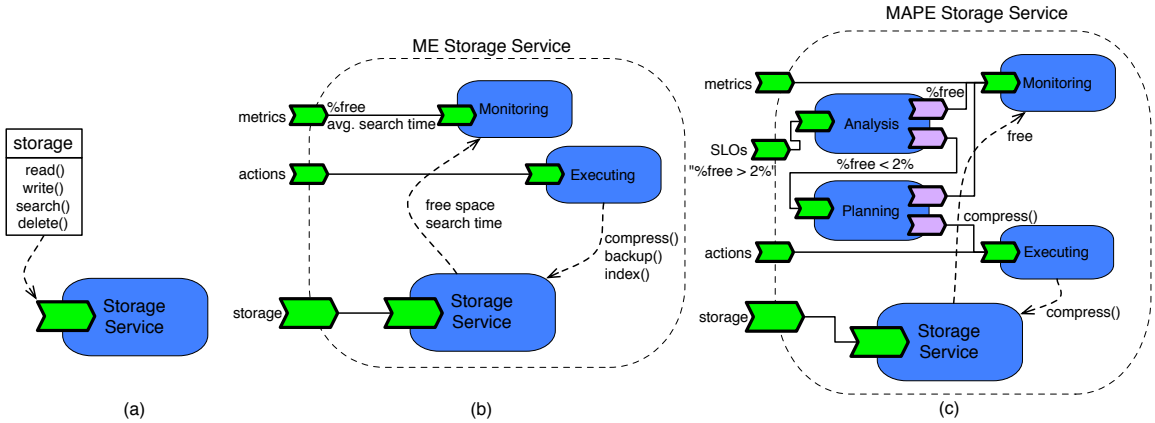


Figure 5.3: (a) Storage service in its basic version, (b) with Monitoring and Executing components, (c) with all the MAPE components

In the following, we describe the components considered in the monitoring and management framework, their function and the design decisions that have been taken into account.

## 5.2 Monitoring

Monitoring usually involves the collection of information from the target service (sensing of data), storage, filtering, and processing of the sensed data to obtain a set of *metrics* that is made available for other components.

The objective of the *Monitoring* component is to collect the required data from the service activity and expose it as a set of metrics that can be queried from other components. For collecting the monitoring data, the *Monitoring* component must include the specific sensors or, alternatively, support the communication with the sensors that are provided by the target service according to a particular protocol. In fact, the *Monitoring* component can itself follow a component-based approach, where sensors and metrics processing elements can be dynamically plugged and unplugged in order to monitor only the values that are needed at a certain time, and avoiding unnecessary overheads. By using the specific means to access the information provided by the target service, the *Monitoring* component is effectively attached to the service, which becomes a “monitored service”.

Figure 5.4 shows the basic design of our *Monitoring* component, and its interfaces. Actually, the *Monitoring* component requires, at least, one service interface, here called *metrics-service*, with the *metrics* signature. This interface allows external components to access and query the metrics computed by the *Monitoring* component. Additionally, the *Monitoring* component has zero or more references called *metrics-reference* that also provide the *metrics* signature, and whose aim is to obtain monitoring information from other *Monitoring* components. Instead of having one reference to all possible components, we consider that the *Monitoring* component needs a reference to the *Monitoring* component of each service referenced by the target service, in order to properly select the source of external monitoring information, as shown in Figure 5.5. In Section 7.2 we describe how we profit of the introspection capabilities of GCM in order to dynamically create a *Monitoring* component that can be associated to each service.

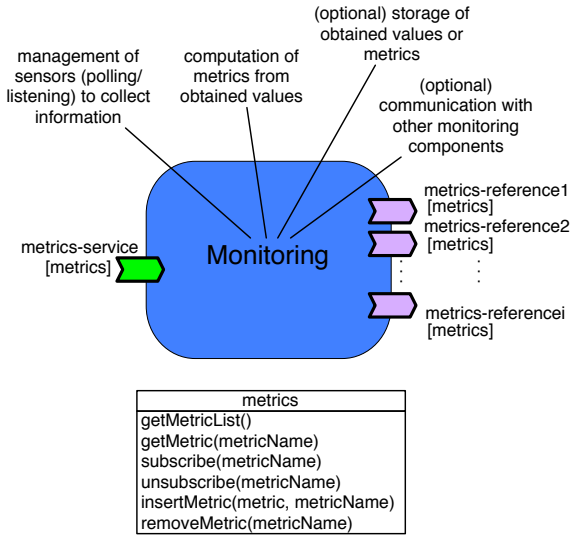


Figure 5.4: Basic SCA monitoring component

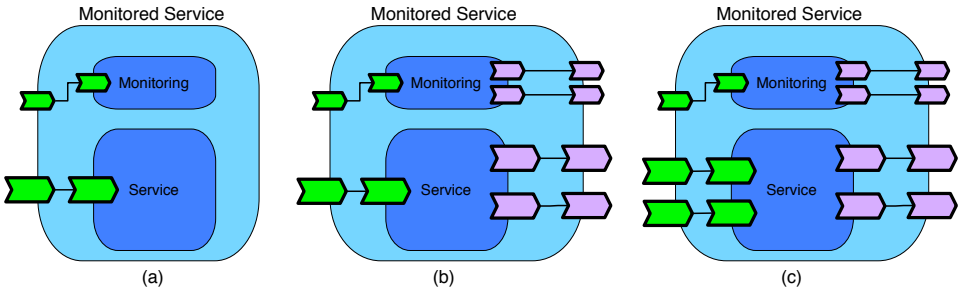


Figure 5.5: Monitoring Component inside a SCA component with (a) one service interface (b) one service and two references, (c) two services and two references

In the presence of a high number of services to be monitored, the computing and storage of collected information can be a high-demanding task, specially if it is done in a centralized manner by, f.e., gathering the monitored information at a single point of analysis before computing the metrics. Consequently, the monitoring task must be as decentralized and low-intrusive as possible. In our approach, the *Monitoring* component attached to each monitored service is responsible for collecting and computing the metrics related to that service, so that the tasks of collection and computing of metrics are distributed among all services. This approach is decentralized and specialized with respect to the monitored service.

However, some metrics may require information that is available in other services; for example, a metric that aims to obtain the energy consumption of a composition may require to obtain the energy consumption of all the services involved in the composition. To address this situation, the *Monitoring* component is capable of connecting to the *Monitoring* components of other services through their *metrics* interface. The set of *Monitoring* components are inter-connected forming a hierarchy that reflects the composition of the monitored services. This arrangement provides a "monitoring backbone" where the metrics collected at each service can flow and can be used by other components.

Figure 5.6 shows an example of an SCA composite $C$ with two inner components $A$ and $B$, and that uses two external services $D$ and $E$. The design below shows the *Monitoring* components $M_a$, $M_b$, $M_c$, $M_d$ and $M_e$, attached to each component connected forming a "monitoring backbone" with bold lines. $M_c$ is connected to $M_a$, because the service interface of $C$ is connected to $A$; and to the *Monitoring* components of both external references, $M_d$ and $M_e$. In the same way, $M_a$

is connected with $M_b$ because $A$ has a reference on $B$; and $M_b$ is connected to $M_c$ as $B$ has a reference on $D$ promoted through the composite $C$. Finally, $M_a$ also has a reference to an external service $E$ promoted to $C$, so $M_a$ is also connected to $M_c$.

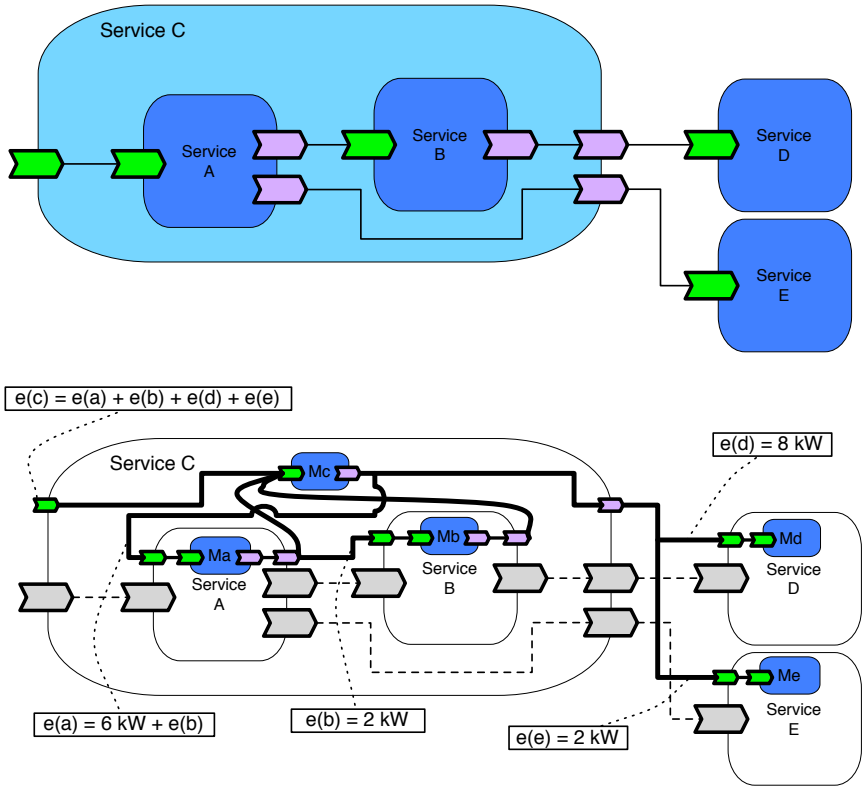


Figure 5.6: An SCA application, and the inner "monitoring backbone"

Figure 5.6 also shows an example of a *metric* named "energy" ($e(i)$) for each component $i$. Each *Monitoring* component $M_i$ is in charge of computing its value $e(i)$ as the sum of its own energy metric, and those of its references. In the case of the composite $C$, the value $e(c)$ is the sum of the values of both internal components, $e(a)$ and $e(b)$, and of its references $e(d)$ and $e(e)$. Using the connection between the different *Monitoring* components, the total value $e(c)$ is computed by $M_c$ and exposed through its *metrics* interface. Note that the means for computing the energy metric for each component may be different, depending on the characteristics of the implementation; however, once the value is computed in the corresponding *Monitoring* component, it is accessible in a uniform way by the other *Monitoring* components.

From the point of view of its input and output, the *Monitoring* component receives a set of monitored data, and produces a set of metrics. The *Monitoring* component abstracts the way to collect the values that it needs. The collection can rely on a simple periodic reading of a value, f.e., by reading the Unix file `/proc/loadavg` to obtain the average CPU load; but arbitrarily complex means can be considered, f.e., detecting single events like the sending of a request and the reception of its response, that can be used to obtain the duration time of a request; another approach may include the collection of event streams that can be used with a Complex Event Processing engine to correlate events and help to detect certain conditions.

The *Monitoring* component should allow two ways to expose the metric it computes. In a *push* mode, the *Monitoring* component receives requests for subscription to *metrics* and upon each change in the value of this metric, it sends an update notification to the requester. In a *pull* mode, the *Monitoring* component receives a request for a metric, and replies with the current value. Depending on the application and the nature of the monitored value, one mode may be more convenient than the other.

### 5.2.1 Interfaces

We describe the main methods on the *metrics* interface of a *Monitoring* component. We expect that an implementation of this component should extend this interface.

- `List<String> getMetricList()`. Obtains the list of available metrics in the *Monitoring* component as a list of names.
- `MetricValue getMetric(metricName)`. Obtains the value of the metric indicated by `metricName`.
- `MetricValue getMetric(metricName, params[])`. Obtains the value of the metric indicated by `metricName`, including optional parameters for evaluating the *metric*.
- `subscribeMetric(metricName)`. Subscribes to be notified of changes in the value of the metric indicated by `metricName`. The ability to communicate these changes must be support by the *metric*.
- `subscribeMetric(metricName, params[])`. Subscribes to be notified of changes in the value of the metric indicated by `metricName`, including optional parameters for evaluating the *metric*. The ability to communicate these changes must be support by the *metric*.
- `unsubscribeMetric(metricName)`. Unsubscribes from changes to the value of the metric indicated by `metricName`.
- `insertMetric(metric, metricName)`. Inserts a new computation logic, named `metric`, for computing the value of a metric named `metricName`.
- `removeMetric(metricName)`. Remove the metric indicated by `metricName` from the *Monitoring* component.

In Section 7.2 we describe our implementation of this interface in the context of the *Monitoring* component that we have implemented over the GCM/ProActive platform.

## 5.3 Analysis

The Analysis phase in our framework is expected to check the *SLA compliance* of the service. We call this component the *Analysis* component.

The *Analysis* component needs a clear description of an SLA. In our design we consider an SLA as a set of simpler terms called *Service Level Objective*s (SLOs). The SLOs are represented as conditions that must be verified at runtime.

The design of our *Analysis* component considers a service interface for receiving the SLOs that must be verified, called *SLOs*. We also consider two reference interfaces: one for notifying about a fault in the compliance of the SLA, with the *alarm* signature; and other to be able to collect the metrics required to check the compliance to the SLA, that implements the *metrics* signature. In contrast to the *Monitoring* component, which can have many *metrics* references, the *Analysis* component only needs one, because it only needs to connect to the *Monitoring* component. The basic design is shown Figure 5.7.

One of the challenges of the *Analysis* component is to be able to understand the conditions that needs to be checked, and extract the required metrics from *Monitoring* component. There exist several languages proposed for representing SLOs and the metrics they require, as presented in Section 3.1.2. Using a component-based approach inside the *Analysis* component it is possible to embed an interpreter for these languages into the *Analysis* component.

For illustrating purposes we use a very simple way to describe SLOs, where they are represented by conditions expressed as triples of the form $\langle metric, comparator, value \rangle$, expressing for example "$maxResponseTime \leq 30sec$". Other more complex expressions may involve conditions that include comparisons between different metrics, or operations on them like "$cost(serviceA) < 2 \times cost(serviceB)$".

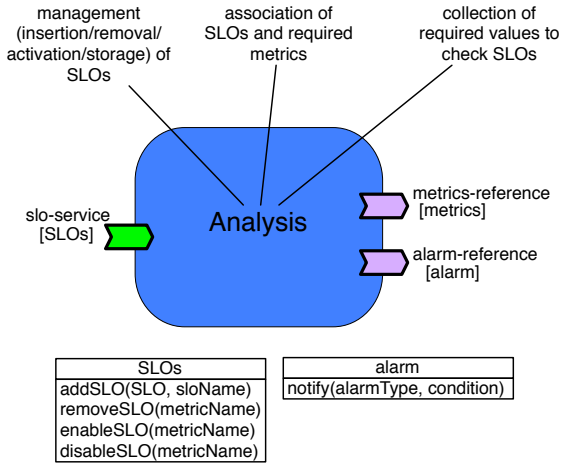


Figure 5.7: Basic SCA analysis component

Whatever the internal composition of the SLOs, it is very important that the *Analysis* component be able to extract from them the name of the metrics that are required to check the compliance, so that it can request the value of the appropriate metrics to the *Monitoring* component using the *metrics* interface. In the example of the SLO $\langle cost, \leq, 20 \rangle$, the *Analysis* component must request the current value of the metric *cost* to the *Monitoring* component.

The periodicity of the analysis phase is also a matter that concerns the *Analysis* component. The *Analysis* component can use a subscription method over the *metrics* interface to be notified of each change in the value of a metric (push mode), so that the *Analysis* component can check the corresponding condition upon each notification. Another possibility is that the *Analysis* component periodically requests the value of a metric to check (pull mode). As mentioned before, it is up to the implementation of the *Analysis* component to determine the best way, and the framework considers the possibility of using both modes with the same or different metrics.

From the point of view of its input and output, the *Analyzer* receives a set of conditions (SLOs) to monitor, expressed in a predefined language. The *Analyzer* checks the compliance of all the stored SLOs according to the metrics reported by the *Monitoring* component, and determines if the SLA is being fulfilled. In case it is not, the *Analyzer* component sends an alarm notification through a required interface. The consequences of this alarm are out of the scope of the *Analyzer* and will be mentioned in the next section. An optional capability is that the *Analyzer* be configured in a proactive way to detect not only SLA violation, but also foreseeable SLA violations, which may be more useful in some contexts, as it can allow to take actions to avoid an SLA violation, as those mentioned in Section 3.1.2.2.

One of the advantages of having *Analysis* component attached to each service is that the conditions can be checked closely to the monitored service and do not involve more resources than those of the interested services, benefiting of the hierarchical composition of the "monitoring backbone". This way, the services do not need to take care of SLAs in which they are not involved.

Figure 5.8 shows an example of an SCA application composed by three services. Service $A$ has an *Analysis* component ($A_a$), and a *Monitoring* component ($M_a$). Service $B$ and $C$ are used by $A$. Service $B$ includes an *Analysis* component ($A_b$) and a *Monitoring* component ($M_b$); service $C$ only includes a *Monitoring* component ($M_c$).

In the example of Figure 5.8, the *Analysis* component $A_a$ must check the SLO "$\langle cost, <, 30 \rangle$" over Service $A$. For checking that condition, it requires the value of the metric *cost* from $M_a$. In $M_a$, the computation of the metric *cost* requires the value of the metric *cost* from both services $B$ and $C$. $M_a$ obtains this information from the corresponding *Monitoring* components $M_b$ and $M_c$ and is able to deliver the response to $A_a$. It is worth noting that $A_a$ is not aware that the computation of $M_a$ actually required additional requests to $M_b$ and $M_c$, as this logic is hidden

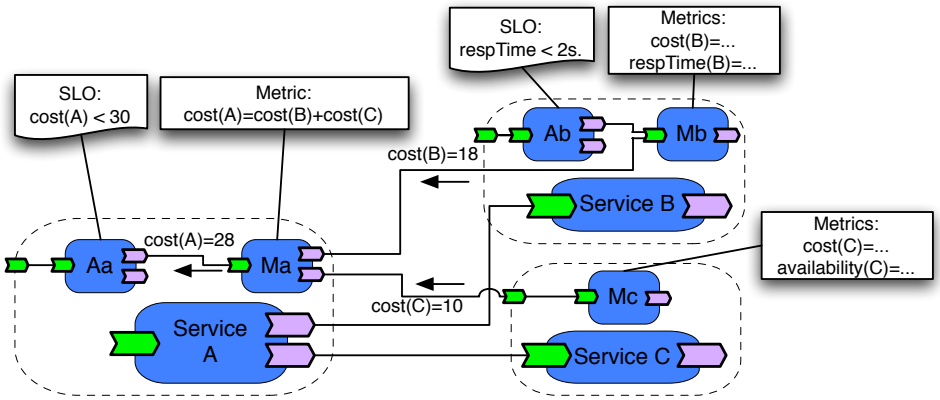


Figure 5.8: SCA Components with Analysis and Monitor components. $A_a$ and $A_b$ have different SLAs. Metric *cost* is computed in $M_a$ by calling $M_b$ and $M_c$. Intermediate promoted interfaces are not shown for clarity.

into $M_a$. In another situation, independent from the previous one, the *Analysis* component $A_b$ checks a condition related to the response time (*respTime*) metric from service $B$, which requires to read the appropriate metric from $M_b$.

### 5.3.1 Interfaces

The *metrics* interface, used for accessing the metrics available from the "monitoring backbone" has been already described in Section 5.2.1. The *alarm* interface includes one relevant method:

- `notify(alarmType, SLO)`. Sends a message indicating that there is a problem with the specified *SLO*. Depending on the value of `alarmType`, this could indicate that `SLO` has been violated, or that it has a risk of being violated.

The *SLOs* interface includes methods for managing the set of SLOs that are checked by the *Analysis* component.

- `addSLO(SLO,sloName)`. Inserts a new SLO to be checked by the *Analysis* component, and identified by `sloName`.
- `removeSLO(sloName)`. Removes the SLO indicated by `sloName`.
- `enableSLO(sloName)`. Enables the dynamic checking of the SLO indicated by `sloName`.
- `disableSLO(sloName)`. Disables the dynamic checking of the SLO indicated by `sloName`.

The *SLOs* interface allows to manage the set of SLOs by an external entity, as well as temporarily disable or enable their dynamic checking.

## 5.4 Planning

The objective of the *Planning* phase is to generate a sequence of actions that can take the service from its current state to a predefined objective state. In our case, the objective state is the condition (the *SLO*) that has been violated, and the event that triggers the computation of a plan is a notification indicating that a condition is not being fulfilled.

For creating such a plan, the *Planning* component must execute a strategy, or planning algorithm that can determine that sequence of actions. This logic can be implemented in a number of ways. On the more simple side, a strategy may be a notification to a human agent (email, SMS, etc.) who would be responsible of taking any further action; another alternative could rely

on a table of predefined actions, such that if some conditions hold, then the corresponding action is generated. On a more complex side, numerous strategies and heuristics, in particular from the artificial intelligence area have been proposed for planning a composition or recomposition of services that complies with certain desired QoS characteristics. Some of them have already been presented in Section 3.1.3, and the aim of our *Planning* component is to be capable of supporting the implementation of such existing strategies.

Given the wide range of different solutions for generating a plan, it does not seem easy to find a common interface to uniform all the possible strategies. However, most of the strategies require as input information about the current status of the service in order to guide the possible solutions. Consequently our *Planning* considers one interface for obtaining information about the state of the service, connected to the *Monitoring* component. The goal is that, through the *Monitoring* component, the strategy embedded in the *Planning* component be able to access the information it requires to feed the selected strategy.

Although a simple implementation would embed one specific strategy, our approach considers that several conditions may be supported by the *Analysis* component, consequently, several conditions may need to be checked and, if it is necessary to take some actions, different strategies may be applied upon each case. That is why we think that a component-based approach applied to the *Planning* component should be able to support different planning strategies that would be activated depending on the condition that needs to be restored. As a basic implementation of the *Planning* we consider that this must be able to support one or more planning strategies, and be able to associate a faulting condition to an appropriate strategy. Figure 5.9 shows the basic design of the *Planning* component that includes a service interface with the *alarm* signature as defined from the *Analysis* component, and that is able to specify a faulting condition and an optional level of severity to indicate that the condition has been violated; or that, if it has not been violated, there exists a possibility that it is, and that an action should be considered. On the reference side, the *Planning* component considers an interface with the *metrics* signature that is able to communicate with the *Monitoring* component and, through it, it is able to obtain the value of the *metrics* that the chosen strategy requires. Finally an interface supporting the *actions* signature should be able to send a set of actions in a predefined language that must be interpreted and executed by the *Executing* component.

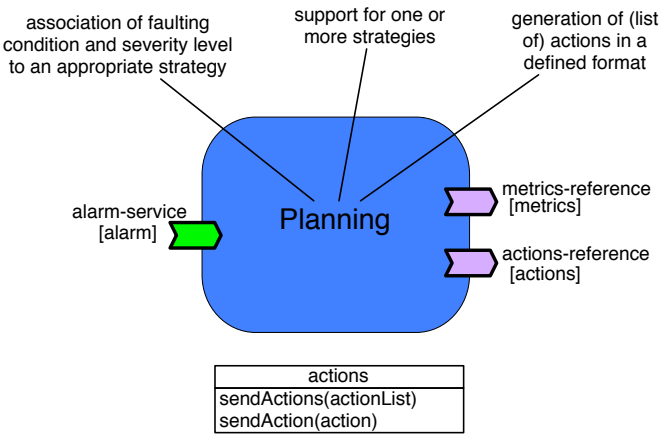


Figure 5.9: Basic SCA Planning Component

We have mentioned that a component-based approach inside the *Planning* should be able to better deal with different strategies by hosting them, f.e., as different components. It is also a concern of our framework, and one of the requisites mentioned in Chapter 4, that these strategies may be replaced at runtime. For example, an application may be driven by cost-saving strategy and, at some point the administrator may need to change the requirements and enforce an energy-saving strategy. In that case, a replacement of the corresponding strategy should be

triggered in the composition of the *Planning* component. However, this task is not an autonomic task of the framework itself and is, instead, driven by an administrator of the management layer. In Section 7.6 we describe a tool that we have designed to support the insertion and removal of strategies, assuming that the platform used for implementing the framework (in our case, GCM/ProActive) supports such kind of runtime modifications.

From the point of view of its input and output, the *Planning* component receives a notification about a faulting condition and, using the available monitoring information, it generates a sequence of actions that can take the service to a desired state.

Figure 5.10 shows an example, where service $A$ uses two services $B$ and $C_1$. The *Planning* component of $A$, $P_a$ receives an alarm from the *Analysis* component $A_a$ indicating the condition $\langle cost, <, 30\rangle$ has been violated, and that an action should be taken. $P_a$ executes a very simple strategy, which intends to replace the component with the higher cost. For obtaining the *cost* of both components $B$ and $C_1$, $P_a$ uses the *Monitoring* component $M_a$ which communicates with $M_b$ and $M_{c_1}$ to obtain the required values. As $C_1$ has the higher cost, the strategy determines that this component must be replaced. $P_a$ uses an embedded reference to a discovery service, to obtain an alternative service, in this case $C_2$ which provides the same functionality as $C_1$ (this is required to not interfere with the functional task of the application) and whose *cost* allows to satisfy the condition $\langle cost, <, 30\rangle$. With all this information $P_a$ is able to produce a single action $replace(C_1, C_2)$ as output.

It is worth to notice that all the logic of the strategy is encapsulated inside $P_a$, and that $M_a$ is only used to obtain the values of the metrics that the strategy may require.

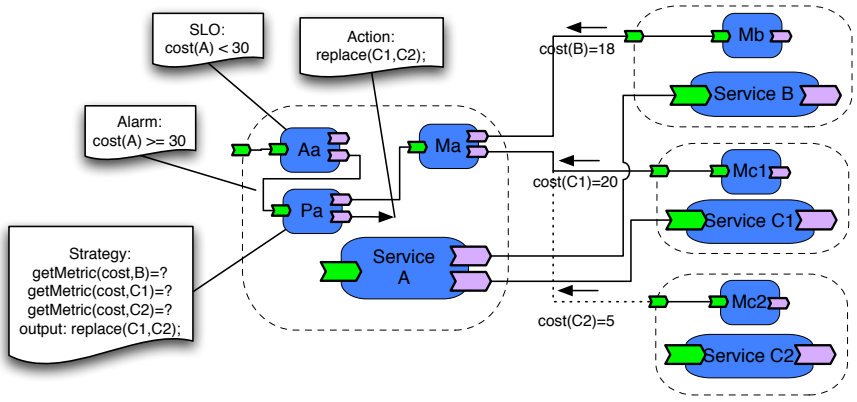


Figure 5.10: Example for the Planning component.

### 5.4.1 Interfaces

The *alarm* interface is used for receiving the faulting condition and its associated severity level. The *metrics* interface is used to access the monitoring information from the *Monitoring* component, that may be needed by the strategy prior to generate the list of actions.

The *actions* interface has the objective of sending an action or a set of actions to an entity (the *Execution* component) that is able to interpret them and execute them over the target service.

- `sendActions(actionList)`. Sends a sequence of actions as a list that must be executed on the target service.
- `sendAction(action)`. Sends a single action to be executed on the target service.

The format and definition of an *action* is a concern of the implementation. Actions may have different content depending on the strategy that generated them and, in that case it would be the responsibility of the *Execution* component to appropriately interpret them. A more general

approach can facilitate the interpreting task by incorporating part of it in the *Planning* component, in a manner that the *Planning* component generates an abstract set of actions, regardless of the strategy that created them. Then, the *Execution* component would only need to associate them to concrete actions on the actuators of the target service.

## 5.5 Execution

The objective of the *Execution* component is to carry out the sequence of actions that have been determined by the *Planning* component, on the target service.

Although it seems reasonable that once the actions have been decided, those be executed immediately, the *Execution* has more relevance than just executing actions. One of the reasons for having a different component is to separate the description of the actions from the specific way to execute them. In the same sense that the *Monitoring* component abstracts the way to retrieve information from the target service and provides a common interface to access the metrics it collects, the *Execution* component abstracts the communication with the target service to provide a uniform way to execute actions on the service.

Figure 5.11 shows the basic design of the *Execution* component. The service interface *actions* provides a way to send actions or a list of actions to the *Execution* component. Additionally the *Execution* component, in a similar way to *Monitoring* component (Figure 5.4) contains zero or more reference interfaces with the *actions* signature that allows to communicate with each one of the services that the target service references. The aim of these *actions* interfaces is to be able to delegate actions to other components and allow a decentralized transmission and execution of actions. Depending on the nature of the actions they can be sent in a synchronous or asynchronous way.

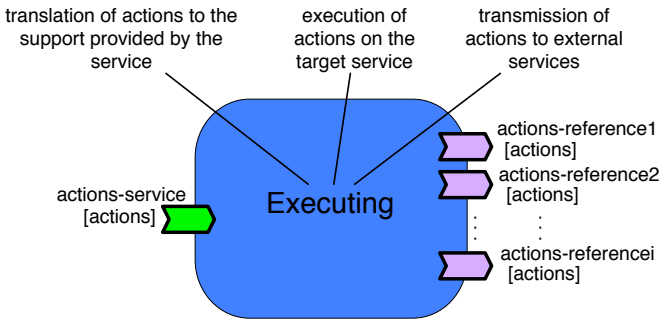


Figure 5.11: Basic SCA Executing Component

One of the challenges in the *Execution* phase is to ensure that the action will not make the application enter in an unsafe state. This problem is left to the execution implementation.

Figure 5.12 shows an example where three actions are generated by the *Planning* component: one to replace a service, one to unbind a service from another, and the third to set a parameter on a referenced service. In the example, the *Planning* component of $A$, $P_a$ has sent a list of actions to the *Execution* component of $A$, $E_a$. The action of replacing component $C_1$ by $C_2$ is executed locally at $A$. However, the unbinding of service of the reference $b_1$ on service $B$ must be executed by the $E_b$; and the setting of the parameter "threads" on service $C_2$ must be executed by $E_{c_2}$. By using the connections between the different *Execution* components, the actions can be delegated to the appropriate place.

Note that so far we have made no assumption about the order of the execution of the actions. Although the expected behaviour is that all the actions are executed sequentially, in some cases some actions may be triggered in parallel. However, the problem of determining which of those actions may be concurrently executed and in a safe way, it is not trivial at not addressed in this

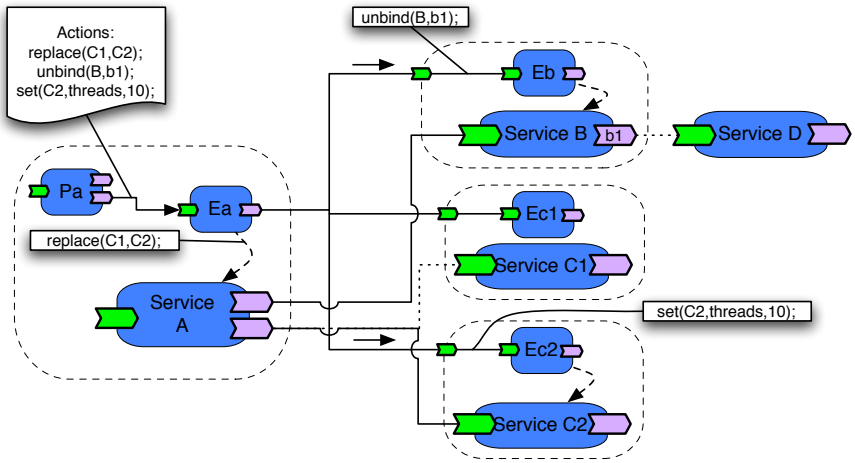


Figure 5.12: Example of propagation of actions through executors

work. In the implementation that we describe on Section 7.5, we assume that the mechanism allow a correct execution of actions according to the order in which they were generated.

## 5.6 Summary

In this chapter we have presented the design considerations taken into account for each one of the components that are expected to implement the different phases of the MAPE autonomic control loop, along with some examples of utilization. The interfaces to let the different phases to communicate have been presented as basic requirements, expecting that a concrete implementation can extend them as needed.

The description has been presented using the SCA notation with the intention that the presentation be general enough to be applied in any SCA compliant platform. Although we have tried to be as technologically-agnostic as possible, some details remain that must be unavoidably addressed by the supporting platform, f.e., the insertion and removal of metrics and strategies in the *Monitoring* component and in the *Planning* component, respectively; the choice of the appropriate way to describe the SLOs on the *Analysis* component; and the language to issue the actions to be executed by the *Executing* component. These considerations have been described in a general way as requirements over the implementation of the framework, and will be properly addressed in the next chapter.

In the next chapter we describe the implementation that we have carried on to show the usefulness and practicality of our approach on a particular technology. We give details on how the components are implemented taking into account the specific features of the hosting technology, and, at the same time, satisfying the presented design considerations.

# 6 Background and Technical Contributions

## Contents



This chapter presents some technical details about the platform over which we provide an implementation of our framework. Once presented the design considerations in Chapter 5, we need to introduce our concrete implementation, which requires to introduce some technical contributions in this platform.

Section 6.1 presents briefly the main concepts of the Fractal component. Section 6.2 describes the characteristics of GCM, whose design is heavily inspired by Fractal, and details which improvements it provides respect to the base Fractal implementation, and they support our solution. Section 6.3 describe the main features of the GCM/ProActive platform, the reference implementation of GCM, and over which we implement our framework. Finally Section 6.4 describes the technical choices and contributions we have made in order to properly support the design that we describe in Chapter 5 for this specific case.

## 6.1 Fractal

The Fractal[1] component model [BCL+06] is a general component model intended to implement, deploy, and manage complex software systems. Among its main features, Fractal includes:

- Hierarchical component model. Fractal components may be *primitive*, basic components; or *composite* components which are components that can contain other sub-components. This offers a uniform view of the application at various level of abstraction.

[1] http://fractal.ow2.org/

- Reflection. Fractal components have reflective capabilities and they are able to introspect their external and internal structure, giving them control over their own content, bindings and lifecycle.
- Reconfiguration. Applications based on Fractal components can be dynamically (re)configured by modifying their bindings dynamically.

A *Fractal Component* is a runtime entity that is encapsulated, has an identity, and supports one or more *interfaces*. An interface is the access point to the functionality of the component and may have one of two roles: *server interfaces*, which accepts incoming operation invocations; and *client interfaces*, which supports outgoing operation invocations. The main elements of the Fractal component model, as well as the usual notation, are indicated in Figure 6.1.

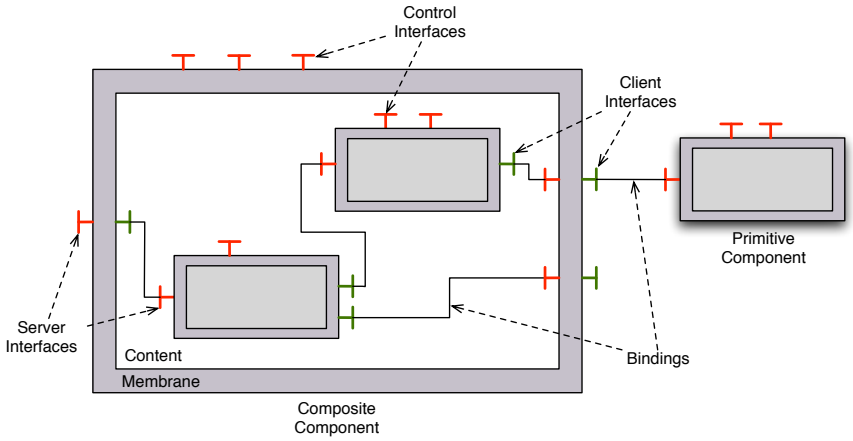


Figure 6.1: Elements of the Fractal component model.

Fractal components are basically composed of a *membrane*, that supports introspection and reconfiguration of the internal features of the component, and a *content* which includes a finite set of other components (called *sub-components*) in the case of a composite component, or an implementation logic in the case of a primitive component.

The membrane of a Fractal component comprises internal an external interfaces. Internal interfaces, which exist in the case of a composite component, are only accessible from the subcomponents of the composite. External interfaces, are accessible for components from outside. The membrane is composed of a set of *object controllers*, which allows to manage the content of the component by, f.e., controlling the lifecycle (start/stop) of the component, reconfigure the bindings to other components, add/remove components to/from a composite, modify attributes of the content. The Fractal model is defined as an extensible system, so that it is possible to create custom controllers according to the management requirements of the application. The management tasks from the controllers are available by special interfaces called *control interfaces*.

An important concept in Fractal is the *binding* mechanism, which allows to build the architecture of a Fractal application. Bindings allow to connect one client interface to one server interface, meaning that the operation invocations emitted by the client interface should be accepted by the server interface. In order to be bound, both the client and the server interfaces must belong to the same address space. This means that bindings cannot cross membranes. For example, in the case of a composite, an external component cannot bind directly to a server interface of a subcomponent. In the same way, a subcomponent cannot bind to the server interface of a component that is external to its composite.

By managing their bindings and composition, a Fractal application can be *structurally reconfigured*. This means that the structure of the application can be modified by changing the bindings of a component to other server interfaces, or by adding or removing components to/from a composite. The only requirement for such changes is that the component or components that

are going to be reconfigured must be in the stopped state, in order to guarantee the integrity of the composition.

The *type* of a Fractal component is determined by the set of interfaces it contains. Apart from their role, interfaces include the concepts of cardinality and contingency. *Cardinality* may be *singleton* indicating that the component has exactly one interface of type $I$, or *collection*, indicating that the component has an arbitrary number of interfaces of type $I$; such interfaces are usually created lazily, i.e., upon request of a bind operation. The *contingency* of an interface refers to the fact that the functionality corresponding to the interface maybe available or not at runtime; this way, a *mandatory* interface is guaranteed to be available at runtime, which means that, for a client interface, it must be bound before starting the component; an *optional* interface has no such guarantee, and the component can be started even if the interface is not bound.

Fractal has a reference implementation called Julia [BCL+06]. Julia is a Java implementation that supports the construction of applications based on Fractal components. Without entering in deep details about the Julia implementation, it is important to mention that Julia relies on a set of Java objects to implement the content of Fractal components, their interfaces and controllers. Additionally, optional *interceptor* objects can be added in the membrane to intercept functional invocations and provide some custom processing. In sum, the membrane of a Julia component is a set of Java objects that implement the management logic. Figure 6.2 shows a generic Fractal component, and its equivalent implementation view in the Julia framework.

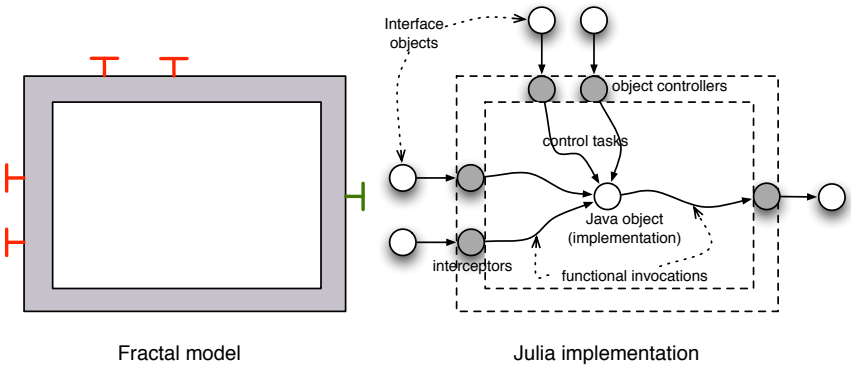


Figure 6.2: Model of a Fractal component versus its Julia implementation

Regarding the management features of Fractal components, the specification[2] exemplifies some useful *controllers*:

- *Attribute Controller*. Exposes getter and setter operations for certain configurable attributes of the component.
- *Binding Controller*. Exposes a `BindingController` interface that allows to bind and unbind the client interfaces of the component to external server interfaces.
- *Content Controller*. Exposes a `ContentController` interface that allows to list, add, and remove sub-components from the content, in the case of a composite.
- *Life-cycle Controller*. Exposes a `LifeCycleController` interface that allows to explicitly control the running state of the component.

Fractal has served as the basis for developing SCA compliant runtime platforms as FraSCAti, which benefits from the reflective Fractal features, and provides reconfiguration capabilities to architectures based on the SCA specification.

[2] `http://fractal.ow2.org/specification/`

Fractal applications can be described using the Fractal ADL[3], an XML-based descriptor language, that allow to describe all the elements (components, bindings, controllers, implementations, interfaces) of a Fractal application and their assembly.

## 6.2 GCM

The Grid Component Model (GCM) [BCD+09] is a component model targetted at the design, implementation and execution of grid-aware component-based applications. The GCM model has been specified by the Institute on Programming Model of the European CoreGRID project, and standardized by the European Telecommunications Standards Institute (ETSI).

The specification of GCM is based on the Fractal Component Model. Consequently, GCM components share many characteristics with Fractal components in terms of hierarchy, composition, interfaces, bindings and reconfiguration capabilities. However, GCM enforces some considerations that are not part of the main Fractal specification:

- Support for distributed deployment.
- Better support for collective communications.
- Strict separation between functional and non-functional concerns.

The architecture of a GCM application can be described using the GCM ADL, which extends the Fractal ADL to include the elements defined by GCM. In the following we describe the main features of the GCM model that we intend to exploit in the development of our solution [OAS07]:

### 6.2.1 Support for distributed deployment

One of the differentiating features of GCM is its generic support for distributed deployment, which is based in the *Virtual Node* (VN) abstraction. A Virtual Node is an abstract reference to the resource where the component will be deployed. This abstraction captures the deployment requirements and allows to separate the tasks of designing the architecture of the GCM application, from the provisioning of the resources where the components will be deployed.

Virtual Nodes possess a *cardinality* property that can be defined as *single* or *multiple*. In the first case, the VN must be mapped to exactly one node on the physical infrastructure, while in the second case it can be mapped to (i.e. encompass) several nodes.

The information about VNs is included in the GCM ADL and it describes a *virtual infrastructure*. At deployment time, the VNs must be associated to a concrete *physical infrastructure*. The provisioning of the physical resources can be delegated to a specialized tool. An example of such a tool is the ProActive Resource Manager, which is able to manage nodes from several source infrastructures, and provide them to the application in a uniform way.

In a situation where the infrastructure support for service-based applications relies more and more in highly available computational resources, offered itself "as a Service", a generic means to describe the infrastructure applies well to this vision, as it allows to abstract the design of the application from the actual infrastructure.

### 6.2.2 Support for collective communications

The support for collective communications is improved in the GCM model by the introduction of *gathercast* and *multicast* interfaces, with the aim of providing efficient many-to-one and one-to-many communications. These interfaces allow to manage a group of interfaces as a single entity, exposing the collective nature of GCM Components. The notation for both interfaces is shown in Figure 6.3.

[3] http://fractal.ow2.org/fractaladl/

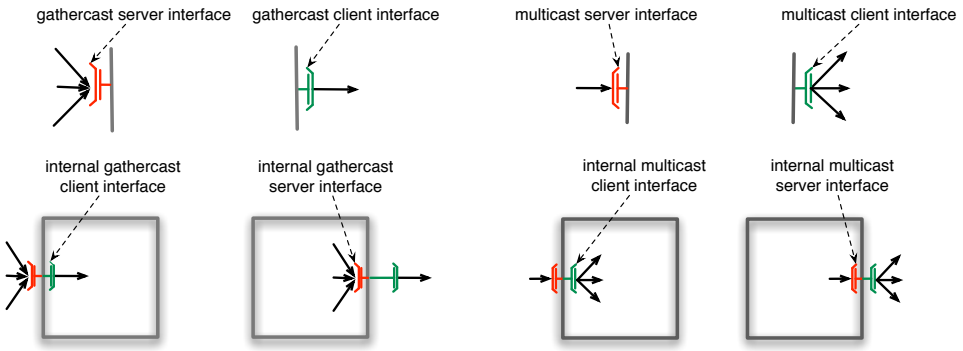


Figure 6.3: Gathercast and Multicast Interfaces in GCM

#### 6.2.2.1 Gathercast Interfaces

Gathercast interfaces allow many-to-one ($M \times 1$) communications, and can be used to perform synchronization, parameter gathering, reduction, and result dispatch. The specific behaviour for these interfaces can be specified using different *aggregation policies*.

Gathercast interfaces receive invocations from multiple bound interfaces, gathering all the parameters, and reducing them to a single invocation. When a result is obtained, the gathercast interface performs the corresponding dispatch to all the invoking partners.

#### 6.2.2.2 Multicast Interfaces

Multicast interfaces allow one-to-many ($1 \times M$) communications, and can be used to perform parallel invocations, parameter dispatch and result gathering. The specific behaviour for these interfaces can be configured using *dispatch modes*. Dispatch modes may be used to define the way the parameters of the original invocation will be divided between the different destinations, or for choosing on which one of the connected interfaces the invocation will happen.

Multicast interfaces transform a single invocation into possible many parallel invocations. When a result is obtained, the possible multiple results can be aggregated and returned to the original invoker.

The existence of gather and multicast interfaces is initially developed for functional interfaces, and allows that a component be bound to an undetermined number of interfaces. In our approach, we expect to be able to communicate with the membranes of all the components involved in an invocation, and the multicast interface constitutes and appropriate means for accessing multiple membranes according to the architecture of the GCM application.

### 6.2.3 Support for Non-Functional concerns

The Fractal specification, and its Julia reference implementation, describe management elements as a set of *object controllers* contained in the membrane of a Fractal component. These object controllers are defined in a static way and they are described using the Fractal ADL.

GCM components extend this notion to allow the introduction of Non-Functional (NF) Components [DFG+08]. NF Components reside in the membrane of GCM Components, and have a similar goal to object controllers: to provide management features to GCM components, and ultimately take charge of Non-Functional tasks.

#### 6.2.3.1 Separation between Functional (F) and Non-Functional (NF) concerns

The motivation for providing a “componentized” membrane to GCM Components, is to allow the composition of more complex activities in the control part, and to provide a more clear separation

of concerns between the functional and the non-functional tasks. For this matter, the GCM ADL can be split and the componentized architecture of the membrane can be described in a separate file from the functional part. This allows to develop both parts in a rather independent way and to associate them only at deployment time, enforcing the separation of F and NF concerns.

A point must me made to explain what we mean by Non-Functional tasks. Non-Functional tasks refer to all those tasks that are not related to the main goal (or functional goal) of a GCM Component. Although a proper definition depends on what is ultimately the functional goal of a GCM Component, or "what is the component expected to do", Non-Functional tasks can be defined as the tasks that support the functional goal of the component, or that allows the component to do what it needs to do. Non-Functional tasks usually include the management of the structure of the GCM application (bindings and compositions), management of the lifecycle of the component, and in general, the supervision of the execution of the functional goal.

#### 6.2.3.2 NF Interfaces

In many cases, management activities may require a meaningful collaboration of several tasks, that can be spread transversally to other components. For example, to get a measure of the energy consumption of the application, it is necessary to aggregate the energy consumption of all the individual components.

GCM includes additional control interfaces, referred to as *NF interfaces*. Where Fractal components provides only *NF server interfaces* to allow access to NF tasks, GCM includes *NF client interfaces*. NF client interfaces can be bound to *NF server interfaces* of other components, in order to communicate with the membranes of different components and allow collaborative NF tasks. We will use the possibility of having a component-oriented view of the membrane of GCM Components, to provide a flexible and collaborative implementation of the MAPE autonomic control loop, as we describe it in Chapter 7.

With respect to internal interfaces, GCM allows the membrane to communicate with the content in order to allow a hierarchical communication and collaborative NF tasks between a composite and its subcomponents. For this case, GCM defines additional *internal NF interfaces*. Internal NF interfaces allow components residing in the membrane of a GCM Component to communicate with components residing in the functional content, and conversely, it allows subcomponents to communicate with the components residing in the membrane.

#### 6.2.3.3 Notation and NF Bindings

Figure 6.4 shows an example of a GCM application, where NF Components have been introduced in the membrane of a composite and a primitive component. Note that GCM allows the co-existence of both object controllers and NF Components (sometimes also called *component controller*) in the membrane. However, object controllers cannot be modified at runtime, and cannot be bound to, or communicate with other NF interfaces as NF Components do.

The description in Figure 6.4 also illustrates the possible bindings using the newly introduced interfaces. A client interface of a NF Component can connect to an *internal NF client interface* that pertains to the membrane (1). This *internal NF client interface* acts as a passage from the membrane to the content. An *internal NF client interface* can bind to the *external NF interface* of a sub-component (2), concretizing the communication from the membrane to the content. This allows to propagate management tasks originated in the membrane, to the membranes of subcomponents. A very simple example can be a stop signal sent from the membrane of a composite that can be propagated to the membranes of its internal subcomponents.

Conversely, the communication can also flow from the content to the membrane. An *external NF client interface* of a subcomponent can connect to an *internal NF server interface* of its parent component (3). This *internal NF server interface* acts as a passage from the content to the membrane. The *internal NF server interface* can connect to the server interface of a NF Component (4), concretizing the access to the management task. This allows the membranes of sub-components to access to management tasks on the membrane of its parent component.

For example, a component that represents a storage resource can use a NF client interface to communicate to its parent that the storage capacity is near full, so that the membrane of the composite can take some action.

The other elements shown in Figure 6.4 indicate the bindings between the external NF server interface of a component with the server interface of a NF Component in the membrane (5); the bindings between the client interface of a NF Component to an external NF client interface (6) allow communication with the exterior. Finally, an external NF client interface can be bound to an external NF server interface of another external component (7), effectively communicating the membrane of two components and allowing to implement collaborative NF tasks. For example, a security manager residing in the membrane can propagate a security certificate to all the components with which it maintains a functional binding in order to authenticate itself.

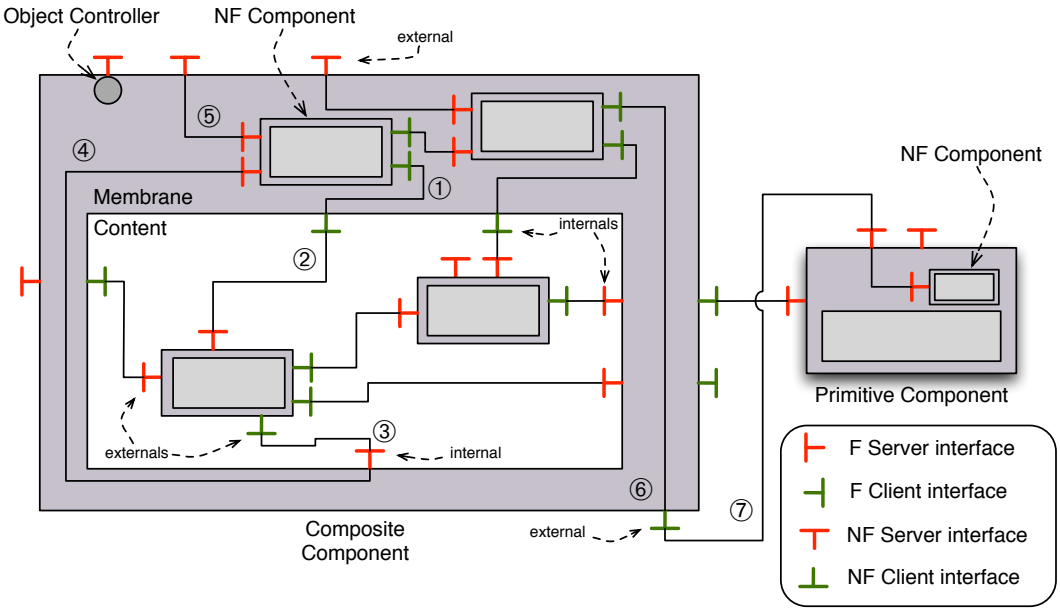


Figure 6.4: Elements of a GCM application including NF Components in the membrane

#### 6.2.3.4 Standard Controllers

GCM provides interfaces for the basic controllers mentioned in the Fractal description (Section 6.1), and define some additional standard controllers:

- *Gathercast Controller*. Allows the management of gathercast interfaces, i.e., the creation of gathercast interfaces and the management of their connections.
- *Multicast Controller*. Allows the management of multicast interfaces, i.e., the creation of multicast interfaces and the management of their connections.
- *Membrane Controller*. The Membrane Controller manages the composition and bindings of NF Components residing in the membrane, as well as the addition or removal of NF Components to/from the membrane. It also manages the bindings to/from internal NF interfaces, allowing the connection of the NF interfaces of sub-components with the membrane activity.

#### 6.2.3.5 Reconfiguration

Basic reconfiguration capabilities in GCM, inherited from Fractal, allow to modify the bindings between GCM Components, and add/remove subcomponents to/from a composite. Also, like

Fractal, the only requirement for executing these structural reconfigurations is that the components be in a *stopped* state (which does not mean undeployed), in order to preserve the integrity of the composition.

Nevertheless, GCM also allows to structurally reconfigure the membrane [BHN09]. In fact, the set of NF Components in the membrane can be seen just like another GCM application and their bindings and composition relationships can be modified at runtime if needed. Regarding the lifecycle of the application, an intermediate state is added in which the GCM Components can be in a *started* state, but their membrane can be in a *started* or *stopped* state. When executing reconfigurations over the membrane composition it is required that the membrane be *stopped*, which means that all the NF components inside the membrane of a single GCM component must be stopped.

## 6.3 GCM/ProActive

The reference implementation of GCM has been developed over the ProActive middleware and is referred to as GCM/ProActive. GCM/ProActive features an implementation of components based on an *active object* model that supports asynchronous communications based on transparent Future objects.

The basic important notion is that in GCM/ProActive all components (primitive and composite) are implemented by *Active Objects*. This implies that components have a single thread of execution, and a queue where requests are stored to be executed in an asynchronous way.

In the following we describe some features of the GCM/ProActive implementation, that we need to consider in the implementation of our framework. We describe, the active object model and its asynchronous communication, the current instrumentation capabilities, and message tagging feature.

### 6.3.1 Active Objects and Asynchronous Communication

An *Active Object* is built using an extensible *meta-object protocol* (MOP) architecture, which uses reflective techniques to abstract the distribution layer and offer asynchronism. The MOP architecture is shown in Figure 6.5.

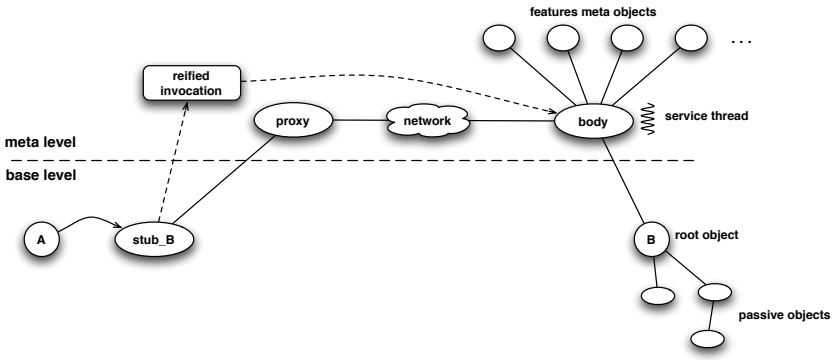


Figure 6.5: Meta-object architecture

An Active Object is concretely built from a “root object” (of type $B$ in Figure 6.5) which is a common regular (passive) object. An object called *body* is attached to the root object, and this *body* references several features meta-objects with specific roles (fault tolerance, security, group communication, etc). Additionally, an object called *stub* is created as sub-type of the root object $B$, so that from the point of view of the invoker, the destination looks like a common object of type $B$.

Asynchrony is provided as follows. An invocation on the *Active Object B* is actually an invocation on the *stub* object, which creates a reified representation of the invocation including the method called and its parameters. This "reified invocation" is given to a *proxy* object that transfers it to the destination *body*, possibly through a network, and places the reified invocation in the *request queue* of the Active Object. The *request queue* is one of the meta-objects referenced by the *body*. Once the reified invocation (the request) is placed in the *request queue*, the *rendezvous* time finishes. If the method invoked is expected to return a result, then a *Future* object is created on the caller side and returned back as a result to the caller. The *Future* is a placeholder for the return value of the invocation and comprises a *Future stub* that is a sub-type of the return type of the invocation, and a *Future proxy* that references the actual returned value. The object that has issued the invocation can now continue executing in a regular way. However, if at some point it requires to read the actual returned value, it will block until the value is available, or it will continue transparently if the value has already been obtained. This feature, that is transparent for the application, is called *wait-by-necessity*.

After the request has been placed in the *request queue* of the active object, it can be served asynchronously at any time according to the serving policy of the active object (by default, a FIFO policy). Once the active object has served the request, it contacts transparently the object that sent the request (and which holds a *Future* object for the response) and updates the result, unblocking any thread that may have been doing *wait-by-necessity* on this result. However, the caller may also have sent copies of this *Future* object as a response to other objects, or as a parameter to subsequent requests. These copies must be also transparently updated, and the mechanism used is called *Automatic Continuation*.

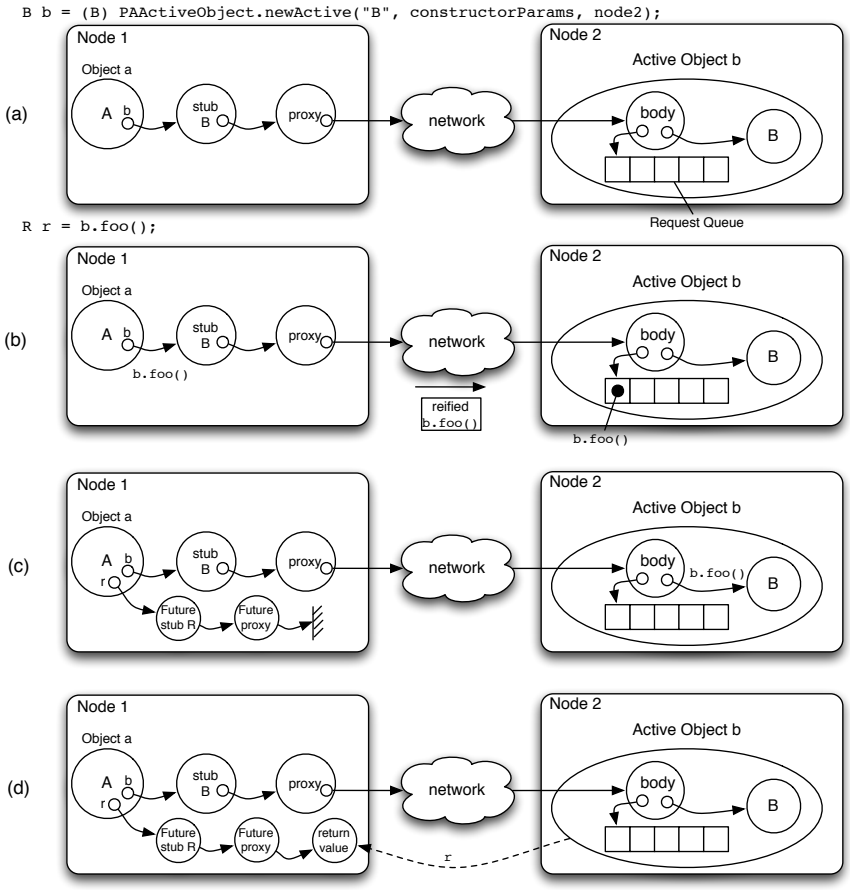


Figure 6.6: Sequence of an asynchronous call to an Active Object

```
A {
   ...
   // instantiate active object of class B on node2
   B b = (B) PAActiveObject.newActive(''B'', constructorParams, node2);

   // use active object as a regular object of type B
   R r = b.foo();
   ...
   // possible wait-by-necessity
   System.out.println(r.printResult());

}
```

Listing 6.1: Creation of an Active Object in ProActive

Figure 6.6 shows the status of the objects after each step listed in Listing 6.1. An object $a$ of type $A$ creates an active object $b$ of type $B$ remotely located in "Node 2", using the ProActive API. This causes the generation of a *stub* of type $B$ on the context of $A$, which represents the remote active object $b$, and a *proxy* in charge of forwarding requests to the actual location of the active object. At the same time, in "Node 2", the active object $b$ is created with a *body*, a *request queue*, and the root object of type $B$ (Fig. 6.6(a)). When a call is made on the (active) object $b$ (Fig. 6.6(b)), the call is made on the *stub* of type $B$. The invocation is reified by the *proxy* and a *rendez-vous* is made in which the *proxy* sends the reified request to the *body* of the active object, which puts it in its *request queue*. After this step, the *rendez-vous* time finishes, and the object $a$ creates a local *Future* object, initially empty, that will receive the result of the invocation, $r$ (Fig. 6.6(c)). From this moment, both object $a$ and $b$ can continue their task in parallel. If $a$ needs at some moment to access or use the object $r$, it will use the corresponding *FutureStub*, but if the value has not yet been computed by $b$, then the thread of $a$ will block at the *FutureProxy* until the result is available (*wait-by-necessity*). On the other side, $b$ decides asynchronously to serve the request and compute the result $r$. When $b$ has finished serving the request, the *body* sends the value $r$ to the caller ($a$) which handles it to the corresponding *FutureProxy* that updates the value (Fig. 6.6(d)). At this moment any thread that was blocked waiting for this result is released.

### 6.3.2 Asynchronous Communications in GCM/ProActive components

GCM/ProActive provides an implementation of GCM on top of the ProActive middleware. Consequently, each GCM/ProActive component is implemented by an Active Object that comprises a single thread of execution, a body, an a queue of requests. The implementation is based on the MOP architecture in the way presented in Figure 6.7. The main difference is that the role of the *stub* in the basic MOP architecture is taken by two interfaces:

- The *Component* interface exposes the GCM Component nature of the active object, and allows to introspect details of the component.
- One *Interface* interface is dynamically generated for each server interface declared by the component, according to the functional methods included in the interface.

These elements act as local representatives of a remote GCM/ProActive component, and use the *proxy* to allow the interaction with the *body* of the active object. The *body* includes additional *component meta-objects* that include the *controllers* mentioned in Section 6.1 and 6.2.

An invocation from a GCM component to another takes a path like the one shown in Figure 6.8. GCM Component $A$ is bound to GCM Component $B$ from the client interface `aItf` of $A$ to the server interface `bItf` of $B$ (this requires that the interfaces be compatible). The binding operation is realized using the `BindingController` of $A$, and the consequence is that the active object $A$ holds a reference to the *Interface* and *Component* representatives of component $B$ (Fig.

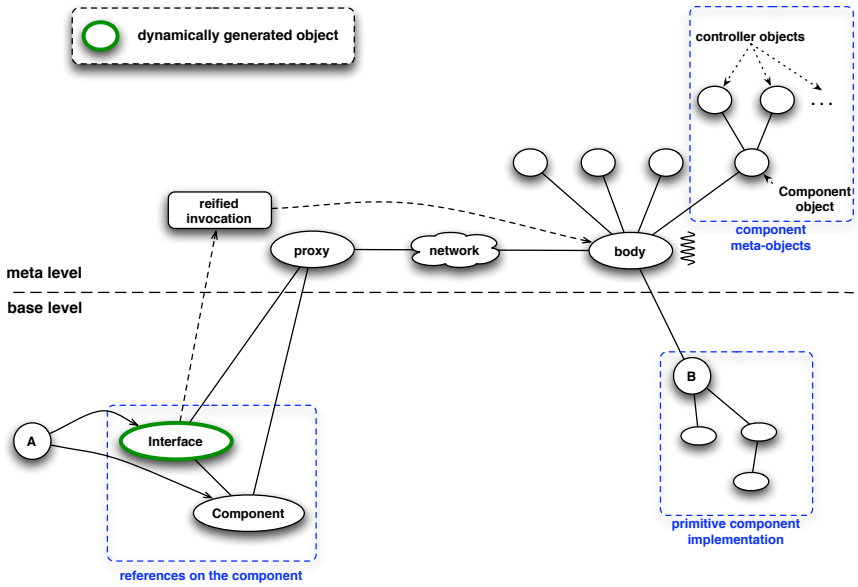


Figure 6.7: Meta-object architecture

6.8(a)). The representatives are used to make an invocation on the client interface `aItf` of component *A*. The call is reified and transmitted as an invocation on the server interface `bItf` of component *B*, which is stored in the request queue of active object *B* (Fig. 6.8(a)). Once the request has been stored in the request queue of *B*, component *A* holds a *Future* object for receiving the result of invocation. Component *B* takes the request from its queue, and when it has finished serving it, sends the result back and updates the *Future* object in component *A* (Fig. 6.8(b)).

Composite components in GCM/ProActive does not contain any implementation logic. Instead they act as containers for subcomponents. Composite components contains passive objects for their associated interfaces and controllers, and a single Active Object that handles the invocations received on the server interfaces of the composite and delegates the call to the corresponding bound subcomponent, acting like a proxy for any external entity that wants to communicate with a subcomponent. In the same way, the composite Active Object handle all the outgoing invocations from a subcomponent through a client interface to an external entity. In order to preserve encapsulation, a subcomponent can make invocations on the internal server interface of its parent component, which forwards the request to the external component bound to the corresponding client interface of the composite, acting like a proxy for outgoing invocations. A simplified view of the invocation flow through GCM/ProActive components is shown in Figure 6.9. The GCM application is the same of Figure 6.4, however NF Components are not shown for clearness.

Bindings are implemented as references from the Java objects that implement the logic of a primitive component, to the object that represents the server interface of another GCM component.

### 6.3.3 Basic instrumentation in GCM/ProActive

The GCM/ProActive middleware has been instrumented using the Java Management Extensions (JMX) technology, in order to provide notifications about events occurring in the middleware. GCM/ProActive provides a *ProActive JMX Connector* that enables to connect to and propagate JMX notifications asynchronously using ProActive communication semantics.

The bodies of active objects are instrumented with MBeans (*BodyWrapperMBean*) that are

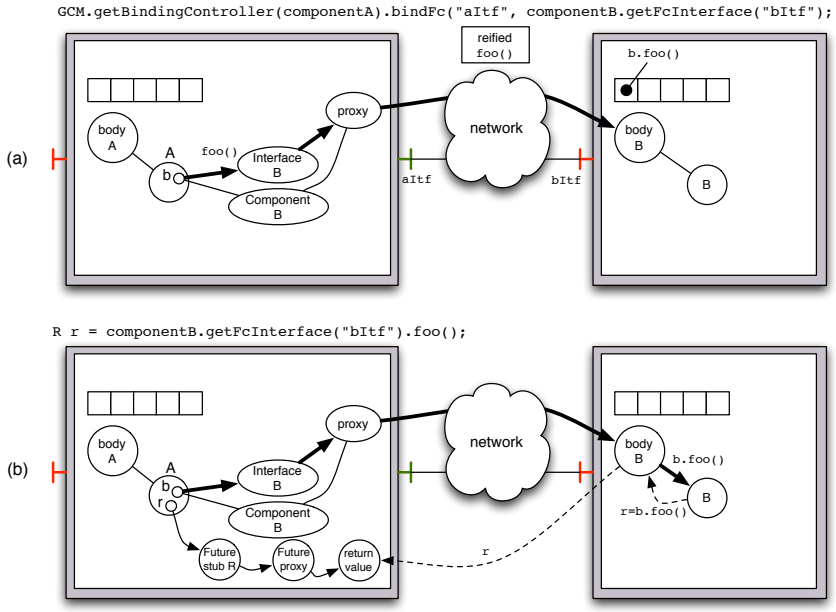


Figure 6.8: Sequence of an asynchronous call in GCM/ProActive components

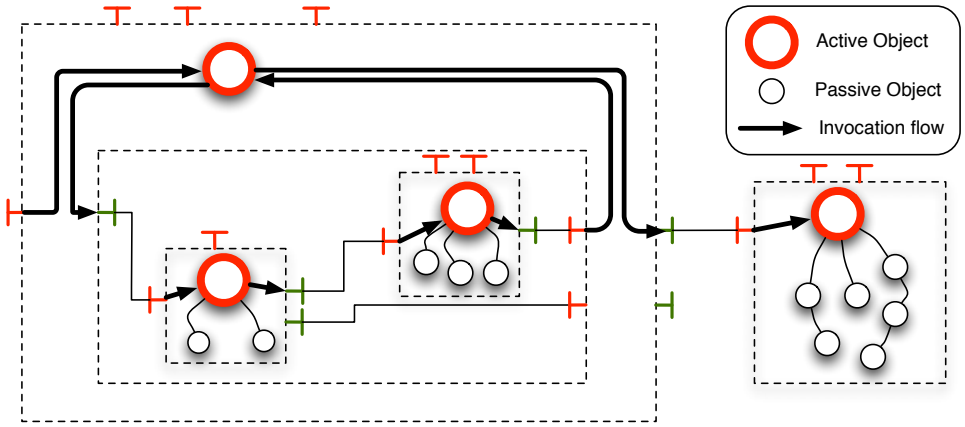


Figure 6.9: Invocation flow in GCM/ProActive

used to generate JMX Notifications. Table 6.1 shows the notifications generated by the ProActive middleware related to the service of a request and the management of Future objects.

| Notification name | Event |
|---|---|
| requestSent | Component sends a request |
| requestReceived | Component receives a request |
| servingStarted | Component starts serving a request from the queue |
| replySent | Component sends a reply |
| replyReceived | Component receives a reply |
| voidRequestServed | Component finishes serving a request that does not return a value |
| waitForRequest | Component is waiting for new requests |
| receivedFutureResult | Component receives an update for a Future value |
| waitByNecessity | Request has been blocked due waiting for a Future update |

Table 6.1: Notifications generated by GCM/ProActive

This set of notifications has been used in applications related to the ProActive middleware, like IC2D, a graphical tool that allows to monitor and benchmark Java ProActive applications. IC2D listens to JMX notifications generated by the *BodyWrapperMBean* objects of ProActive applications to graphically represent communication among active objects, and obtain performance information about the time spent in the internal tasks and communication of a ProActive-based application.

However, the current implementation does not allow to correlate the time spent serving different requests that have a causal relation, making harder to isolate sources of poor performance. Also, the *replySent* notification is sent both at the end of the *rendez-vous* time, and upon any update of a Future object, making harder to compute the effective time spent by an active object serving a specific request, as it does not consider the fact that the reply to a request may also include a Future value which must be later updated again.

Information about deployment details can be obtained also through *MBeans*. ProActive runtimes and nodes are instrumented with a *ProActiveRuntimeWrapperMBean* and a *NodeMBean*. The first one allows to collect information about the virtual machine that hosts the ProActive application, like the amount of heap memory, number of threads, number of bodies, and CPU load. The second one exposes information about the concrete node that hosts the application like the list of active objects and the name of the assigned Virtual Node.

### 6.3.4 Message Tagging in GCM/ProActive

ProActive provides a message tagging API that allows to attach custom information to messages sent between active objects, called *tag*s. The *tag*s added to a request by one active object can be retrieved and read by another active object. The API allows to identify individually different *tag*s, so any number of *tag*s can be attached without interfering, other than the additional amount of information sent. At the same time, the API allows to define a task that can be applied to a *tag* before its propagation according to the specific treatment that application needs.

A very simple example of this feature is to implement a counter, as a *tag* whose task is to increment a value by one before each propagation. This *tag*, when read at the each destination active object indicates the depth of the invocation.

Another example application of message tagging is to identify distributed flows in the invocation of a request. A distributed flow allows to find all the communications that are causally related in a distributed application. In this case the task associated to the *tag* would be to replicate an identification value through every invocation. The requests that share the identifier belong to the same distributed flow.

### 6.3.5 Additional Features

#### 6.3.5.1 Legacy Code Wrapping

GCM/ProActive components can be used to wrap existing legacy code, allowing to manage a non-componentized application in a component-oriented way. This wrapping method has been developed with the aim of automatically deploying and executing native applications like C/C++/Fortran MPI code, on Grid environments in a rather transparent way.

For wrapping a legacy code as those mentioned, some small pieces of the MPI code must be modified, however, to use the ProActive communication API that allow the wrapping. Once an applications is wrapped inside GCM/ProActive component, this can be deployed and managed like any other GCM/ProActive application. By using a C/JNI library that includes simple management operations (`init` and `terminate`) communication primitives (synchronous/asynchronous `send` and `receive`), it is possible to communicate native processes with Java active objects from GCM/ProActive.

#### 6.3.5.2 GCM/SCA

The membrane of GCM components allows to adapt the NF characteristics of the components. One example has been developed in the adaptation of GCM as an SCA compliant runtime platform. This platform, called SCA/GCM comprises two additional controllers. The SCAIntentController and the SCAPropertyController. The SCAIntentController allows to attach *intent handlers* to a given server interface, with the objective that all incoming invocations are intercepted by the *intent handler* before its normal processing. This allow to associate objects implementing *SCA Intents* to SCA/GCM components. The SCAPropertyController allows to access and modify properties on SCA/GCM components. Properties are fields of the GCM Components that have been previously identified as such using SCA annotations.

## 6.4 Technical Contributions

Once presented the technological background that supports our solution we justify the technical choices we have made when using GCM/ProActive as support for our framework, in the light of the requirements for our solution, expressed in Section 4.1.2 and 4.1.3. Then, we briefly describe the technical contributions that we have provided in order to implement our framework in this concrete technology, these being further described in the following chapters.

### 6.4.1 Technical Choices

All along the description of our solution in Section 4.1.3, we have aimed to solve the requirements that we mentioned in 4.1.2 for obtaining a generic and flexible solution for monitoring and management of service-based applications.

We have described the GCM model and the relevant features of its reference implementation GCM/ProActive for our solution. As we have chosen the GCM/ProActive technology to provide an implementation of our solution, we describe how this technology helps us to address the requirements that we have mentioned.

- Extensibility. By defining a set of component interfaces for the functionality provided by each phase of the MAPE loop, we expect to provide a skeleton where different implementations of each phase can be plugged.
- Flexibility. The runtime reconfiguration features available in GCM allows to give an appropriate support for inserting and removing elements from the framework, ultimately obtaining reconfiguration at the level of the monitoring and management framework itself.
- Heterogeneity. Although it is supported by a specific technology, the componentized and dynamic nature of GCM/ProActive allows to plug adaptors at the monitor and at execution phases in order to receive information from various sources, and send actions to different execution supports, by developing adaptors for each situation. Moreover, by using the VN abstraction, it is possible to abstract several infrastructure destinations.
- Efficiency. By attaching NF Components to the membrane of GCM Components, the MAPE loop can be specialized for each component, avoiding transmitting unnecessary information to f.e. a centralized location, and allows to take decisions and execute actions at a more fine grained level. At the same time, by using NF interfaces, it is possible to establish a collaboration between the membranes of different components when needed.
- Autonomicity. By implementing different phases of the autonomic control loop inside the membrane of GCM Components, it is possible to provide an autonomic behaviour to services that can be highly personalized. The framework defined this way allows a more easy insertion of autonomic capabilities, while preserving the abstraction, encapsulation, and separation of concerns between the F and NF activities when developing the application.

These features offered by GCM/ProActive allows to address the requirements we have pointed out in Section 4.1.2 showing that it is an appropriate choice for implementing our solution. However, some aspects are not completely covered in the current version, so we have provided some technical contributions in order to properly support our implementation.

### 6.4.2 Technical Contributions

Once presented the technologies that we have chosen to support the implementation of our framework, we detail the technical contributions that we have provided.

- Extension of collective behaviour to NF interfaces. In order to keep the separation of concerns between F and NF activities, and allowing a more transparent collaboration between membranes, the GCM collective multicast interfaces described in 6.2.2 are extended to provide the same capabilities to NF interfaces.
- Definition of interfaces for MAPE-like behaviour. In order to provide flexible implementations of the MAPE autonomic control loop, we have defined a basic set of interfaces that allow the different phases to interact in an independent way of the specific implementation. These interfaces must be provided by any custom GCM component to be introduced in the framework.
- Set of NF Components for building autonomic control loops. As a consequence of separating the phases of the MAPE autonomic control loop, we have developed a set of basic and flexible NF Components that can be inserted and removed to/from GCM components, and that can be used to compose efficient autonomic control loops, or to attach custom monitoring and management tasks.
- Definition of an API for managing the monitoring and management framework. We have provided and implemented an API that allows to introduce the NF Components that implement the MAPE phases in the membrane and compose them in a flexible way to provide the monitoring and management behaviour required. This API allows to use components from the set of predefined NF Components that we have provided, or to use custom NF Components that comply with the interfaces. At the same time we have provided a console for managing the framework using this API.
- Instrumentation of the GCM/ProActive implementation. In order to get a detailed information about the requests completion and performance of GCM/ProActive requests, even considering that these requests may come from wrapped applications, we have introduced events and improved the meta-information propagated by the requests considering the asynchronous communication protocol and the Future update mechanism. These additions allow to get a real decomposition of the time spent serving requests. We have also provided sensors that allow to correlate the GCM/ProActive components with the concrete nodes where they have been deployed.

## 6.5 Summary

We have presented our solution in the light of the requirements that we have identified from the context of service-based applications, and the state-of-the-art. Once describe our scientific contribution, we have presented the technological background we use to support our solution and that will be used to provide a concrete implementation, namely the GCM model and its reference implementation GCM/ProActive. We describe how the features found in GCM/ProActive allow us to support the requirements for our thesis.

In the next chapter we present the design of our solution from a technologically independent point of view, and then we describe the details of our concrete implementation in the GCM/ProActive middleware, using GCM/SCA as a suitable composition framework for SCA-based applications that can be turned autonomic.

# 7 Implementation

## Contents



Chapter 5 presented the design of our framework, including the general guidelines and consideration that should be taken into account in an implementation. Chapter 6 gave the technical about the GCM/ProActive middleware which we use a support for a particular implementation of our framework that we describe in this chapter to demonstrate the feasibility and practicality of our solution.

Section 7.1 gives a general view of the implementation of our framework and the technical contributions that we have provided. Sections 7.2 to 7.5 detail the implementation we have provided for the different phases. Finally, Section 7.7 summarizes this chapter.

## 7.1 Framework Implementation

As mentioned in Section 6.2, the GCM model allows the dynamic insertion of Non-Functional (NF) Components in the membrane of a regular GCM component, that can be bound to NF Interfaces, and whose objective is to take care of NF concerns of the component.

We provide an implementation of our framework as a set of NF Components inserted in the membrane of a regular (functional) GCM component. Figure 7.1 shows the general structure of a GCM component implementing a service called *Service A*, where a set of NF Components have been inserted in its membrane. Additionally, the NF Components are interconnected following the design of Figure 5.2.

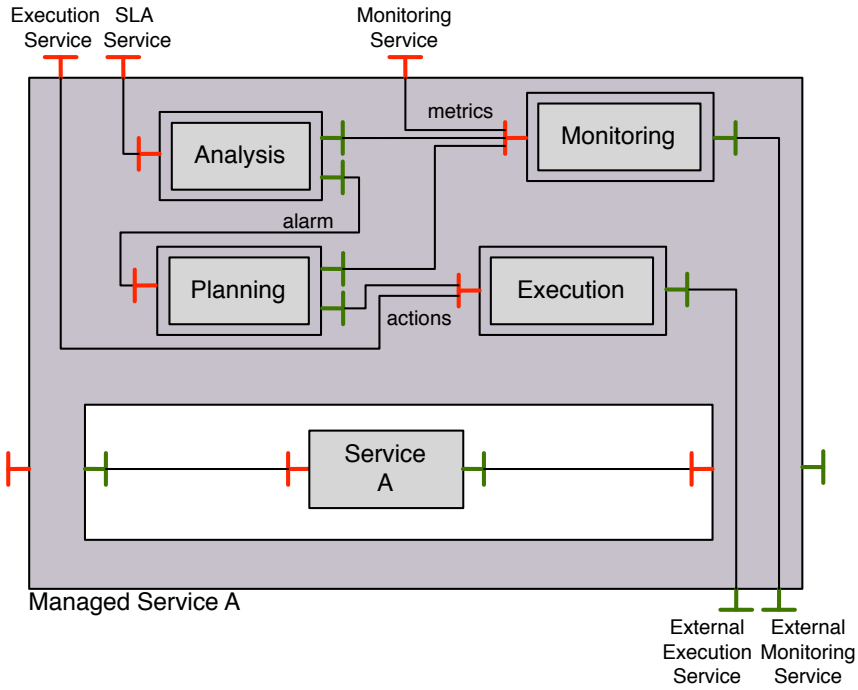


Figure 7.1: Framework implementation inserted into the membrane of a GCM/ProActive component implement *Service A*

The example shows the four MAPE NF Components inserted in the membrane of *Service A*: *Monitoring*, *Analysis*, *Planning*, and *Execution*. In addition to the existent functional interfaces of *Service A*, and the common NF interfaces that a GCM component usually includes (such as *Lifecycle*, *Binding* and *Name*), the *Managed Service A* component includes three NF Service interfaces that allow to access the monitoring and management capabilities of the extended service: *Monitoring Service*, *SLA Service*, and *Execution Service*. As long as the original *Service A* has F client interfaces, the extended *Service A* includes also NF Client interfaces to interact with the *Execution* and *Monitoring* interfaces of other components.

The implementation of the framework includes:

- An implementation of the four MAPE components that complies with the design described in Chapter 7.
- A library of elements that can be plugged and unplugged to the provided elements in the framework, to modify its behaviour.
- An external console application that can manage the insertion and removal of the MAPE NF Components in the membrane of a GCM application.

The intended use of the framework is that the programmer, instead of creating a GCM Component using the standard NF Type, will create the component with an extended NF Type that will add the required NF interfaces to the component. The addition of these interfaces does not oblige the component to have MAPE NF Components in its membrane, but it is only a technical need to have the appropriate points of entrance to interact with those components in case

they are present. The initial component composition of the membrane can be described at deployment time using the NF ADL section of the GCM ADL, in a way that the needed MAPE NF Components will be automatically deployed. However, this composition can be modified at runtime using the console application provided. In fact, the console is used not only to insert or remove elements from the membrane, but also to bind them in an appropriate way and to add or remove elements to customize its behaviour.

Once the MAPE NF Components are inserted in the membrane, and configured using the console application, they can be started. In a similar way as described in Section 5.1, the *Monitoring* components use internal event listeners and sensors to collect and store information about the GCM/ProActive application. Using the information collected, the *Monitoring* computes a set of *metrics* that can be queried and read by the *Analysis* component in *push* or *pull* mode. The *Analysis* component checks a set of *conditions* using those *metrics* and may decide to trigger an *alarm* message to the *Planning* component if some condition deviates from the intended behaviour. The *alarm* message is analyzed by the *Planning* component which executes one *strategy* and may require obtaining more *metrics* information from the *Monitoring* component. The output of this *strategy* is a set of actions in a standard language that must be associated to concrete actions and carried on by the *Executing* component. This general function is displayed in the sequence diagram of Figure 7.2.

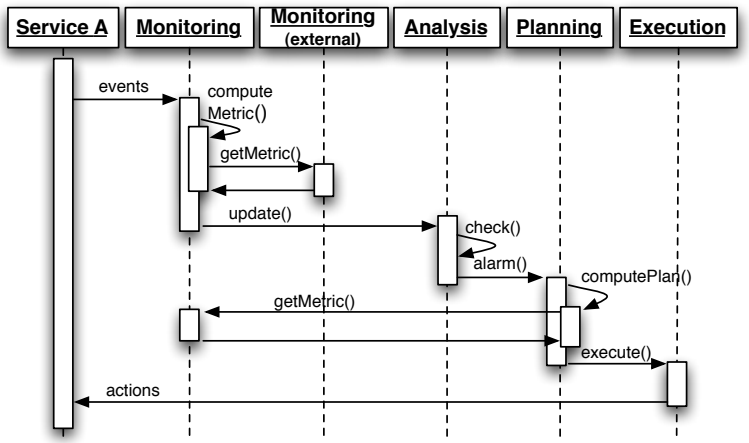


Figure 7.2: Sequence diagram of the framework execution. The different tasks may run in parallel with the functional task of *Service A*

It is important to mention that the steps of the cycle are executed while the managed service continues its task, due to the fact that each NF GCM component comprises its own activity thread. This feature is advantageous because it avoids disrupting the activity of the managed service if it is not necessary, meaning that the managed service can continue serving other tasks while the MAPE components devise a solution for the disrupted condition. Of course, this situation is not possible in all cases and it depends on the nature of the condition. For example, if the disrupted condition is about the a response time longer than expected, the service time can continue service its current request while the MAPE components analyze its situation; but if the condition refers to a low amount of disk space that must be solved by moving or compressing data to make more room, then the service will be disrupted anyway.

The fact that each MAPE component executes its own thread also brings some challenges. Along with having the ability to update and check multiple conditions, it is possible that one *alarm* be triggered while the solution for another one is still being done. This is not a problem if both conditions are not correlated, and by executing concurrent *strategy*, both (or more) situation may be solved. But it is not uncommon to think that may be correlated. In fact, a blocking problem in one service, for example, low storage space, may trigger one *alarm* about it and,

soon after, trigger another *alarm* related to a low response time (as the service is a blocked state). It is in general not an easy task to decide automatically when two *alarms* (or actions) can be executed in parallel in a safe way or not, and the safest solution, stopping the activity of other components to ensure that only one conditions is evaluated simultaneously, invalidates the advantage of concurrent execution. In order to not limit this capability, the MAPE components can use the *MembraneController* of their hosting component to stop and restart the activity of other components when some exclusive execution is required, and leaving this decision to the programmer of the task.

The following sections describe in detail how we have implemented each MAPE NF Component and, when needed, how it is attached to the GCM application.

## 7.2 Monitoring

We have implemented a *Monitoring* component with three basic tasks in mind: (1) collect monitoring information from the GCM/ProActive application; (2) store the monitored information for later use; (3) compute a set of metrics that can be queried and communicated to through a *metrics* interface.

The internal structure of the *Monitoring* component is shown in Figure 7.3.

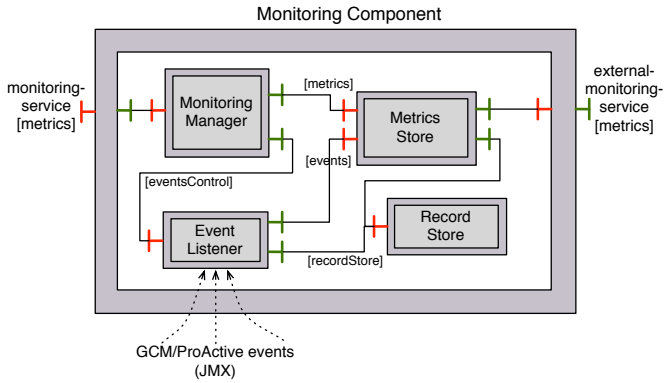


Figure 7.3: Internal composition of the Monitoring component

The *Monitoring* component is a composite that includes the following components:

- *Event Listener*. This component acts as a listener for JMX Notifications coming from the GCM/ProActive middleware. It comprises a server interface *eventsControl* to enable or disable the subscription to events. The Event Listener has two client interfaces. The *events* interface allows to notify about the arrival of new event. The *recordStore* interface allows to store a record about the occurrence of an event. The objective of the Event Listener is to connect to the JMX interfaces of the GCM/ProActive application and provide a representation of the occurrence of events in the form of a *Record*.
- *Metrics Store*. This component implements the *metrics* interface. It comprises a list of *Metric* objects that can compute a value and can use the *recordStore* interface to obtain the data (as a *Record* that it needs to compute those values). The computation of a *Metric* is activated when a new *event* is received, or it can be executed at a periodic interval.
- *Record Store*. This component implements storage for monitoring *Record*s. The interface allows to query the existing records and obtain a subset of them; and allows to store or update existing records.
- *Monitoring Manager*. This component controls the communication with the *Monitoring* components of other GCM components and receives queries from them. It also includes

one client interface for each F client interface of the hosting GCM component. It is used to start/stop the collection of events through the *eventsControl* interface.

### 7.2.1 Instrumentation in GCM/ProActive

The provided instrumentation of GCM/ProActive (Section 6.3.3) allows to aggregate the time spent by a GCM/ProActive application during execution, separating effective communication time, serialization time, waiting time, and serving time, all which are relevant from the point of view of grid-oriented applications.

Service-based applications, however, require more fine-grained analysis. This means, being able to compute the service time of specific requests and the path flow that a request has followed in order to identify the involved services and resources. In a common synchronous environment the addition of an identifier for each request and a set of timers activated and deactivated after each request sent or after each reply received would be enough to discriminate the time used by a service. However, in the asynchronous and grid-oriented environment of GCM/ProActive, a proper decomposition of time is not direct. Regarding the events notified by a GCM/ProActive application (Section 6.3.3), the *replyReceived* notification is sent at the end of every *rendez-vous*, mainly because after this time a synchronous communication has ended, and the caller component can continue its task independently of the called component. Also, the service time incurred by serving a task should take into account the time since the start of the service until the final sending of the response. This seems quite evident, however the use of Future objects for asynchronism provokes a situation where a component may send a reply for a request, because it has finished serving that request, however the response does not yet include the final response, and instead it may contain another Future object that is a placeholder for a response awaited from another ongoing request to another component. So, even if a the component has finished serving a request, it still may contain one *thread* in charge a waiting for the final response to that request and that is in charge of executing an *automatic continuation*. From that point-of-view even if the component has finished executing the task and may be now executing another request, the service time for the original request is still not finished.

We have introduced additional notifications in GCM/ProActive to detect the moment when a final result has been sent to the caller, indicating that the service call has finished and no more additional operations related to that request remains. The list of notifications is shown in Table 7.1 and the moment where these notifications are sent are coarsely shown in Figure 7.4. It is worth to mention that, depending on the moment that the notification is sent, the component $A$ may generate only a *futureUpdate* notification, which contains yet another *Future* (the one that is created by $B$ after calling $C$); or component $A$ may receive both the *futureUpdate* and the *realReplyReceived* notifications in which case the request sent by $A$ is considered as served. By the same reasons, the *body* of component $B$ may generate a *realReplySent* notification when the final result (with additional *Future*s involved is sent), or well a *replyAC* in the case that it is only forwarding a partial results which must still be updated via *automatic continuation*. In the case component $C$, no additional request is sent to other component, and the only notification sent when it finishes serving the request is a *realReplySent*.

| Notification name | Event |
|---|---|
| realReplyReceived | Component receives an updated value with the final result |
| realReplySent | Component sends an updated value with the final result |
| replyAC | Component sends an update value including Future objects |

Table 7.1: Notifications introduced in GCM/ProActive

Due to asynchronism, it is possible that the reception of a Future object update is not necessarily related to the last request that was sent by the caller. We profit of Message Tag insertion

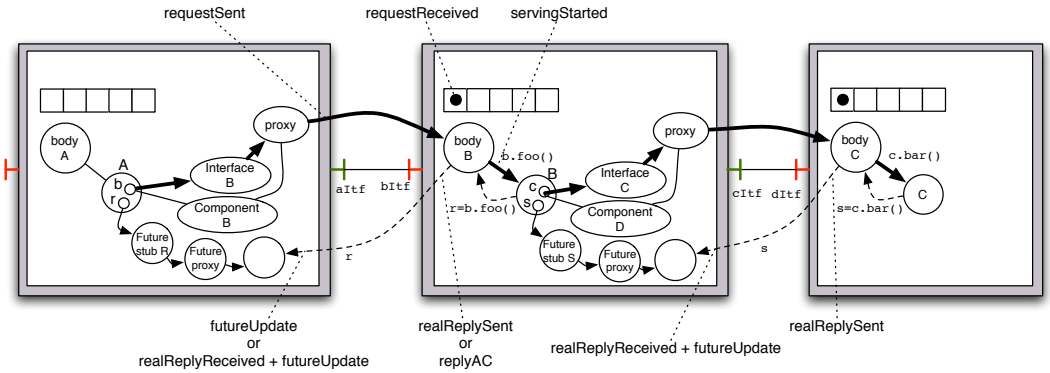


Figure 7.4: Schema of notifications sent during the service of an asynchronous component request

feature to introduce a *tag* in the requests related to monitoring of component requests, called *CMTag* (Fig. 7.5). The *CMTag* is created when a new component request is prepared to be sent, and whose parent request (the request that is being served when this new request is sent) does not include such a *tag*. In that case, the identifier (*rID*) of the current request is set as the *flowID*. If the parent request already includes a *CMTag*, the information stored in that *tag* is used to create the new one, using the same *flowID*. The *CMTag* includes the *rID* of the current request, the *rID* of the parent request (or *null* if there is none), information about the caller component, interface and method called, and a *flowID* that is set to the *rID* of the first request of this flow. This is used as a convenient way to quickly identify all the requests that are part of the same flow without needing to follow each step of the flow.

The construction of the *CMTag* does not impose a big overhead in the processing of a request, as all the information needed is available in the context of the *body* of an active object. In Section 7.2.3 we use this for computing the *request path* of a component request.

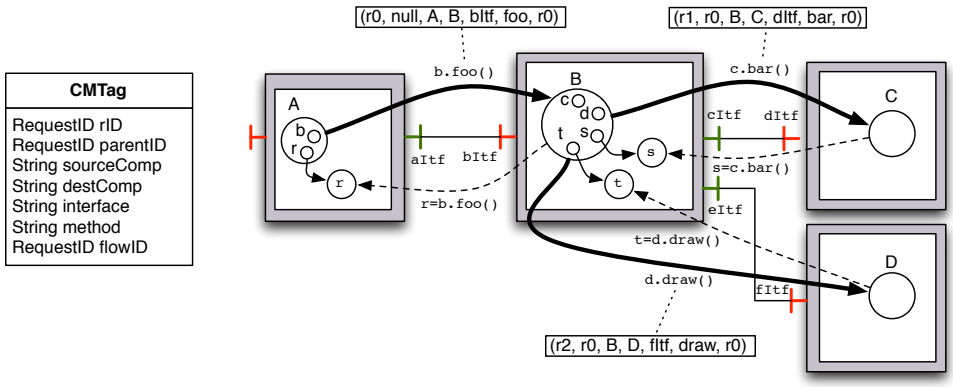


Figure 7.5: Component Monitoring Tag (*CMTag*), and propagation example

Figure 7.5 also shows an example application with four components (the intermediate supporting elements are not shown to simplify the description), and the *CMTag*s propagated through them. In the example, component $A$ makes a request to component $B$ with *rID* $r_0$. While computing $r_0$, $B$ require two calls to components $C$ and $D$, obtaining the results in Future references $s$ and $t$ respectively. However, due to the asynchronism, there is no guarantee in the order that both references will be updated with the final value. The information stored in the *CMTag* of each request is used to associate the response with the original request and determine a causality relationship. The information of the *CMTag* also allows to create a *Record* (see further in Section 7.2.2) that is complemented with the timestamps of an event and determine the serving

time of a request both from the point of view of the caller and from the point of view of the receiver.

### 7.2.2 Model for storing metrics

We have provided a model for defining *metrics* and inserting them into the *Monitoring* component. The implementation is oriented to capture performance related information about the service of a request, however the model has been devised in a way that can be extended to include additional concerns. The class diagram is shown in Figure 7.6

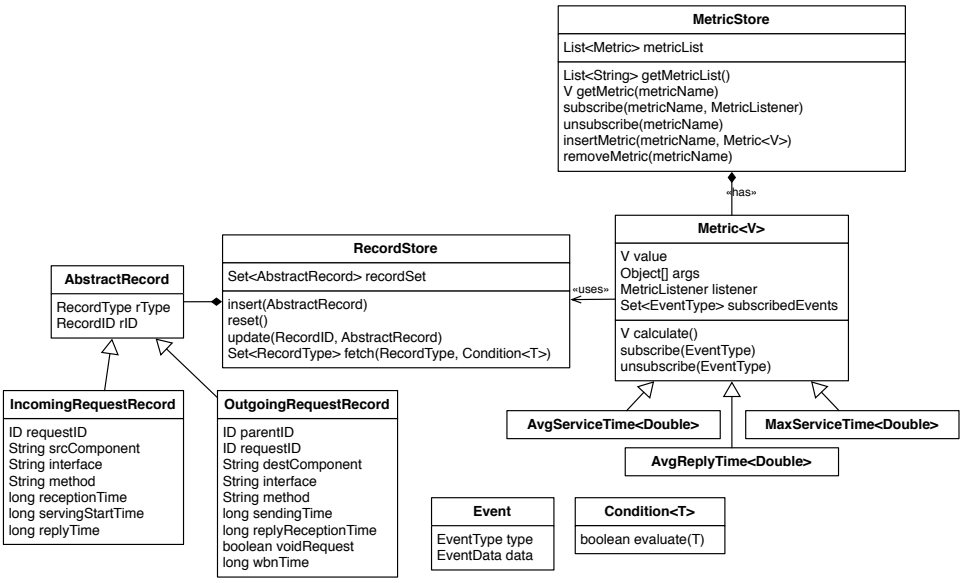


Figure 7.6: Class diagram for managing *Metrics* and *Records* in the *Monitoring* component

The model considers that the *EventListener* has subscribed to the JMX Notifications generated by the *BodyWrapperMBean* attached to the GCM/ProActive component. The *EventListener* transforms the JMX notifications into *Record*s that are stored in the *RecordStore*, which acts as a database. Using the interface of the *RecordStore*, the *Record*s can be inserted, queried and updated. The *fetch()* method can include an object that implements a *Condition<T>* interface. An object with this interface implements a boolean evaluation method that can be applied to objects of type $T$ and can be used to filter the *Record*s obtained from the *RecordStore*.

We have provided two types of *Record*s, with the objective of obtaining performance information about the requests and being able to trace their path. However, by extending the *AbstractRecord* class, other type of *Record*s may be included.

- The *IncomingRequestRecord* stores data about a request that has been received on a functional server interface, including the sender of the request, the time when the request was received, the time when it was taken from the request queue to be served, and the time when the final reply regarding this request was sent.
- The *OutgoingRequestRecord* stores data about a request that has been sent through a functional client interface, including the destination component, the time at which the request was sent, the time at which the final reply was received in case a response is expected, and the time that the component was blocked in *wait-by-necessity* (*wbn*).

Upon reception of a JMX Notification, the *EventListener* may decide to create or update a *Record* in the *RecordStore*. For example, when a *requestSent* notification is received, the *EventListener* builds a new *OutgoingRequestRecord* using the timestamp information from the

notification, and the request data from the *CMTag*, and stores it in the *RecordStore*. The *Record* is however incomplete, as the time of the reception of the reply is yet unknown. Later, when a *realReplyReceived* is received, the *EventListener* recovers the corresponding record thanks to the *id* indicated in the *CMTag* and updates the *OutgoingRequestRecord* with the *replyReceptionTime* obtained from the new notification. By correctly propagating the request *id*s through the *CMTag*s, this scheme can work regardless of the effective arrival time of the notifications.

The *MetricStore* component stores objects of type *Metric*<*V*>. This is an abstract class that profits of Java generics to describe an object that computes a value of type $V$. The *Metric* is instantiated with a set of arguments and must implement a *calculate()* method that updates and return a value of type $V$. When inserted into the *MetricStore*, the *Metric* object is provided with a reference to the *RecordStore* (through the binding from the *MetricStore* component), so it can send queries to it and obtain the *Record*s it may need. The logic implemented by a *Metric* element can be invoked in three different ways: (1) the *Metric* can be subscribed to set of *Events* when inserted, so that any occurrence of one of these *Events* will trigger an update of the *Metric* value; (2) the *Metric* can be updated at a regular interval, in which case the *MetricStore* uses a thread to periodically execute the *calculate()* according to the specified period; and (3) no update mechanism is specified, and the *Metric* is only update when it is requested from the *Monitoring* server interface.

As an example, Listing 7.1 shows an implementation of a *Metric* used to obtain the average response time of all requests received by a component on a given interface. In our implementation, this *Metric* is subscribed to an *requestServed* event, so that it is updated every time that a request has been served by the hosting component.

```
public class AvgRespTimePerItfMetric extends Metric<Double> {
      ...
      public Double calculate(final Object[] params) {
            List<IncomingRequestRecord> recordList = null;
            recordList = records.getIncomingRequestRecords(new Condition<
                IncomingRequestRecord>(){
                  // condition that returns true for every record that
                      belongs to a given interface
                  @Override
                  public boolean evaluate(IncomingRequestRecord irr) {
                        String name = (String) params[0];
                        if(irr.getInterfaceName().equals(name)) {
                              return true;
                        }
                        return false;
                  }
            }
            );
            // and calculates the average
            double sum = 0.0;
            double nRecords = recordList.size();
            for(IncomingRequestRecord irr : recordList) {
                  if(irr.isFinished()) {
                        sum += (double)(irr.getReplyTime() - irr.
                            getArrivalTime());
                  }
            }
            value = sum/nRecords;
            return value;
      }
}
```

Listing 7.1: Implementation of the avgRespTimePerItfMetric

The *MetricsStore* also provides the *Metric* objects with interfaces bound to all the *Monitoring* components of the other GCM/ProActive components that the hosting component is bound to. The *Metric* can obtain, from the *MetricStore*, the name of all components bound and use it to make invocations on each *Monitoring* component that it may need. The actual binding from the *Monitoring* interface follow the same cardinality of the corresponding functional interfaces, meaning that multicast *metrics* interfaces are possible.

### 7.2.3 Example: Request path computation

We exemplify a distributed computation of a *metric* called *RequestPath*. The objective is to identify the path followed by the service of a specific request through a set of services, specifying the time taken by the service to finish the task and delivering the final response. Figure 7.7 shows an example of a GCM/ProActive application including a composite component and a multicast interface, where a request with *id* $r_0$ is sent by component $Z$ and it triggers a set of additional requests shown with bold arrows and their respective *id*.

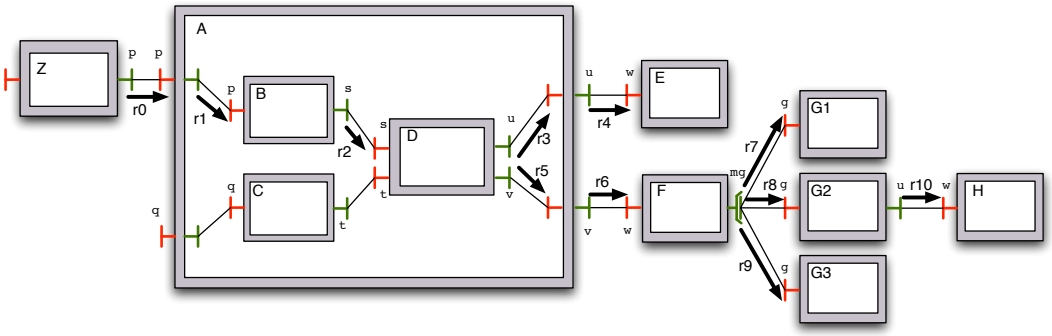


Figure 7.7: A GCM/ProActive application and a description of a flow triggered by request $r_0$

The result of the computation is a *RequestPath* object, which is a tree-like structure of *PathItem* objects. A *PathItem* object includes data about a particular request: the components, interfaces, and method involved, the time that took for the a source component to obtain a response to the request (this is, the service time from the point of view of the client), called "client time", and the time that the destination component took to send the response (this is, the service time from the point of view of the server), called "server time". The computation of each *PathItem* requires the participation of both the "client" and the "server" components of each request. As both have measured the time to serve a request from their position, it is expected that these times defer due to middleware tasks and network propagation. By analyzing a complete *RequestPath* it is possible, for example, to identify the component where most of the time was spent, and how much of that time corresponds in middleware/network propagation.

The computation relies on the information stored in the *RecordStore* component of each service. The *RequestPath* object is built by asking each component involved in the service of a request to compute a partial *RequestPath* and merging the responses on a complete *RequestPath*. The response can be either a single *RequestPath* or more than one partial *RequestPath* which must be merged to compose the complete *RequestPath*.

The *Monitoring* component of $c$ receives a *computeRequestPath* invocation, with the *requestID* (*rid*) $r_0$ of the incoming request, the *flowID* $f_0$, and a set of names of *visited* components. The expected return value is a tree representing the path followed by the request from the moment that it was received on $c$. As every *Monitoring* component is expected to return a value, it is necessary to keep a list of visited components in order to not send an invocation to a component that is already computing a subtree, otherwise a deadlock may happen. Each *Monitoring* component executes the same *metric*:

- Add $c$ to the set of *visited* components.

- Obtain the *IncomingRequestRecord* $I_0$, whose *id* is $r_0$.
- Create a *requestPath* $rp_0$ element with the *flowID* $f_0$.
- Get $I$, the set of *IncomingRequestRecord*s with *flowID* $f_0$, and whose request reception time $t_{reqRecv}$ is later or equal than the one of $I_0.t_{reqRecv}$. This set includes at least one request ($I_0$), or more than one request if the hosting component has been invoked two or more times during the same *flowID* (even if it was through another server interface).
- For each *IncomingRequestRecord* $I_i$ in the set $I$:
  - Create a new *pathItem* $p_i$ that will be the *head* of a branch that starts in this component. $p_i$ includes the request *id* of $I_i$ and the "server time" obtained from it as $I_i.t_{replySent} - I_i.t_{reqRecv}$.
  - Obtain the set of all the *OutgoingRequestRecord*s whose parent has the *id* $I_i.rid$, and add it to the set $O$. The set $O$ may be empty if no additional request was sent while serving the request $I_i.rid$.
  - Order the elements of $O$ by their sending time $O_j.t_{reqSent}$. For each *OutgoingRequestRecord* $O_j$ in the set $O$:
    - If the destination component is not in the *visited* list, obtain the *requestPath* $rp_j$ from the destination component by calling *computeRequestPath* with the *id* of $O_j$.
      - Add the set of *visited* components obtained from $rp_j$, to the current set of *visited* components.
      - Add the *head* of $rp_j$ as a child of $p_i$.
      - Complete the information in the *head* of $rp_j$ with the "client time" obtained from $O_j$ as $O_j.t_{replyRecv} - O_j.t_{reqSent}$.
      - Copy the other branches possibly obtained in $rp_0$ to $rp_j$.
    - Else, create a new *pathItem* $p_j$ that will be the *head* of a branch obtained from another *requestPath*. $p_j$ includes the request *id* of $O_j$ and the "client time" obtained from $O_j$ as $O_j.t_{replyRecv} - O_j.t_{reqSent}$.
      - Add $p_j$ as a child of $p_i$.
- After the previous step, the *requestPath* may include a set of branches apart from the one with *rid* $r_0$. Those branches must be checked against the list of *incomplete* leafs. If some of them have the same *rid*, then they must be merged by including the remaining information.

At the end of the execution, the result is only one branch representing the *request path* of $r_0$, and no additional branches. As an example of an intermediate computation, when the *computeRequestPath* method is called on component $A$, its *Monitoring* component finds that three incoming requests have been made: $r_0$, $r_3$ and $r_5$. For each one of these requests, a *pathItem* object is created with the "server time" obtained by the records stored in $A$, and a call is made to the *Monitoring* component of the destination component to complete the branch (Fig. 7.8(a)). The call is made on each involved component, and a partial branch is created for each request (Fig. 7.8(b)). The leaves of these branches may have some incomplete information, which may be obtained in the head of the other branches before computing the complete branch (Fig. 7.8(c))

The final result is only one branch representing the complete request path, as shown in Figure 7.9. The arrows pointing to an empty space actually are branches that have already been computed and merged, and they are shown this only for clarity.

### 7.2.4 Additional considerations

Though this implementation is enough to fit our purposes, it does not impose a particular way to implement the monitoring task. One of the advantages considered when using a component-based approach is that the functionality of each element can be encapsulated from the rest. That said, the *RecordStore* can be implemented by a database engine if more complex processing

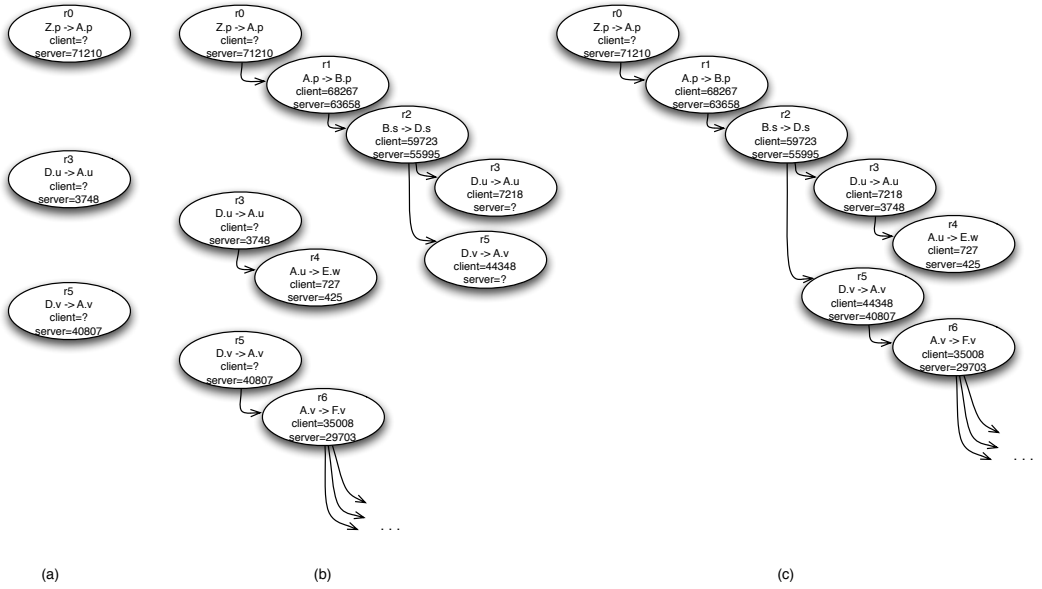


Figure 7.8: Tree obtained from a request path computation

is needed. Also, the functionality of the *RecordStore* and the *EventListener* can be partially supported by a Complex Event Processing engine like Esper, while keeping the rest of the MAPE components unchanged.

## 7.3 Analysis

The *Analysis* component is implemented with the tasks of (1) receiving the description of an SLO and store it as a verifiable condition, (2) identify the metrics required to verify the associated condition, and (3) verifying at runtime the stored conditions, either by subscription to metrics or by periodic updates, and generate alarms if needed.

The internal composition of the *Analysis* component is shown in Figure 7.10

The *Analysis* component is a composite that includes the following NF components:

- *SLO Store* stores the list of SLO objects and maintains their status. An SLO can be disabled and re-enabled at runtime, in particular when an action is being taken regarding this SLO.
- *SLO Analyzer* receives the SLO object received by the SLOs interface, and transforms it to an internal representation that can be stored in the SLO Store, and determines the metrics needed to verify the compliance. It is expected that by replacing this component with an appropriate implementation, other models for representing SLOs can be used.
- *SLO Verifier* checks the compliance of the enabled SLOs from the SLO Store. It is able to communicate with the *Monitoring* component in order to obtain the values of the metrics needed. The metrics can be obtained by subscribing to them and checking them every time an update is obtained from the *Monitoring* component, or in a periodic way.
- *SLA Manager* receives the representation of an SLO and manages the analysis process. It uses the *SLO Analyzer* to obtain a common representation of an SLO and store it in the *SLO Store*; and gives this SLO object to the *SLO Verifier* to check the compliance.

### 7.3.1 SLO Representation

One of the main challenges of the *Analysis* phase is to be able to interpret an SLO, expressed as a condition, and be able to verify it by obtaining the required runtime values. As shown in Section

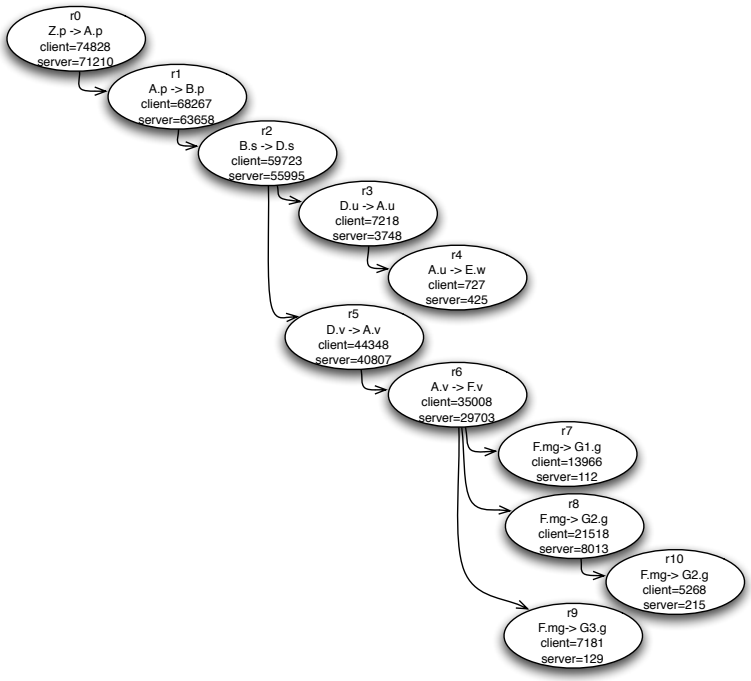


Figure 7.9: Tree obtained from a request path computation. Some portions of the branches have been omitted for clarity.

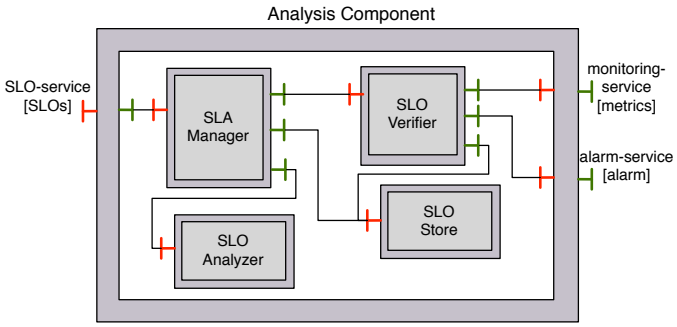


Figure 7.10: Internal Composition of the Analysis component

3.1.2, several languages and models have been proposed to represent and verify SLAs. In order to demonstrate the use of our framework we have provided a simple yet useful model. Although not powerful enough to express complex conditions, this model allows to obtain a simple and quick runtime verification.

An SLO object represents a triple $\langle metricN, comparator, threshold \rangle$, where *metricN* is the name of a metric that can be obtained through the *Monitoring* component; *comparator* is a condition that can be applied over the metric; and *threshold* is threshold value that can be used by the *SLO Verifier* to decide if an alarm must be generated. The *SLORule* object implementation, as shown in Figure 7.11, considers the *comparator* as *Condition<T>* object, where $T$ is the type of the value obtained by the *metric* and against which the *threshold* value is compared. Optionally, the SLO object can include a *preventiveThreshold* value, that can be used to decide the level of the alarm generated (if any).

On the other side, when an *Alarm* object is generated, it includes the *SLORule* object that was being verified when the alarm was triggered, and an *AlarmLevel* whose aim is to indicate a severity about the fault. We have defined three levels: *OK*, *Preventive*, *Faulting*, to indi-

cate respectively that the SLO is accomplished, that the obtained value is between the *preventiveThreshold* and the *threshold* value. Actually, when the SLO is accomplished (*OK* level), no alarm should be generated.

### 7.3.2 SLO Verification

The sequence for verifying the compliance of an SLO is shown on Figure 7.11. The *Analyzer* component receives a description of an SLO in a predefined language (1). In this example we have assumed an XML-based language, although the objective is to support existent languages by implementing the task of the *SLO Analyzer*. In the next step, the *SLO Analyzer* converts the representation to an *SLORule* object (2), that is stored in the *SLO Store* (3). The *SLO Store* may have several *SLORule* elements (4).

The *SLA Manager* coordinates all the steps. The *SLA Manager* orders the *SLO Verifier* to initiate the compliance checking of the SLOs stored in the *SLO Store* (5). Depending on the characteristics stored for the *SLO*, the *SLO Verifier* can monitor the compliance in two modes: either using a *pull* mode where the *SLO Verifier* executes periodically a *getMetric* call on the *Monitoring* component to recover the needed metric(s) and check the SLO; or a *subscription* mode, where the *Analysis* component subscribes to the corresponding metric on the *Monitoring* component (6) and the checking is performed upon every update notification received (7). In case that the obtained value for the metric does not satisfy the condition specified by the SLO, an alarm with *Faulting* level is created and sent through the *alarm-service* (8).

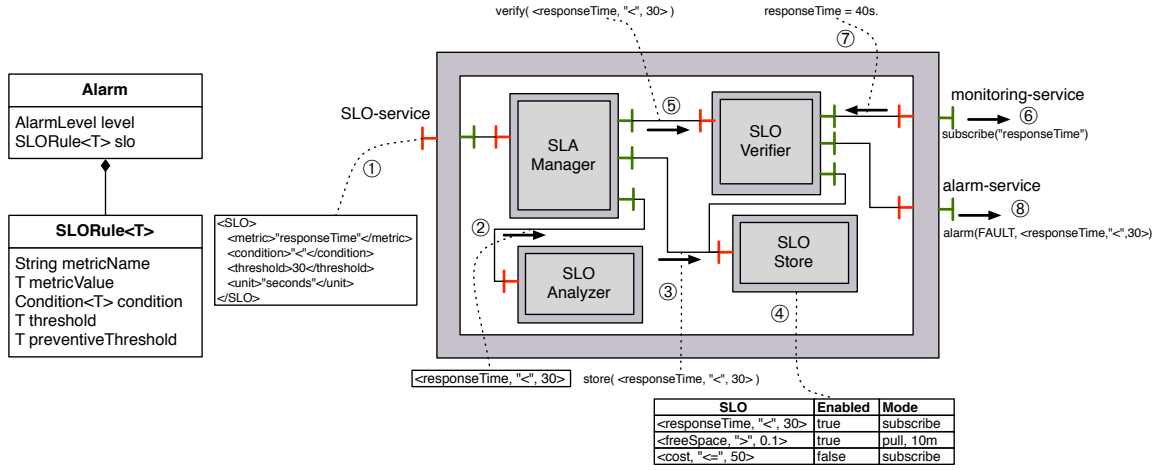


Figure 7.11: Steps in SLO verification

A preventive approach can be provided by using the optional *preventiveThreshold* defined for the *SLORule*, however the way to actually consider the preventive threshold, and how to interpret its value is a task that corresponds to the implementation of the *Condition*. The design of the *Condition* may consider the existence of the *preventiveThreshold* and discriminate if the obtained value for the metric is located outside of both thresholds, or between the preventive threshold and SLO threshold, and in that case return an appropriate *AlarmLevel* that will be used to construct the *Alarm* object.

Listing 7.2 shows an example of a simple *Condition* that obtains *Double* value from a metric and compares it to a maximum value specified by the *Threshold* values. Another example may consider, instead of a fixed preventive threshold value, a percentage of the maximum allowed value. For example, an SLO that checks the available disk space may consider a *preventiveThreshold* as a percentage of minimum free space allowed, and the implementation of the *Condition* must interpret this percentage.

```
public class PreventiveCondition implements Condition<Double>
{ ...
      public AlarmLevel evaluate(final SLORule<Double> rule) {
         double currentValue = rule.getMetricValue();
         if(currentValue < rule.getPreventiveThreshold()) {
           return AlarmLevel.OK;
         }
         else if(currentValue < rule.getThreshold()) {
           return AlarmLevel.Preventive;
         }
         else {
           return AlarmLevel.Faulting;
         }
      }
}
```

Listing 7.2: An implementation of a simple Condition that uses a preventive threshold

The example assumes that the *SLORule* is subscribe to the updates of the *Double* value of a *Metric*. For this reason, the *evaluate()* method only needs to read the value that has already been updated into the *SLORule*. In other situations, the invocation to the *getMetricValue()* method of the *SLORule* may involve an invocation to the *Monitoring* component to obtain the needed value.

## 7.4 Planning

The *Planning* component receives an *Alarm* object and, based on the contents, it takes the following actions: (1) associate the problem detected with one available strategy, (2) execute a strategy, (3) send the actions obtained from the strategy to the *Execution* component.

The internal composition of the *Planning* component is shown on Figure 7.12.

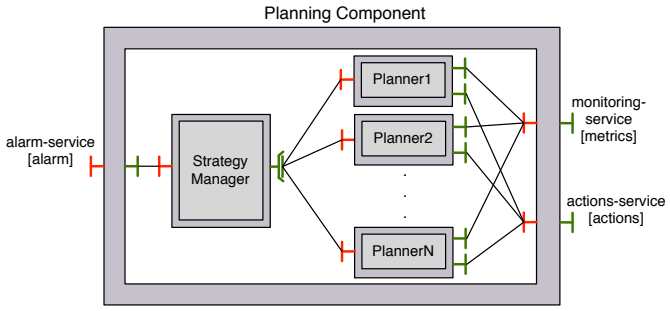


Figure 7.12: Internal Composition of the Planning component

The main component is the *Strategy Manager*, whose task is to receive the *Alarm* object that includes the description of the *SLO* that provoked the alarm. The *Strategy Manager* implements a simple diagnosis method based on a table that associates the metric mentioned in the faulting *SLO* with a list of available strategies. The objective of this table is to decide on a set of strategies that should be able to solve the problem that has caused the SLO fault and restore that condition on the component.

Once a particular strategy has been chosen, the *Strategy Manager* invokes the corresponding *Planner* component to execute it and generate a plan. The *Strategy Manager* can select from a list of several *Planner* components bound to the *multicast* interface. This kind of interface is used to support an undefined number of bindings. As a fail-safe support a default *null* strategy that does not generate an action is always accessible.

### 7.4.1 Selection of a Strategy

The first step of the *Strategy Manager* component is the selection of the *Planner* component to invoke. This decision is guided by a table of ordered *DecisionEntry* objects. Each entry represents a criterion based on the *metricName* and *AlarmLevel* reported by the SLO. An extension of this implementation could consider also, for example, the range of the current value of the metric or other criteria to make a more fine-grained decision.

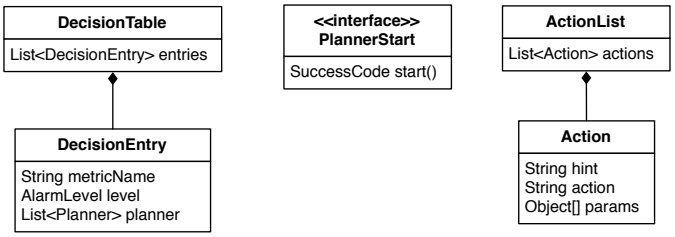


Figure 7.13: Objects related to the *Planning* component

The *DecisionEntry* object, as described in Figure 7.13 includes the name of the metric associated to the received *SLO*, the *AlarmLevel* indicating if the *SLO* has been violated, or if it is a preventive notification, and a list of references to *Planner* component that should be chosen in this case. Once the *metricName* and *AlarmLevel* are matched, the first reference to a *Planner* component is executed. A *Planner* component implements a *PlannerStart* interface that comprises a single method that is the entry point to the execution of the strategy.

The method *start()* from the *Planner* component is expected to return a *SuccessCode* value indicating if the *Planner* component succeeded to generate a plan or if it encountered a problem, for example, if some required input value could not be obtained, or if the strategy is not able to produce a response in this situation. If the *Planner* component does not succeed, then the *StrategyManager* invokes the next *Planner* component of the list until obtaining a *Success* response. If all *Planner*s are tried a default *Planner* that implements an empty strategy is executed, which does not generate any actions, and returns a *Null* code. The order of the list defines the order in which the entries are checked.

### 7.4.2 Implementation of a Strategy

Several different strategies or planning algorithms can be used to produce a list of actions to be executed over the component, and the solution may include techniques from different areas, like those shown in Section 3.1.3. The support provided for implementing planning strategies is the insertion of a component called *Planner* that encapsulates a planning algorithm. A *Planner* component must implement a server interface with a *start()* method that is the starting point for executing the planning algorithm. The planning algorithm can use an interface to access the *Monitoring* and obtain the value of the metrics it may require. The output of the planning algorithm should be an *Action* object or a sequence of these *Action*s, each one of them representing an action that must be executed on the hosting component.

The insertion and removal of strategy is one of the features that we see as important to obtain a dynamic management layer. Through a console application we provide an interface to interact and modify the composition of the *Planning* component allowing to insert and remove *Planner* components based on the description of an implementing class that complies with certain requirements (basically, to support *PlannerStart* interface), which is described in Section 7.6.

An example implementation of a strategy that find the bound component with the higher cost and generates an unbind action is shown in Listing 7.3. The implementation considers as starting point the method *SuccessLevel start()*, and uses the *Monitoring* interface to collect the *cost* metric of all the bound components. Then it invokes a previously known discovery

service to obtain a replacement component and, in case it finds it, creates an action in a generic language to trigger the change. The sending of an action is an asynchronous call with void return, meaning that once the actions are sent, the *Planner* component finishes its job. The *SuccessCode* return only indicates if the *Planner* component was able to create an action or not. The concrete execution of the action, with its possible success or failure, is a concern of the *Execution* component.

```
public class ReplacementPlanner implements PlannerStart
{ ...
      public SuccessCode start(Object[] params) {
            double maxCost = 0.0;
            Component maxComp = null;
            for(comp : getBoundedComponents()) {
                  double cost = monitoringService.getMetric("cost",comp);
                  if(cost > maxCost) {
                        maxCost = cost;
                        maxComp = comp;
                  }
            }
         Component replacement = findReplacement(maxComp);
         if(replacement == null) {
            return SuccessCode.FAIL;
         }
         Action action = new Action("replace(" + maxComp.getName() + ","
         + replacement.getName() + ")"); actionService.sendAction(action);
         return SuccessCode.OK;
      }
  ...
}
```

Listing 7.3: An implementation of a strategy to replace a bound component

Figure 7.14 shows the steps involving the reception of an *SLO*, the selection of a *Planner* that uses the *Monitoring* interface to obtain the values of the *metrics* it needs, and that sends actions through the *Actions* interface.

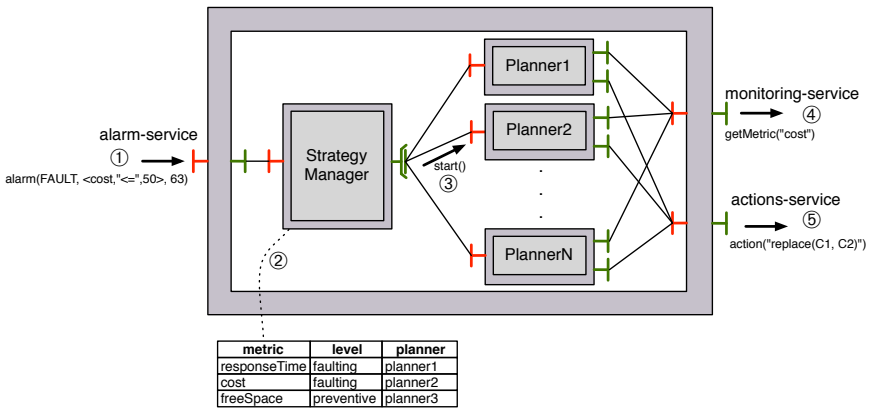


Figure 7.14: Steps in the *Planning* component while selecting an executing a strategy

### 7.4.3 Representing actions

Several questions remain. For example, what is the kind of *Action*s that can be executed, and how can they be represented. In a similar way to the *Monitoring* component, whose actual capa-

bilities depend on what the supporting platform can provide, our range of actions are limited by the kind of actions that the GCM/ProActive middleware supports. At the level of the component implementation, GCM/ProActive provides features for structural reconfiguration of components, by modifying bindings, inserting and removing components, and modifying attributes of a component. These kind of actions can be specified using an intermediate DSL language to describe them.

Nevertheless, a *Planner* component does not need to speak this specific language, but just to agree in one language that can be interpreted by the *Execution* component. In the example presented in Section 7.4.2, the generated action describes a command in a language that is not necessarily a single primitive action. The appropriate translation that must be done by the *Execution* component must generate the corresponding concrete action or sequence of actions that can be executed depending on the middleware support.

### 7.4.4 Additional Considerations

Given the active nature of GCM components, where each component has a *thread* of execution, it is a certain possibility that in a given situation more than one *SLO* may trigger an *Alarm* on the *Planning* component. The *Strategy Manager*, consequently may trigger the execution of different *Planner* components concurrently, and this may potentially raise problems as different planning algorithms may trigger conflicting actions. For example, one *Planner* component may decide to unbind a component, and other may decide to only modify a parameter. However other set of planning algorithms can be executed concurrently without any interference.

In our implementation we have chosen not to restrict *a priori* the concurrent execution of *Planner* components, as we believed that this concurrency problems must be taken into account at the moment of designing and introducing the planning algorithm inside the *Planner* component. It is also possible to devise a model where the conflicting behaviour can be foreseen and avoided in advance, however this has not been a focus of our work.

## 7.5 Execution

The *Execution* component receives a set of *Action* objects and translates them into concrete actions that can be run over the component-based application. The *Execution* component (1) translates the received *Action* into commands that can be executed on the application, and (2) executes the commands by connecting to the means provided by the supporting middleware.

The composition of the *Executing* component is shown on Figure 7.15

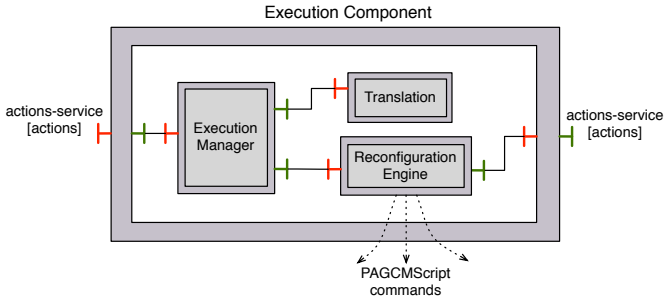


Figure 7.15: Internal Composition of the Execution component

The *Execution* component is a composite that includes the following NF components:

- *Translation* component receives an *Action* object, commonly created by a *Planner* component, and translates it to the appropriate executing support. The intention is to separate

the description of *Action*s that may be used by the *Planner* components (and they may actually be different), from the way to describe these actions on the supporting platform. The *Action* element, as shown in Figure 7.13, can use a *Hint* field to guide the *Translator* in providing an appropriate representation.

- *Reconfiguration Engine* encapsulates an interpreter for a scripting language designed for GCM/ProActive applications, called *PAGCMScript*. This scripting language allows to describe and execute actions over GCM/ProActive components, and it is also able to delegate tasks to other components.
- *Execution Manager* coordinates the steps of the *Execution* component. Uses the *Translation* component to obtain commands to give to the *Reconfiguration Engine* and is also responsible for synchronization, avoiding the concurrent execution of reconfiguration over a given component that may generate inconsistencies.

In a similar way to the *Monitoring* component, the *Execution* component has one client *Action* interface bound to the server *Action* interface of each functional component to which the hosting component is bound, and with the same collective nature. The purpose of this is to be able to send actions to each specific bound component, allowing to implement distributed modifications. The decision about the locality of the execution is taken by the embedded reconfiguration engine using the introspection capabilities of GCM/ProActive.

### 7.5.1 PAGCMScript

GCM/ProActive provides a scripting language called PAGCMScript to describe actions that can be executed in GCM/ProActive applications. These actions include the management of bindings and composition relationships, the creation, deployment, undeployment and removal of components, the management of the life cycle of the components, and interaction with their NF interfaces.

PAGCMScript was developed as an extension of the scripting language FScript [DLLC09], mentioned in Section 3.1.4. FScript provides a simple means to extend the model and define primitive actions as Java classes that can be used in more complex scripts. Like FScript, PAGCMScript relies on the FPath domain-specific language, and models a GCM/ProActive application as a directed labeled graph where components, interfaces and attributes are nodes of the graph, and their edges define binding and composition relationships, and the ownership of interfaces and attributes. The extensions added in PAGCMScript include the GCM specific features like support for collective multicast/gathercast interface, migration of components between virtual nodes, and remote deployment and execution of components.

Using the PAGCMScript model, a component can be aware of its bindings and send actions to external components using the *Actions* interface. This characteristic allows to implement scripts that can be executed in a distributed way by delegating the execution to the involved components.

Listing 7.4 shows an example of a replacement action defined using PAGCMScript. Once defined, this action can be invoked from other scripts. The actual implementation of the primitive methods used is a Java class that takes care of the platform details.

```
action replace(oldComp, newComp) {
      for p : $oldComp/parent::* {
            add($p, $newComp);
      }
      for client : $oldComp/interface::*[client(.)][bound(.)] {
            itfName = name($client);
            server = $client/binding::*;
            unbind($client);
            bind($newComp/interface::$itfName, $server);
      }
```

```
        for client : bindings-to($oldComp) {
              itfName = name($client/binding::*);
              unbind-unit($client, $oldComp/interface::$itfName);
              bind($client, $newComp/interface::$itfName);
        }
        copy-attributes($oldComp, $newComp);
        copy-state($oldComp, $newComp);
        for p : $oldComp/parent::* {
              remove($p, $oldComp);
        }
}
```

Listing 7.4: Replacement action using PAGCMScript

The *Reconfiguration Engine* component embeds a *PAGCMScript* engine able to interpret and execute the commands, by interacting with the GCM/ProActive application.

### 7.5.2 Example: Replace a remote component

As an example, consider a GCM/ProActive application like the one shown in Figure 7.16, where the *Planning* component of $Z$ has issued an action to replace the slowest component in the composition with another one. The reference of the new component has been provided as a parameter, however the replacement action must be taken within the context of the component that is bound to the old component.

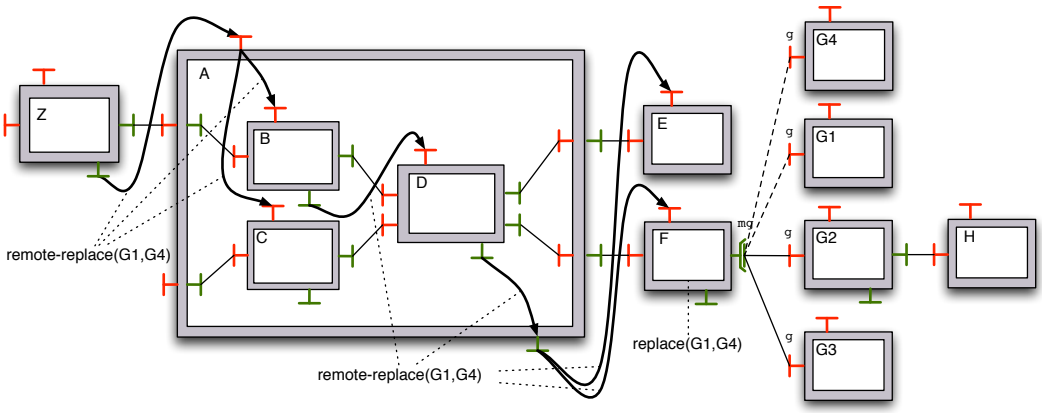


Figure 7.16: Propagating an action to a remote component using PAGCMScript

The corresponding action on each *Execution* component is described in Listing 7.5.

```
action remote-replace(oldComp, newComp) {
   server = $this/binding::oldComp;
   if ($server == null) {
     for externalActions : $this/binding::*[client(.)][bound(.)] {
       invoke($externalActions, "remote-replace", union($oldComp, $newComp));
     }
   }
   else {
     replace($oldComp, $newComp);
   }
}
```

Listing 7.5: Remote replacement action using PAGCMScript

The *remote-replace* action is executed in each component and propagated to through the bound components until finding one that is bound to the old component. If a branch does not find a proper place to execute the action, it is ignored. Regarding Figure 7.16, we must remind that all the propagation actions take place through the NF interfaces (shown in the top-side and in the bottom-side) of each component, which are internally bound to the corresponding interfaces of the *Execution* component present in each membrane, which are not explicitly shown in Figure 7.16.

It is important to mention that whereas FScript engine relies on a model of the complete Fractal application, the PAGCMScript engine does not make such assumption as the application and separate parts of it may evolve in an independent way, making costly the task of maintaining a complete coherent view on each component. Instead, PAGCMScript uses the Fractal model to be aware of the environment of its own hosting component and its relationships to other components. This information can be used to trigger actions locally on the hosting composite component, or in a subcomponent, as the composite components know their complete internal model, or send it through one of their *Actions* interface to any of the bound *Execution* components.

### 7.5.3 Additional Considerations

The use of PAGCMScript allows to interact in a more simple way with a GCM/ProActive application than by programming the Java code, which requires a better knowledge of the GCM/ProActive API. However, this does not impose that all the operations must be done through PAGCMScript. In fact, PAGCMScript provides an extension mechanism (itself inherited from the core FScript) that allows to include additional methods that can be invoked from the script, and which execution is carried on by a user defined Java class which can access the full GCM/ProActive API.

The set of actions provided by PAGCMScript comprises the structural reconfiguration of a component-based application, the creation or removal of components, the lifecycle (start/stop) of a component, and their deployment or undeployment over remote nodes. This set effectively constrains the actions that can be taken by *Planner* components. All along the presentation of the framework we have pushed the advantage that the component-based approach to develop each phase allows to independently evolve each implementation. However, this advantage must be taken with care, as this does not mean that each implementation can be developed blindly. As in any application whose function is based on the interaction of components through interfaces, the methods to be used and the objects used to represent *SLO*s and to describe *Action*s must be agreed and understood by each implementation in the *Analysis*, in the *Planning* and in the *Execution* phases.

## 7.6 Console Application

We have described the implementation of each phase of the MAPE components inside the membrane of a GCM/ProActive component-based application. However, the effective insertion and modification of the components inside the membrane is a matter that has not been addressed yet.

According to the ideas we have pushing through this thesis, the insertion, removal and connections of the MAPE components with the running application is not a matter that the programmer of the application, or at least, of its functional part, should need to take care of. In the context of the GCM/ProActive implementation we have provided an extension of the GCM API for component creation, and developed a basic console application that can take care of the management of the MAPE components inside the membrane of the GCM/ProActive application, so to the keep the functional and non-functional concerns as much separated as possible.

The GCM API defines a method to instantiate a component specifying what is called its *functional type* (F type) and its *non-functional type* (NF type). The F type comprises all functional interfaces defined for the component, and the NF type includes all the NF interfaces (also called

*control* interfaces). To support the interaction with the MAPE components we have defined, the component requires at minimum the presence of the appropriate NF interfaces. In fact, according to our framework, a component does not need to be instantiated with the MAPE components inside. Instead, they can be inserted and removed at runtime; however the corresponding interfaces must exist at the moment of instantiating the functional component.

Listing 7.6 shows one of the methods provided by the GCM API to instantiate a component based on its F and NF type; and the method *createMMType* to create a *Type* object capable of supporting the monitoring and management interfaces of our framework. This method uses the functional type of a component to create a NF type that includes the interfaces for *Monitoring*, *SLOs* and *Execution* according to the criteria given in Sections 7.2 to 7.5. Recall that, in the case of the *Monitoring* and *Execution* components, one NF client interface is created for connecting to the corresponding NF server interface of each functional component bound, and using the same collective characteristic (either singleton or multicast). The method also needs to consider the hierarchy of the component, as in the case of composite, additional internal NF interfaces must be created to be able to interact with the subcomponents.

```
/**
 * Instantiates a GCM component, according to their fType, nfType, specifying
 * a content and their properties in the controllerDesc
 *
 */
public Component newFcInstance(Type fType, Type nfType, any contentDesc, any
    controllerDesc);

/**
 * Creates a type for GCM component that can be used to provide Monitoring and
 * Management capabilities (that's why it is 'MMtype'). The type is created
 according to the F interfaces * of the component, and its hierarchy.
 *
 */
public Type createMMtype(PAGCMTypeFactory typef, Type fType, String hierarchy
    );
```

Listing 7.6: GCM API for instantiating components, and provided method for building an NF type

Once a component has been instantiated using the *newFcInstance* method and the NF type obtained with *createMMtype*, the component can be started and executed in the usual manner. From the functional point of view this is only place where we introduce a modification in the component development.

### 7.6.1 Inserting MAPE components in the application

The insertion of the MAPE components in the membrane is carried on by the console application, that uses the GCM API to instantiate NF components, insert them in the membrane of a GCM/ProActive component, and bind them to the appropriate interfaces, which can belong to the hosting (parent) component, or to the other MAPE components. Being based on the GCM API, the Console application could be replaced by any other external application that use the GCM API to manipulated the components inside the membrane.

The Console works with a reference on a single component, in which case this component is the target of all the actions related to the insertion and removal of the MAPE components; or it can work with a set of components, in which case all these components are considered as "managed components" and every action related to the MAPE components is done over them.

The console application obtains a reference on a single component using a RMI registry looking up on its "Component" interface (recall the remote objects that represent a GCM/ProActive

component, shown in Section 6.3), read from an RMI Registry, using the command “a” (add). Once the reference has been obtained, the GCM component is adde to the list of “managed components” of the Console. Additional components can be added by inserting their RMI references. The list of “managed component” can be retrieved using the “ls” command. From the list of “managed components”, the user can select one of them as the “current” component with the command “use”.

The Console includes a convenience command called “disc” (from “discover”) that uses the introspection capabilities of GCM to find references on all the components (internally and externally) bound to the current component. This command avoid to manually finding and entering a reference for each component that needs to be managed. The command executes recursively, so that all the bound elements are reached. Once executed, the command “ls” shows the available components, and the command “desc” (from “describe”) shows their information, describing their name, interfaces and attributes.

The MAPE components can be inserted using the command “addMon”, “addAnalysis”, “addPlanning”, and “addExecuting” respectively. Each command instantiates the corresponding component and its subcomponents as defined in the previous section, and inserts them into the membrane of the current GCM component. Once inserted, the command also attempts to bind them to the corresponding external and internal interfaces, and to the other existing MAPE components that can be also present in the membrane. The commands “removeMon”, “removeAnalysis”, “removePlanning”, “removeExec” stop and remove the corresponding components from the membrane.

### 7.6.2 Interacting with the MAPE components

Once the components have been created and instantiated, the console allows to interact with them by inserting and removing metrics, SLOs, and planning strategies.

The command “startMon” allows to start or stop the monitoring and management activity of the MAPE components in a specific GCM component. If no GCM component is specified, it acts over all which are register in the Console domain.

The command “addMetric” allows to instantiate a metric and insert it into the *Monitoring* component (specifically in the *MetricsStore*), so that it can be computed and read later. The command “removeMetric” allows to remove the metric.

The command “addSLO” allows to insert a condition that must be verified at runtime inside the *Analysis* component. Once an *SLO* has been inserted, the command “subscribeSLO” allows to subscribe the *SLO* to a *metric* through the *Monitoring* component. In that case, each update of the *metric* will trigger a checking on the subscribe *SLO*, implementing a *push* mode checking. Alternatively, the command “checkSLO” allows to specify a time after which the *SLO* will be periodically checked regardless of any update to the involved metric, implementing a *pull* mode checking. The command “removeSLO” allows to remove that condition.

The command “addPlanner” allows to insert *planner*s inside the *Planning* component. The command “hookPlanner” correlates a condition with an specific *planner* inserting an entry in the table of the *StrategyManager* as described in Section 7.4 to guide the triggering of a *planner*.

#### 7.6.2.1 Request Path computation

As an example, once the *requestPath* metric, that computes the method described in Section 7.2.3, has been inserted using the command “addMetric”, it can be invoked using the “runMetric” command and obtain the request path of an individual request. The command requires the identification of the initial request, and the result of the invocation is obtained using a tree format shown in Listing 7.7, which is a textual representation of the result obtained in Figure 7.9.

Listing 7.7: Request Path delivered

```
Path from componentZ, for request -434886621
```

```
Request Path from request -434886621
* (-434886621) componentA.p1.p1:                    client: 74828   server: 71210
*    (-835071484) componentB.p2.p1:                 client: 68267   server: 63658
*       (-2072786098) componentD.s.s1:              client: 59723   server: 55995
*          (-1047893631) componentA.u.u1:           client:  7218   server:  3748
*             (-835071483) componentE.u2.u1:        client:   727   server:   425
*          (-1047893630) componentA.v.v1:           client: 44348   server: 40807
*             (-835071482) componentF.v.v1:         client: 35008   server: 29703
*                (-975670994) componentG2.g.gcall:  client: 21518   server:  8013
*                   (-124465018) componentH.w.www:  client:  5268   server:   215
*                (-975670993) componentG3.g.gcall:  client:  7181   server:   129
*                (-975670992) componentG1.g.gcall:  client: 13966   server:   112
>
```

Further examples of the utilization of the Console commands described are presented in Chapter 8.

## 7.7 Summary

In this chapter we have described the details of the implementation of our proposed framework over the GCM/ProActive middleware. The implementation considers the particular semantics of GCM/ProActive component-based applications, and supports the interfaces that we have defined in chapter 5.

Although we present a particular representation for the key objects of the framework, like *metrics*, *SLO*s, *alarms* and *actions*, the implementation has been made to be as flexible as possible while preserving a basic common interaction that allows to obtain the information required and integrate a complete feedback control loop.

We have also provided an external, console-based application that, without being an integral part of our framework, is able to interact with it and modify its configuration according to the monitoring and management needs of the application, while keeping a separation between the functional tasks and the non-functional management of the application.

# 8 Evaluation

## Contents



In this chapter we present the experimentation we have made with the implementation of our framework over the GCM/ProActive middleware. The experimentation is divided in two parts. In Section 8.1 we execute experiments to analyze the overhead incurred by the execution of the MAPE components concurrently with the functional application. In Section 8.2 we describe from a working point of view the use of the framework to insert and modify a set of MAPE components into a concrete application, showing the practicality of our proposition.

## 8.1 Evaluations

### 8.1.1 Execution Overhead

We have executed a sample application with several components, similar to that shown in Figure 7.7. Each execution performs a distributed computation through all the components to build a return message, so that each execution generates a communication that ultimately reaches every other component. We run a repetition of $n$ messages in two versions of the application: one with no MAPE components inserted, and another with a version of each MAPE component inserted in all the membranes. The *Monitoring* component computes metrics related to response time; the *Analysis* component includes an *SLO* that compares the response time in a *push* mode (subscription) upon each update of the *respTime* and, in case it is bigger than 1 second, it sends an alarm to a *planner* component. The *planner* only checks the last value obtained for the *respTime* metric from the *Monitoring* component, but does not generate actions. In order to isolate the execution of the application, in this experiment all the components are deployed in a single node.

| #msgs | Normal (sec) | w/MAPE (sec) | Diff. | %Overhead |
|---|---|---|---|---|
| 1000 | 6.983 | 7.999 | 1.016 | 14.550 |
| 2500 | 17.202 | 19.293 | 2.091 | 12.157 |
| 5000 | 34.392 | 39.179 | 4.787 | 13.920 |
| 10000 | 68.567 | 77.553 | 8.986 | 13.105 |
| 20000 | 140.377 | 158.907 | 18.53 | 13.200 |

Table 8.1: Execution Overhead in non-distributed application

The times obtained for each execution depending on the number of requests, and the overhead obtained for the total execution is shown in Table 8.1.

After some variations, we observe that the overhead incurred stabilizes around 13% of the initial time. Although it seems important, we must highlight that this case represents one of the worst cases of an execution, as the only thing that this application does is to send requests to other components, while little work is done by each individual service. In a more general situation, an application would be expected to do some other activity that only sending requests. However, this experiment allows us to test the behaviour of our framework implementation under a high load and still obtaining correct results.

### 8.1.2 Communication Overhead

In this experiment we use a distributed version of the application, where each component is deployed in a different node. We expect that the biggest part of the time spent by the application be effectively the communication time with each other.

In the following setting, the “Normal” column shows the execution time without any MAPE component inserted, and the “w/MAPE” columns shows the execution with all the MAPE components inserted and running in all the membranes, and in the same node of each functional component. Then, we repeat the execution with the same set of MAPE components as described in the previous example. The results are shown in Table 8.2.

| #msgs | Normal (sec) | w/MAPE (sec) | Diff. | %Overhead |
|---|---|---|---|---|
| 1000 | 29.661 | 33.687 | 4.025 | 13.571 |
| 2500 | 72.199 | 82.184 | 9.985 | 13.829 |
| 5000 | 138.722 | 156.741 | 18.019 | 12.989 |
| 10000 | 271.452 | 314.203 | 42.750 | 15.748 |
| 20000 | 539.263 | 624.274 | 85.011 | 15.764 |

Table 8.2: Execution Overhead in a distributed application with MAPE components executing in the membrane of the functional components

In this case, the overhead reaches around 15% of the “not managed” execution time. This is not a big increment with respect to the previous situation, while the amount of network communication is bigger. Once again, we must mention that this particular experiment reflects a situation where the components spent most of the time sending and receiving requests, which consequently triggers reactions over the application. The node where each component runs must support the execution of both the original functional node, and the activity of additional NF components.

Overall, the insertion of the MAPE components in this implementation implies a bigger load in the execution of the managed component, which is natural. In a worst-case scenario, the overhead incurred does not account for more than 15% of the not managed execution. This measure however, may not be completely accurate, as the actual overhead incurred by the MAPE

components may depend on many additional factors. For one, the specific logic applied to the *metrics* implementation, and to the *planner* strategies may require much more additional processing. Moreover, the *planner* strategy may require (it is not forbidden to) temporarily stop the functional execution of the component if some computation needs to be performed in an isolated way.

Another factor is the supporting implementation. In our case we have conducted our experiments over a distributed environment supported by the GCM/ProActive middleware. This particular implementation profits of asynchronism to allow the concurrent execution of the MAPE components. Each implementation of the framework, however, may profit of their particular characteristics and optimize the implementation.

## 8.2 Example: Tourism Planner

In this section we aim to illustrate the functionality that can be achieved with our framework. We present an example service-based application, initially designed without monitoring and management capabilities. We show how, through the use of our framework we can add monitoring and management features to the application and introduce some autonomic capabilities.

### 8.2.1 Context: A Tourism Planner Application

The example considers a tourism office who has composed a smart service to assist visitors who request information from the city and provides suggestions and planning of touristic activities. The application considers that some of the services can be provided and hosted locally, while others are remotely provided by third party applications, some freely available, and some that require a payment.

The local services include a database of seasonal activities maintained by the tourism office; a composer engine that is able to produce a document in PDF format; and a printer service. The externally provided services comprise a weather prediction service, a banking service that allows to process payments, a mapping service to compute itineraries and provides location services, and a set of attraction services, where each one of them represents an entity that provides tickets to events or manifestations. We assume for simplicity that there exists a common interface through which these attractions can be contacted and accessed, though if it was not case an intermediate adaptor component can be used.

The application receives requests from a *front-end* component that allows interaction with the user. The user selects a range of dates and other preferences, and the *Tourism Service* provides a list of available events and attractions using both the external attraction providers, or the database of local events. The proposition can also be guided according to the user preferences, f.e., by price range, location or weather conditions. Once the user has made up his selection, the *Tourism Service* allows him to pay for any tickets, if needed, using the *Bank Service* and later it generates a PDF document including the information of his activities selected, complemented with a weather report and maps showing the itineraries. Optionally the document may be sent via email.

The overall design of the application is shown in Figure 8.1. The composition relationship intends to illustrate the fact that some services are locally developed and, consequently, belong to the same composite *Tourism Service*, while the other services are externally provided and are accessed through references of the composite.

The application is initially designed without any activity related to monitoring or management. However, in order to be able to insert some MAPE components later, it is necessary that the required interfaces be previously declared. In the context of our implementation, this is achieved by using the *createMMtype()* method as mentioned in Section 7.6. The equivalent design using the GCM notation is shown in Figure 8.2 for the *Tourism Service* composite. Once a component has been created using the NF Type provided by the *createMMType()* method, the

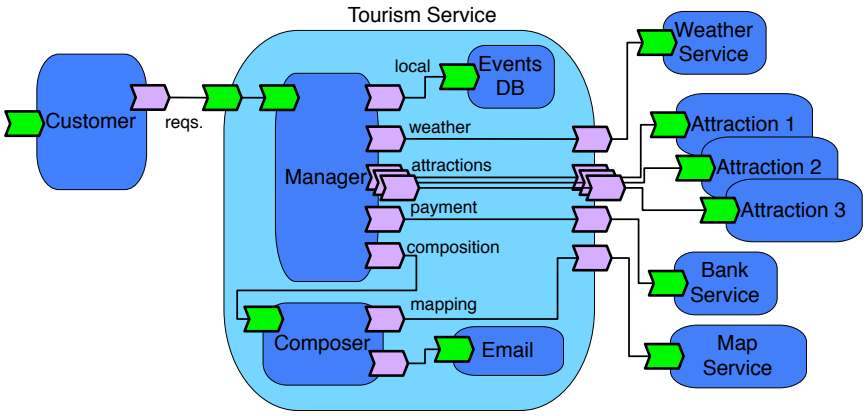


Figure 8.1: Scenario. SCA description of the application for tourism planning.

components are created with NF Type that can support the monitoring and management features, however they are not yet bound to any concrete component as the membrane is initially empty.

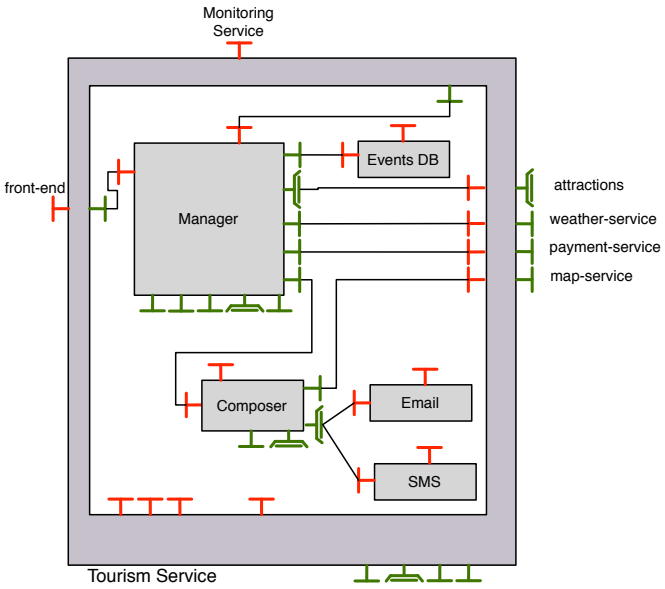


Figure 8.2: GCM description of the Tourism Service composite. NF Interfaces are available but no NF Component is in the membrane

In the following section we describe steps oriented to provide monitoring and management features to the application, and compose some autonomic behaviours.

### 8.2.2 Monitoring the response time of the composite

Initially, the application has no activity related to monitoring or management. However, some users report long waiting times for obtaining responses, which ultimately lead them to leave the site. An initial step to solve this issue is to insert monitoring components in all the locally hosted components and determine where is the time spent. Using the Console application, a basic *Monitoring* component is attached to the *Tourism Service* composite and all its internal subcomponents.

Listing 8.1: Adding existing components to the Console

```
+-------------------------------------------------------------------------+
|     REconfigurable Monitoring and Management of Services -- Console     |
+-------------------------------------------------------------------------+

> a rmi://cobreloa.inria.fr:6699/EntryPoint
Looked-up [TourismService] @ [rmi://cobreloa.inria.fr:6699/EntryPoint]

> use TourismService
Current: TourismService

> disc
Discovering from component TourismService
... ... ...
... ... ...

> ls
   TourismService
   Manager
   EventsDB
   Composer
   Email
   SMS
   WeatherService
   BankingService
   Atracction1
   Attraction2
   Attraction3
   MappingService

>
```

Listing 8.1 shows the initial steps for adding an application to the domain of the Console. The application registers a reference to one of its interfaces in a defined RMI registry, in this case under the name `EntryPoint` and the Console uses the command 'a' (add) to lookup for a GCM component in that location and adds it to the list of managed components. Once selected with the command 'use', the manager can use the command 'disc' (discover) to find all the components bound to the *TourismService* and add them to the set of managed components. The command 'disc' uses introspection to find out references on all the reachable components, this is, the set of components bound to *TourismService* (both internal and external), and those can are bound to them. This is a necessary step as components may be distributed and there is no single component that contains the information of all the application.

Listing 8.2: Inserting Monitoring Components

```
> addMon
Adding monitoring components to TourismService ...
Adding monitoring components to Manager (internal) ...
Adding monitoring components to EventsDB (internal) ...
... ... ...
Adding monitoring components to MappingService (external) ...
Enabling monitoring bindings on component Tourism Service ...
Enabling monitoring bindings on component Manager ...
Enabling monitoring bindings on component EventsDB ...
... ... ...
Enabling monitoring bindings on component MappingService

> startMon
Starting monitoring ...
```

```
Monitoring Components started.
>
```

The insertion is realized through the command 'addMon' (add Monitoring), as shown in Listing 8.2. The 'addMon' command inserts the generic *Monitoring* component provided by our framework in the membrane of all the managed components and performs the NF bindings needed to connect the *Monitoring* components of the related components. Finally the command 'startMon' initiates the work of the *Monitoring* components.

Figure 8.3 shows the *Tourism Service* composite once the *Monitoring* has been inserted in its membrane, and in each one of its subcomponents. The NF bindings are shown as solid lines inside the membrane, and as dashed lines in the functional part.

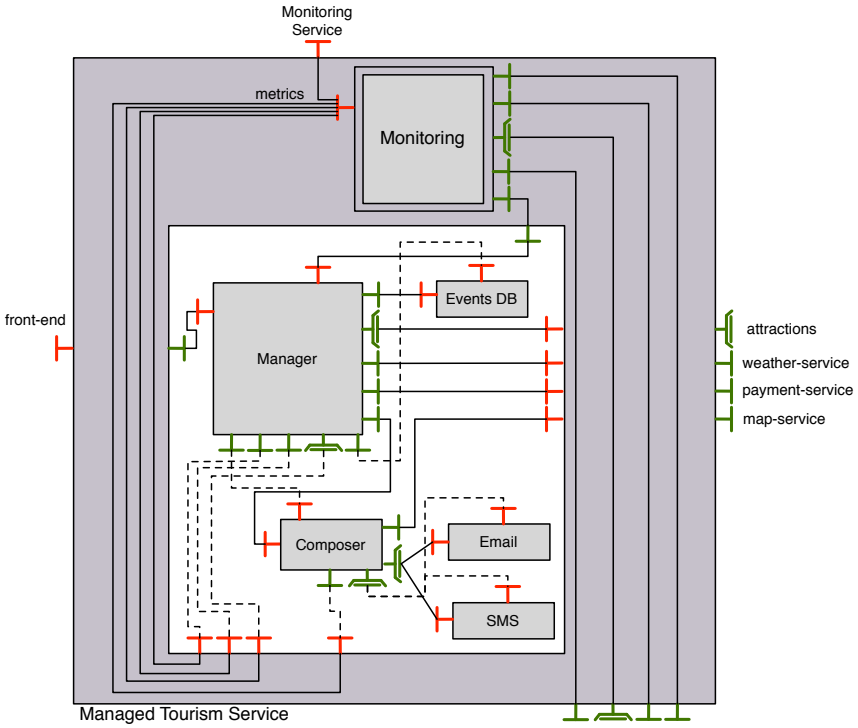


Figure 8.3: GCM description of the Tourism Service composite, once the *Monitoring* component has been inserted in the membrane of all components, and its NF Interfaces are bound

The *Monitoring* components are started, however no *metric* is being computed at this moment. In order to obtain information about the response time of the components, we introduce the metrics *respTime* and *avgRespTime*; the first metric computes the response time found in a specific request, and the second computes the average response time of all request in a specific interface.

Listing 8.3: Inserting metrics in Monitoring components

```
> addMetric TourismService respTime respTimeTS
Metric respTimeTS (type: respTime) added to TourismService

> addMetric TourismService avgItfRespTime avgItfTimeTS "reqs"
Metric avgItfTimeTS (type: avgItfRespTime) added to TourismService

> runMetric TourismService respTimeTS 1131284372
TourismService.avgInc1 = 56245

> runMetric TourismService avgItfTimeTS
```

```
TourismService.avgInc1 = 26351.6

> lsMetric TourismService
Metrics in component [TourismService]
   avgItfTimeTS (type: avgItfRespTime)
   respTimeTS (type: respTime)

>
```

Listing 8.3 shows the steps for inserting the mentioned metrics in the *TourismService* component. The command "addMetric" allows to insert a metric from a library of available metrics, specifying the type of the metric, an instantiation name, and a set of optional constructor parameters. In this case, the metric *respTime* is instantiated as *respTimeTS* and does not require constructor parameters. The metric *avgItfRespTime* is instantiated as *avgItfTimeTS*, parameterized with the interface named "reqs". The instantiation name is required as to differentiate the inserted metric from other that may have the same type, but different initial parameters, f.e. the *avgItfRespTime* metric for measuring the time of another interface. Similar steps should be repeated for each component where the metrics are required.

The command "runMetric" in Listing 8.3 executes the indicated metric and allows to specify execution parameters for it. In the case of *respTimeTS* the indication is the *rID* of a specific request. The list of requests served by a component and, in particular, their *rIDs* can be obtained by using the "l" (logs) command on a component.

By using the Console, the manager of the application can check if the average response time of the application is inside a reasonable value or not. If not, he can use the *respTime* metric to obtain the response time of a particular request, however this only gives information about the total time and does not provide much insight to find a component that can be mentioned as "responsible" for the time spent. It is also possible for the manager to ask the *respTime* metric on other components, however it would require for him to analyze the logs of each component to find out the appropriate requests to ask for.

A more convenient solution can be obtained by inserting the *requestPath* metric in each component, as shown in Listing 8.4. From here, the manager can see a decomposition of time spent by a request and can identify the component where the longest part of time was spent.

Listing 8.4: Request Path computation

```
1  > addMetric TourismService requestPath rp
2  Metric rp (type: requestPath) added to TourismService
3  ...
4  > addMetric MappingService requestPath rp
5  Metric rp (type: requestPath) added to MappingService
6
7  > runMetric TourismService rp 1131284383
8  Path from TourismService, for request 1131284383
9  Request Path from request 1131284383
10 * (1131284383) TourismService.reqs.buildDoc:                    client: 7943 server: 7646
11 *    (-516789329) Manager.events.getEvent:                      client:  410 server:  398
12 *    (-516789328) Manager.weather.getWeather:                   client: 2224 server: 2118
13 *       (1131284384) TourismService.weather.getWeather:         client: 2011 server: 1841
14 *    (-516789327) Manager.attraction3.getTicketData:            client: 3019 server: 2867
15 *       (1131284385) TourismService.attraction3.getTicketData: client: 2860 server:  702
16 *    (-516789326) Manager.composer.buildDoc:                    client: 5066 server: 5002
17 *       (1278875256) Composer.mapping.getLocation:              client: 3200 server: 3109
18 *          (1131284385) TourismService.mapping.getLocation:     client: 3006 server: 2955
19 *       (1278875257) Composer.email.send:                       client: 1434 server: 1137
20 >
```

The tree obtained in Listing 8.4 (all time are given in *msec.*) shows that the complete invocation took almos 8 *secs.* (line 10: client perception of the invocation of the method *buildDoc* on the *reqs* interface of component *TourismService*. If we analyze the first level of the invocations generated by this call (line 11, 12, 14 and 16), we can see that the invocation on the *Composer*

component (line 16) took the larger slice of time respect to the other invocations. If we look deeper in the invocations made by *Composer* (lines 17 and 19), it is possible to see that the call to the *Mapping* component was the longest of them. However, another inspection can show, thanks to the fact that we have collected the time from both the client side and from the server side, that the invocation to *Attraction3* (line 15) took 2.860 *secs*. from the client side, but the service reported to have taken only 0.702 *secs*. to obtain the response, which could point to a latency of almost 2 *secs* in that invocation, which is high compared to the latencies obtained in the other invocations to external services (lines 13 and 18).

### 8.2.3 Automating SLO Monitoring

In the current state, a manager is able to analyze the response time of an application and obtain some feedback. However, this still requires a human administrator to query the needed information from the application.

The application can be improved by inserting an automated analysis. The next step inserts the following additional components:

- An *Analysis* component that includes an *SLO* that checks the *avgRespTime* metric and raises an *alarm* when this reaches a certain threshold.
- A *Planning* component that includes a *strategy* that sends an email notification to the manager of the application.

The first step is to insert the *Analysis* and *Planning* components. As this behaviour is only required in the *TourismService* component, it is not necessary to attach them to the other components, as shown in Listing 8.5.

Listing 8.5: Adding Analysis and Planning components

```
> addAnalysis TourismService
Adding analysis components to TourismService ...

> addPlanning TourismService
Adding planning components to TourismService ...
```

Secondly, it is necessary to add the *SLO* description and a very simple *planner* that upon an *alarm*, notifies a human manager by sending an email.

Listing 8.6: Inserting SLO and Planner elements

```
> addSLO TourismService sloART avgItfTimeTS lowerThan 20
Adding SLO <avgItfTimeTS, lowerThan, 20> to TourismService ...

> subscribeSLO TourismServce sloART avgItfTimeTS
Subscribing SLO sloART to metric avgItfTimeTS ...

> addPlanner TourismService planningEmail PlanningEmail
Adding planning component planningEmail (class: PlanningEmail) to TourismService

> hookPlanner TourismService avgItfTimeTS FAULT planningEmail
Adding entry <avgItfTimeTS, FAULT, planningEmail>
```

Listing 8.6 shows the different steps mentioned. An *SLO* limiting the maximum average response time acceptable to 20 sec. is added, and subscribed to the metric *avgItfTimeTS* in order to be checked each time that the metric *avgItfTimeTS* is updated, forcing a *push* checking mode. The metric is updated, in the *Monitoring* component, each time that a new request is served, according to the *metric* implementation. In the *Planning* component, a new component that implements a strategy is inserted. What is needed is the class that implements the logic of the planning strategy, and in this case it is a very simple one that sends an email to a predefined

email address. Finally, an entry is inserted in the decision table of the *Planning* component that defines that upon a faulting *SLO* regarding the metric *avgItfTimeTS*, the planner called "planningEmail" should be invoked.

With this setting a simple autonomic response is attached to the *TourismService* that relies upon a single check to decide on an action.

At a later step, we can add a checking that gives more information about the fault, by computing a request path of the last request served and included in the report. The steps require to remove the previous *planner* component and replace it for a new one that implements the new logic, and they are shown in Listing 8.7.

Listing 8.7: Replacing the Planner component

```
> removePlanning TourismService planningEmail
Removing planning component planningEmail from TourismService...

> addPlanner TourismService planningEmailReqPath planningEmailReqPath
Adding planning component planningEmailReqPath
(class: PlanningEmailReqPath) to TourismService

> hookPlanner TourismService avgItfTimeTS FAULT planningEmailReqPath
Adding entry <avgItfTimeTS, FAULT, planningEmailReqPath>
```

An important step is to add the entry in the association table of the *Planning* component, pointing to the new *planner*. We did not explicitly removed the old entry, however the *Strategy Manager* evaluates the entries in the order they are inserted and, if one fails, in this because the destination *planner* is not available anymore, then the next entry is tried.

### 8.2.4 Autonomically replacing a service

The application can be improved by automating the analysis. So far, the control loop is not complete as the last step is just a notification to a human administrator who should decide on an action.

The next improvement involves an automated way to deal with the problem. The *planner* component is replaced with another one that is capable of executing an action over the application.

The first step is to replace the existing *planner* component by another that executes the following strategy:

- Get the *requestPath* metric from the *Monitoring* component.
- Analyze the obtained tree and identify the component $C_1$ where the biggest share of time is spent.
- Use a discovery service to obtain a replacement component $C_2$ that provides the same functionality of $C_1$, with a compatible interface.
- Send as output the sentence "replace($C_1$, $C_2$)"

This description makes some assumptions. First, it assumes that discovery service is capable of finding an appropriate replacement component in a relative short time. For our case, we have provided a set of possible replacement components for each interface. Also, there is no guarantee that a discovery service is capable of finding a component with compatible interface. Most of the time different services with the same functionality are available but with different interfaces. The problem of automatically adapting one interface to be used with another existing one is major research topic itself, not addressed in this work. Second, there is also no guarantee that the service found allows to solve the response time problem. In practice, strategies like those mentioned in Section 3.1.3 consider the probable benefit of a service before selecting one

specific component. In our case, this would require that the *Planning* component be capable of contacting the *Monitoring* interface of the replacement component and obtain previously stored (historical) information that allows to estimate that the performance of this component will be better than the current one. Finally, even if a component with better response time is found, this does not guarantee that the problem will be solved, neither that this will be the optimal solution. It is, indeed, possible that the replacement of another can have a better impact on the overall performance. However, the objective of this example is to illustrate that a new *planning* component that considers the current situation of the application can be inserted and carry on a modification action.

Next step is to properly interpret the sentence generated by the *planner* component. In our implementation we use the PAGCMScript language to execute actions over the GCM/ProActive application. We insert an *Execution* component in each one of the managed components, and bind them to the corresponding *Planning* component, and between them in a similar way to the *Monitoring* components.

The *PAGCMScript* engine embedded in the *Execution* component is capable of interpreting the sentence sent by the *Planning* component. In fact, the command "replace($C_1$,$C_2$)" is an action defined in PAGCMScript that is capable of propagating the action until the component over which this action must be applied is found, like shown also in Section 7.5.2. The sequence of commands to replace the *planner* component and insert the *Execution* components is shown in Listing 8.8.

Listing 8.8: Planner update, and insertion of Execution component

```
> removePlanning TourismService planningEmailReqPath
Removing planning component planningEmailReqPath from TourismService...

> addPlanner TourismService planningReplacer planningEmailReplacer
Adding planning component planningReplacer
(class: PlanningReplacer) to TourismService

> hookPlanner TourismService avgItfTimeTS FAULT planningEmailReplacer
Adding entry <avgItfTimeTS, FAULT, planningReplacer>

> addExec
Adding execution components to TourismService ...
Adding execution components to Manager ...
Adding execution components to EventsDB ...
... ... ...
Adding execution components to MappingService ...
Enabling execution bindings on component Tourism Service ...
Enabling execution bindings on component Manager ...
Enabling execution bindings on component EventsDB ...
... ... ...
Enabling execution bindings on component MappingService
```

The flow of the autonomic action is shown in Figure 8.4. In the figure, the bindings related to the *Execution* components are shown, as solid line inside the membrane, and as dashed lines in the functional part. The bindings related to *Monitoring* are hidden for clarity, but they are present in the application as shown in Figure 8.3.

This example shows a complete autonomic control loop that considers the global state of the application to determine a response, and is able to propagate it at the appropriate component. It is worth to mention that this global autonomic loop considers information obtained from the *Monitoring* components of each component of the application, that is used by the *Analysis* and *Planning* components of the *TourismService* composite, where the decision is taken. Both *Analysis* and *Planning* components are inserted only in the *TourismService* composite. Once a decision is made, the action is propagated to the *Execution* components of each functional component.

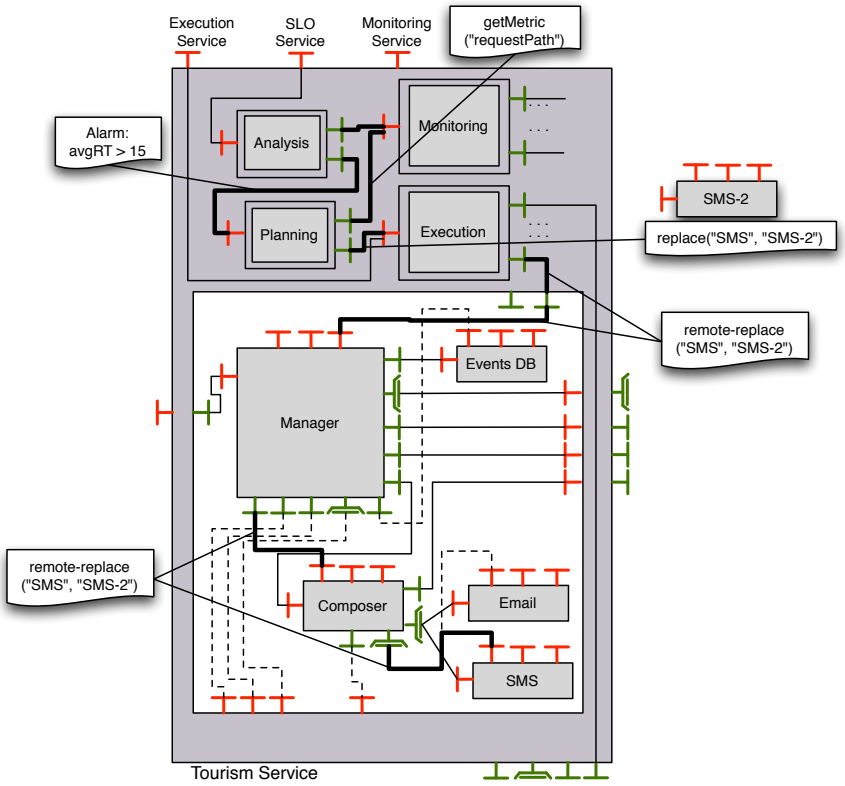


Figure 8.4: Flow of an autonomic action in the *TourismService* component. Some interfaces and bindings not related to the flow are hidden.

### 8.2.5 Inserting local components

One feature of the framework is to allow the insertion of local control loops according to the needs of the application, which not necessarily require to involve all the elements of the composition.

A modification on the application is inserted and a third component to send the final document is added to the *Composer* component. In this case a *Twitter* component that publishes a short link about the itinerary that has been created. As with the rest of the components, the *Twitter* component must be created with the necessary NF interface using the *createMMType()* method. However, when inserting the *Twitter* component into the *TourismService* composite, and binding it with the *Composer* component, the NF bindings of the *Composer* are not automatically updated, as this is not a concern of the composer of the functional part.

Suppose that the sender components (*Email*, *SMS-2*, and *Twitter*) have failures or timeouts. A particular autonomic task can be inserted related to this three components for stopping using one of them once a certain number of failures have been detected.

For this, a *Monitoring* component must be inserted in the new component, and the required NF bindings must be made. From the Console, this is possible by invoking the “addMon” command both in the *Composer* component and in the new inserted *Twitter* component.

Listing 8.9: Inserting Monitoring on new components

```
> addMon Twitter
Adding monitoring components to Twitter ...
> addMon Composer
Adding monitoring components to Composer ...
Enabling monitoring bindings on component Composer ...
> startMon Twitter
Starting monitoring ...
```

```
Monitoring Components started.
>
```

Listing 8.9 shows the steps for inserting the *Monitoring* component on the *Twitter* and for updating the NF bindings. In fact, the bindings for the *Email* and for the *SMS-2* components already exist. The command "addMon" over the *Composer* component updates the bindings if required and creates the new one (for the *Twitter* component).

For accounting the number of failures, a metric called *failCounter* is inserted in the three *Email*, *SMS-2* and *Twitter* components. This metric increments a counter each time that the component receives an error code while performing its task. In the *Composer* component, another metric is inserted called *failAggregator*. This metric uses the *Monitoring* interfaces of the three connected components to subscribe to the *failCounter* metric, and maintains a counter with the maximum number of failures among the three of them.

In order to insert an autonomic action, an *Analysis* component and a *Planning* component are added to the *Composer* component. The *Analysis* component includes an *SLO* that describes $\langle failAggregator, <, 5 \rangle$. This way, the *Analysis* component will raise an alarm when the any of the three components reports at least 5 failures. The *Planning* component requires an associated *planner* that asks for the value of the *failCounter* of each component and determines the component $C_s$ with more failures, and issues a sentence "disable($C_s$)". For executing this action, and *Execution* component is attached to the *Composer* component. The PAGCMScript engine expands the sentence to an unbind action for the indicated component. Figure 8.5 shows the control loop inserted in this subset of components, which works concurrently and independently from the more global control loop described in the previous section.

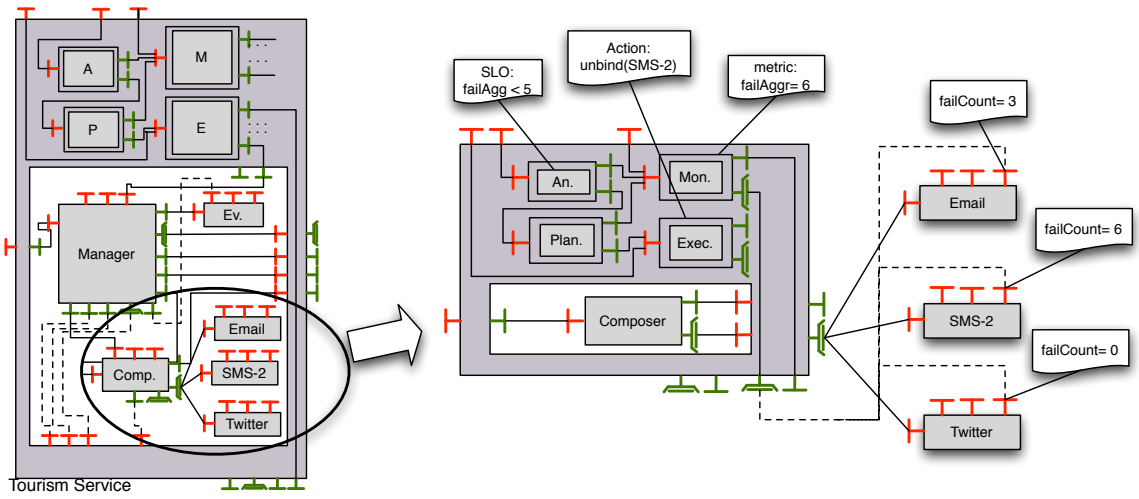


Figure 8.5: Autonomic control loop related to the number of failures in a subset of components. This loop is inserted only for a subset of components.

### 8.2.6 Inserting additional SLOs

The condition inserted in the *Tourism Service* component is not the only one that can be. One of the objectives is to able to handle several concurrent conditions, provided that they do not generate conflicting situations.

Suppose the manager of the application needs to limit the maximum cost of maintaining its application. As some services are externally provided, some of them may require a payment, and some others may be available for free. Among those that require a payment, they may have different models for charging, so a different metric to compute the cost of using each service must be provided.

In this case, assumed the different models of each service, we introduce a metric named *cost* for each service. The implementation of each metric is different in each component (they are made from different classes that extends *Metric<Double>*), but they share the same name.

- The *WeatherService* features a monthly subscription that allows an unlimited number of requests. In this case, the metric *cost* counts the number of requests served during the month and returns an estimated cost per request, which will be used to compute the final cost for the end-user.
- The *PaymentService* charges an amount proportional to each transaction made. We insert a metric *cost* that accumulates the cost incurred by each request and returns an average cost according to the number of requests processed.
- The *Attraction2* charges a fixed amount per request. We insert a metric *cost* that returns this fixed amount.
- The rest of the services are free. Their metric *cost* belong to the same class, and return 0 on each call.

Given these metrics the computation of the metric *cost* in the *Monitoring* component of the *TourismService* can be made as shown in Listing 8.10. In this case, it is possible to see that, though the metrics for each external service are indeed different, by inserting them through our framework we provide a uniform way to access them. If the cost model of one of the external service would change, then it would be necessary to change only the associated metric in the involved component, while the cost metric in the *MonitoringService* component remains the same.

```
public class CostMetric extends Metric<Double> {
      ...
      public Double calculate(final Object[] params) {
         double cost = 0.0;
         Interface[] externalMonitors = GCM.getClientInterfaces(
             metricStoreRef);
         for(Interface external : externalMonitors) {
            MonitoringService mon = (MonitoringService) external;
            cost = cost + mon.getMetric("cost");
         }
         return cost;
      }
}
```

Listing 8.10: Implementation of the *cost* metric for the TourismService component

Using this metric, the next step is to insert an *SLO* that periodically checks if the cost per request reaches a threshold: $\langle cost, <, 2.0 \rangle$. This *SLO* will be associated to a *planner* and *Execution* components that attempt to replace the component with the higher cost by another found through a discovery service.

Listing 8.11: Adding a new SLO

```
> addSLO TourismService sloCost cost lowerThan 2.0
Adding SLO <cost, lowerThan, 2.0> to TourismService
> checkSLO TourismService sloCost 1m
Enabling periodic checking of SLO sloCost at 1 minute ...
> addPlanner TourismService planningCost PlanningCost
Adding planning component planningCost (class: PlanningCost) to TourismService
> hookPlanner TourismService cost FAULT planningCost
Adding entry <cost, FAULT, planningCost>
>
```

The new *SLO* is inserted and associated to a *planner* component called *planningCost* as shown in Listing 8.11. The *planningCost* component does a similar work as metric *CostMetric* (Listing 8.10), but it also identifies the component with the higher cost and uses the discovery service to find a replacement. Figure 8.6 shows in bold lines the requests made by the *Monitoring* component of *TourismComponent* in order to compute *cost* and generate a replacement action.

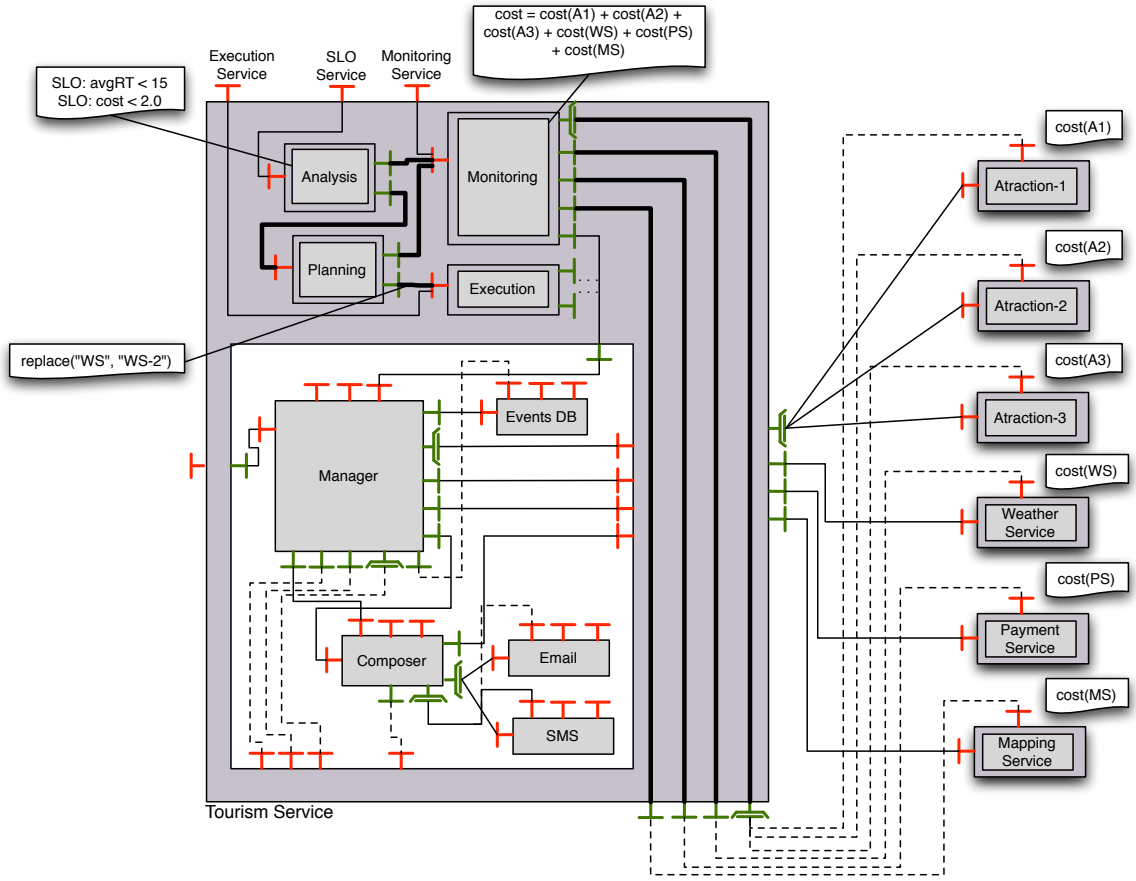


Figure 8.6: Autonomic control loop related to the number of failures in a subset of components. This loop is inserted only for a subset of components.

Once again as mentioned in the previous case, the proposed strategy may not be the most convenient. However, it shows that two different conditions, and consequently two different control loop managing different actions can coexist. What has not been mentioned is how to deal with conflicting actions. Indeed, it is possible that once the replacement action be executed and the *WeatherService* is replaced by *WeatherService-2*, the new component has a response time that violates the *SLO* $\langle avgRT, <, 15 \rangle$, as the execution of the *planner* component *planningCost* is different and independent to that of the coexisting *plannerReplacer*. It is one of the features of the component based approach that both *planner* components can coexist and possibly execute concurrently, completely unaware of each other. However, it would be a responsibility of the programmers of the *planner*s to check that this concurrent execution is safe. A simple solution to avoid this situation can be to avoid concurrent execution and do not invoke a *planner* component when another is executing. This kind of behaviour should be inserted by the *StrategyManager* (inside the *Planning* component, recall Section 7.4. Although it is a certain possibility, in our experiments we have chosen not to do like this in order to show that concurrent execution is possible, and illustrate the capabilities of the framework. A better solution would be to automate the decision about which executions of *planner* components are possible. One alternative is to rely on additional declaration from the programmer side to help the *Strategy Manager* to

automatically decide if it allows a concurrent execution or not. However, we think that this decision is hard to generalize and deserves a bigger amount of research to find an appropriate means to represent this information without incurring in a big burden for the programmer.

### 8.2.7 Providing a Self-healing response on a service

The examples presented deal mostly with issues related to the implementation of the service. However, as mentioned in Chapter 4, a service can be affected in several levels, in particular in the infrastructure level.

Consider that the manager of the application has the services that are internal to the *TourismService* composite locally hosted on an *on-premises* infrastructure. Upon a high demand, the local infrastructure may be insufficient to handle the load and, consequently, the manager decides to host the components in an external IaaS *cloud*. For the manager to know details about the hosting infrastructure, we insert a metric called *nodeInfo* in each of these components. The metric *nodeInfo* collects information about the hosting infrastructure (CPU type, host name, IP address, ...) in a *NodeInfo* object. This metric is not attached to any event in particular, as it is not related to a functional event from the component, so the data is obtained in a *pull* mode. The way to obtain this information, this is, the implementation of the metric, depends largely on the infrastructure support. In our implementation, the GCM/ProActive middleware provides two JMX MBeans called *ProActiveRuntimeWrapperMBean* and *NodeMBean* that allow to obtain information about deployment details, the hosting virtual machine, the virtual node used, and the infrastructure characteristics.

An event that can affect the behaviour of the application is the unavailability of some component due, for example, to a network problem or some event affecting the remote infrastructure. A simple means to obtain the availability of the inner components is to insert an *availability* metric in the *Monitoring* components. Each component that has a client binding to another one implements this *availability* metric that checks periodically the availability of the bound component using heartbeats. Using the heartbeats and the uptime, it computes an *availability* percentage.

Given this metric, an *SLO* that restricts this limit can be inserted in each one of these components to ensure $\langle availability, >, 0.99 \rangle$. This *SLO* is associated to a planning strategy that intends to deploy a new copy of the component in another prefixed infrastructure. The *planner* component only requires to generate a PAGCMScript sentence indicating "migrate($C$, *dest*)", where $C$ is the component to be migrated, and *dest* indicates a predefined hosting infrastructure.

With this setting, each of the subcomponent of *TourismService* implements a local autonomic *self-healing* control loop where they check the availability of a service that they use and, upon an availability problem they trigger a migration to another node.

### 8.2.8 Providing a Self-optimizing response on a service

When considering a set of nodes available for executing services, a *self-optimizing* behaviour can be implemented by considering the performance of the nodes. Instead of relying only in response time, this example attempts to use infrastructure information to guide a better utilization of nodes.

The criteria to use will be the CPU load, and the memory available on each node. For measuring this, two metrics are added to each component: *cpuLoad* and *freeMem*, to measure CPU load and available memory, respectively. The implementation of these metrics depends highly on the details of the machine used, however for our implementation we rely on the information provided by the GCM/ProActive middleware.

The *Analysis* component will use the same *SLO* previously inserted, which is related to the *avgItfTimeTS* metric. The modification will be at the level of the associated *planner* component. For this, the existing *plannerReplacer* component is removed and replaced by another *planner* called *plannerMigration*, associated to the same existent *SLO* $\langle avgItfTimeTS, <, 20 \rangle$.

The *plannerMigration* component uses the *Monitoring* component to find the values of the metrics *cpuLoad* and *freeMem* on each subcomponent and determines the one that is more loaded. In this case, the criteria is to find the component whose hosting node has the biggest *cpuLoad* value and then the lowest *freeMem*. Having found this node, the *plannerMigration* locates a new available node (a new instance from a cloud environment) and migrates one component residing in that node to the new obtained instance. The output is, thus, a sentence "migrate($C_m$, *newInst*)", where the component $C_m$, randomly selected among the component running in the loaded node, is the one chosen to migrate to the new node indicated by *newInst*.

As in the previous example, we have made some assumptions that facilitate the overall implementation of the *planner* components, but allowing to show that a section of a running autonomic control loop can be modified in a transparent form for the rest of the components. However, transparency does not mean that any *planner* component can be replaced by any other. If no care is taken, it is possible that the new instance obtained require a cost that may increase the overall cost of the application, and possibly cross the threshold defined by the metric that measures the cost per request: $\langle cost, <, 2.0 \rangle$. This simple action may trigger another autonomic response handled by the *planningCost* component, which may increase the value of the *avgRespTime* and triggering another autonomic response. Easily it may happen that the application enters in a cycle of autonomic response triggered by each other. The analysis here is that the design of autonomic response is not an easy task and if care is not taken, it is not hard to get into a *livelock* situation like the one described. Although it is not a part of this research work to provide a solution to those situations (which is by itself a broader research area), we think that the separation of activities that we provide through our framework allows a more flexible development of autonomic actions, and facilitates their integration.

## 8.3 Summary

In this chapter we have presented our evaluation of the performance and feasibility of the implementation that we provide of our framework, in the context of the GCM/ProActive middleware. We have divided the evaluation in two steps: (1) we perform experiments that show that the execution of the NF MAPE components in the membrane of a GCM/ProActive application that delivers a high number of requests, and consequently, triggers a high set of events that must be analyzed by the MAPE components, does not harm the overall execution of the functional part of the application, even when the overhead is certainly measurable. Indeed, our implementation relies heavily on the asynchrony and in the separation of activities in each GCM component in order to execute monitoring and management task in parallel to functional execution of the application. However, particular middleware support, or other SCA compliant platform may introduce particular way to optimize the implementation. (2) We illustrate through a concrete example, how the MAPE components can be inserted and replaced inside a running application in order to better deal with problems that may arise, and allowing the application to adapt to environmental conditions by the way of inserting autonomic control loops where they are required, and by modifying the composition of this loop.

# 9 Conclusion and Perspectives

## Contents



This chapter describes the conclusions obtained during our work, and presents the perspectives and research paths that we have identified.

## 9.1 Contribution

In this thesis we have presented a framework intended to improve at runtime the adaptability of service-oriented applications that are built around a component-based approach. The framework proposed is itself designed around a component oriented approach that encapsulates the different phases of the MAPE autonomic control loop, *Monitoring*, *Analysis*, *Planning*, and *Execution* and binds them through predefined interfaces. Our proposition allows to connect them in a logical way at runtime and, in this way, to provide an implementation of the monitoring and management tasks over a service-based application that can be reconfigured at runtime enabling the introduction of autonomic behaviour to the application.

We have acknowledged the difficulty of providing effective autonomic tasks. In fact, commonly autonomic tasks are proposed as global, general goals over an application, however when they need to be applied, they must be subdivided in smaller more simple tasks that can be executed in the appropriate place of the application, and interact in a meaningful way to achieve the overall goal. This subdivision is not evident at all, and it is not a main concern of our work to solve this problem; however we believe that our proposition helps to provide an implementation of autonomic behaviour in several ways:

- By providing a separation of phases of the autonomic control loop we allow to develop the logic for computing metrics and evaluating actions in separate ways. This facilitates the implementation of adaptation strategies, by means of the *planner* components, as the developer does not need to provide the access to the specific metrics that it may require, because this is encapsulated in the *Monitoring* component, although he/she needs to know the appropriate metric name to ask for it. The *Monitoring* component implements the specific logic for computing the metrics, which may depend on the possibilities available by the supporting infrastructure.
- By separating the phases of the MAPE control loop, we do not force to have complete closed loops over all the components. This way, a component may only have, f.e., *Monitoring* and *Execution* components, and avoid the insertion of the other MAPE components if they are not locally needed.
- By attaching the control loop to each component involved, we allow to execute actions and take decisions close to the specific component. The advantage is that the analysis and

planning phases (and consequently the additional components inserted) can react in a more local way, and does not need to propagate the collected information to a global centralized manager.

- By allowing the insertion of multiple *metrics*, *SLO*s, or *planning* strategies in the corresponding components, we allow the insertion of several MAPE loops that can execute in parallel, possibly including the same set of components, and consequently allowing independent developments, and more efficient executions. Nevertheless, this advantage must be taken with care, as the possibility of having autonomic tasks that execute in parallel over non-empty intersections of components, brings the risk of the occurrence of contradictory or inconsistent actions over the application.
- By inserting the required MAPE components at runtime instead of forcing their creation from the initial design time, and allow to plug them in, or plug them out as needed, the framework can be less intrusive in the addition of monitoring and management tasks (1) because we can insert the MAPE components only in the components that need them, and (2) because we can remove them if they are not needed anymore.

We have provided an implementation of our framework over the GCM/ProActive middleware, profiting of the distributed and reconfigurable component model that it supports. Indeed, the reconfiguration capability turns to be very important for being able to insert and remove components of our framework at runtime. At the same time the ability to separate the definitions for the functional content and for the non-functional part turned as a convenient way to improve the separation of concerns in our approach, as the developer of the functional content does not need to take care of the implementation of the management tasks. Most SCA runtime implementations do not allow a dynamic reconfiguration of an application (FraSCAti being the most notable exception), nor do they allow the addition of monitoring and management tasks at runtime. We believe that this thesis shows the utility of having such kind of approach to improve separation of concerns, and facilitate the development of more adaptable service compositions.

## 9.2 Perspectives

The possibilities of further research after this work are far from being over. The implementation of autonomic strategies itself is an active research area. We highlight the following points as promising research directions for future work.

- We think this framework can provide a good support for the development of collaborating strategies, and provide a higher reasoning level where the planning that cannot deliver a solution in a certain level can be processed by a planner in a higher level of a hierarchical setting. In fact, we have shown the possibility of collaborating *Monitoring* components that use information stored on other *Monitoring* components to compute the values of some metrics, and the set of interconnected *Execution* components that can drive the execution of actions to specific components in the application. In the same sense the *Planning* component can form a “Planning backbone” where the decision can, in a first step, be taken in the local level and, in a second step if the situation cannot be solved locally, forwarded to a higher level *planner*. Another interesting alternative is to avoid a greedy approach like the one just mentioned, and instead of executing immediately the decision locally taken, submit it to the higher level planner where it can be compared with other solutions.
- A problem that we have mentioned, but not addressed, is the safe execution of the actions decided by the *Planning* component. In our experiments we have used simple actions, designed in a way that we are certain that they can be executed in parallel, without rendenring the system in an inconsistent state. However, an improvement would be to have a mechanical way to determine if two or more actions can be concurrently executed over the system, with little effort from the side of the programmer. Indeed, a simple criterion can

consider the spatial dimension: two actions defined to execute in different, unrelated sections of a composition can be executed at the same time; however it is not evident to deduce the consequences that an action can have, even if it is executed from separate components and how they can impact the composition of the application. Nevertheless, we consider that the structured communication that we enforce through the framework can be a good support for experimentation in this area.

- So far, our implementation requires the existence of appropriate interfaces, called NF interfaces, as connection points to communicate the MAPE components inserted inside the membrane with those inserted in the membrane of other components. This restriction imposes that the functional component be instantiated with the appropriate interfaces. However, given that the construction of the membrane is handled by the middleware, it would be more convenient to allow also the insertion and removal of the needed interfaces, eliminating the need to enforce a specific method (*createMMtype()*) to instantiate components that are "manageable" and making more transparent the addition of autonomic behaviour for the developer of the functional task. This dynamic change of the NF type of the component (addition/removal of NF interfaces) can be also a reason to promote a dynamic adaptation of interfaces. In fact, using the framework, a basic level of adaptation can be achieved. The insertion of specific logic for computing metrics in the *Monitoring* component, which may be a different logic to compute the same indicator in each component, is already a way to make the *Monitoring* component adaptable to the monitoring features provided by each implementation. A similar fact happens with the *Execution* component where the insertion of an appropriate translation engine for each managed component allows to adapt the *Execution* component to each service. Nevertheless, this is only a basic level of adaptation restricted to the functionality of these specific components and the appropriate conversion elements (components, or other units of logic) must be provided by the administrator of the application.

# Bibliography


[ACD+07] Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. Web Services Agreement Specification (WS-Agreement). `http://www.ogf.org/documents/GFD.107.pdf`, March 2007. Open Grid Forum, Grid Resource Allocation Agreement Protocol (GRAAP) WG.

[ADG10] Françoise André, Erwan Daubert, and Guillaume Gauvrit. Towards a generic context-aware framework for self-adaptation of service-oriented architectures. In *Proceedings of the 2010 Fifth International Conference on Internet and Web Applications and Services*, ICIW '10, pages 309–314, Washington, DC, USA, 2010. IEEE Computer Society.

[All] Globus Alliance. MDS – Monitoring and Discovery System. `http://www.globus.org/mds/`.

[AMD] AMD. AMD CodeAnalyst Performance Analyzer. `http://developer.amd.com/cpu/CodeAnalyst/Pages/default.aspx`.

[App] Apple. Optimizing your application with shark 4. `http://developer.apple.com/tools/sharkoptimize.html`.

[AR09] Mohammad Alrifai and Thomas Risse. Combining global optimization with local selection for efficient qos-aware service composition. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 881–890, New York, NY, USA, 2009. ACM.

[ASR10] Mohammad Alrifai, Dimitrios Skoutas, and Thomas Risse. Selecting skyline services for qos-based web service composition. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 11–20, New York, NY, USA, 2010. ACM.

[AST09] R. Asadollahi, M. Salehie, and L. Tahvildari. Starmx: A framework for developing self-managing java-based systems. In *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09. ICSE Workshop on*, pages 58 –67, May 2009.

[BAP05] Jérémy Buisson, Françoise André, and Jean-Louis Pazat. A framework for dynamic adaptation of parallel components. In *Parallel Computing: Current & Future Issues of High-End Computing International Conference ParCo*, volume 33 of *NIC Series*, page 65, Malaga Spain, 2005.

[BAP06] Jérémy Buisson, Françoise André, and Jean-Louis Pazat. Performance and practicability of dynamic adaptation for parallel computing: an experience feedback from Dynaco. Research report, 2006.

[BCC+01] Francine Berman, Andrew Chien, Keith Cooper, Jack Dongarra, Ian Foster, Dennis Gannon, Lennart Johnsson, Ken Kennedy, Carl Kesselman, Mellor-Crum John, Dan Reed, Linda Torczon, and Rich Wolski. The grads project: Software support for high-level grid application development. *Int. J. High Perform. Comput. Appl.*, 15:327–344, November 2001.

[BCD+09] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. Gcm: a grid extension to fractal for autonomous distributed components. *Annals of Telecommunications*, 64:5–24, 2009. 10.1007/s12243-008-0068-8.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[BDIM04] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[BDNG06] L. Baresi, E. Di Nitto, and C. Ghezzi. Toward open-world software: Issue and challenges. *Computer*, 39(10):36 –43, 2006.

[BF05] P. A. Bonatti and P. Festa. On optimal service selection. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 530–538, New York, NY, USA, 2005. ACM.

[BG07] Domenico Bianculli and Carlo Ghezzi. Monitoring conversational web services. In *2nd international workshop on Service oriented software engineering: in conjunction with the 6th ESEC/FSE joint meeting*, IW-SOSWE '07, pages 15–21, New York, NY, USA, 2007. ACM.

[BGG04] Luciano Baresi, Carlo Ghezzi, and Sam Guinea. Smart monitors for composed services. In *Proceedings of the 2nd international conference on Service oriented computing*, ICSOC '04, pages 193–202, New York, NY, USA, 2004. ACM.

[BGP06] Luciano Baresi, Sam Guinea, and Pierluigi Plebani. Ws-policy for service monitoring. In Christoph Bussler and Ming-Chien Shan, editors, *Technologies for E-Services*, volume 3811 of *Lecture Notes in Computer Science*, pages 72–83. Springer Berlin / Heidelberg, 2006. 10.1007/11607380_7.

[BHN09] Françoise Baude, Ludovic Henrio, and Paul Naoumenko. Structural reconfiguration: An autonomic strategy for gcm components. In *Proceedings of the 2009 Fifth International Conference on Autonomic and Autonomous Systems*, pages 123–128, Washington, DC, USA, 2009. IEEE Computer Society.

[BRL07] Fabien Baligand, Nicolas Rivierre, and Thomas Ledoux. A declarative approach for qos-aware web service compositions. In Bernd Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *Service-Oriented Computing – ICSOC 2007*, volume 4749 of *Lecture Notes in Computer Science*, pages 422–428. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-74974-5_38.

[BRL08] Fabien Baligand, Nicolas Rivierre, and Thomas Ledoux. Qos policies for business processes in service oriented architectures. In Athman Bouguettaya, Ingolf Krueger, and Tiziana Margaria, editors, *Service-Oriented Computing – ICSOC 2008*, volume 5364 of *Lecture Notes in Computer Science*, pages 483–497. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-89652-4_36.

[BS03] M. Baker and G. Smith. Gridrm: an extensible resource monitoring system. In *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pages 207 – 214, dec. 2003.

[BSR+06] Rainer Berbner, Michael Spahn, Nicolas Repp, Oliver Heckmann, and Ralf Steinmetz. Heuristics for qos-aware web service composition. *Web Services, IEEE International Conference on*, 0:72–82, 2006.

[BWRJ08] Lianne Bodenstaff, Andreas Wombacher, Manfred Reichert, and Michael C. Jaeger. Monitoring dependencies for slas: The mode4sla approach. *Services Computing, IEEE International Conference on*, 1:21–29, 2008.

[cca] Cca forum homepage. `http://www.cca-forum.org/`.

[ccm] Corba 3.1 specifications. `http://www.omg.org/spec/CORBA/3.1/`.

[CDPEV05] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. Qos-aware replanning of composite web services. In *Proceedings of the IEEE*

*International Conference on Web Services*, ICWS '05, pages 121–129, Washington, DC, USA, 2005. IEEE Computer Society.

[CDPEV08] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. A framework for qos-aware binding and re-binding of composite web services. *J. Syst. Softw.*, 81:1754–1769, October 2008.

[CEGM+10] Clovis Chapman, Wolfgang Emmerich, Fermín Galán Márquez, Stuart Clayman, and Alex Galis. Software architecture definition for on-demand cloud provisioning. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 61–72, New York, NY, USA, 2010. ACM.

[CFFK01] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium on*, pages 181 –194, Aug 2001.

[CGC+10] Stuart Clayman, Alex Galis, Clovis Chapman, Giovanni Toffetti, Luis Rodero-Merino, Luis M. Vaquero, Kenneth Nagin, and Benny Rochwerger. *Monitoring Service Clouds in the Future Internet*, volume 0, pages 115–126. IOS Press, 2010.

[CGM10] S. Clayman, A. Galis, and L. Mamatas. Monitoring virtual networks with lattice. In *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*, pages 239 –246, 2010.

[CH04] Humberto Cervantes and Richard S. Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. *Software Engineering, International Conference on*, 0:614–623, 2004.

[Cha04] David Chappell. *Enterprise Service Bus*. O'Reilly Media, Inc., 2004.

[Cha07] David Chappell. Introducing SCA. `http://www.davidchappell.com/articles/introducing_sca.pdf`, July 2007.

[CKF+02] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. *Dependable Systems and Networks, International Conference on*, 0:595, 2002.

[CMJ+08] Tony Chau, Vinod Muthusamy, Hans-Arno Jacobsen, Elena Litani, Allen Chan, and Phil Coulthard. Automating sla modeling. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, CASCON '08, pages 10:126–10:143, New York, NY, USA, 2008. ACM.

[CRS07] Denis Conan, Romain Rouvoy, and Lionel Seinturier. Scalable processing of context information with cosmos. In Jadwiga Indulska and Kerry Raymond, editors, *Distributed Applications and Interoperable Systems*, volume 4531 of *Lecture Notes in Computer Science*, pages 210–224. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-72883-2_16.

[CS09a] Marco Comuzzi and George Spanoudakis. Describing and verifying monitoring capabilities for sla-driven service based systems. In J. Eder E. Yu, editor, *Proceedings of the Forum at the CAiSE 2009 Conference*, volume 453, pages 43–48, Amsterdam, 2009.

[CS09b] Marco Comuzzi and George Spanoudakis. A framework for hierarchical and recursive monitoring of service based systems. In *Proceedings of the 2009 Fourth International Conference on Internet and Web Applications and Services*, pages 383–388, Washington, DC, USA, 2009. IEEE Computer Society.

[CS10] Marco Comuzzi and George Spanoudakis. Dynamic set-up of monitoring infrastructures for service based systems. In *Proceedings of the 2010 ACM Symposium*

*on Applied Computing*, SAC '10, pages 2414–2421, New York, NY, USA, 2010. ACM.

[DAT08] Demian Antony D'Mello, V.S. Ananthanarayana, and Santhi Thilagam. A qos broker based architecture for dynamic web service selection. *Asia International Conference on Modelling & Simulation*, 0:101–106, 2008.

[DDF+06] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1:223–259, December 2006.

[DFG+08] Marco Danelutto, Paraskevi Fragopoulou, Vladimir Getov, Francoise Baude, Denis Caromel, Ludovic Henrio, and Paul Naoumenko. A flexible model and implementation of component controllers. In *Making Grids Work*, pages 31–43. Springer US, 2008. 10.1007/978-0-387-78448-9_3.

[DL05] Pierre-Charles David and Thomas Ledoux. Wildcat: a generic framework for context-aware applications. In *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, MPAC '05, pages 1–7, New York, NY, USA, 2005. ACM.

[DL06] Pierre-Charles David and Thomas Ledoux. An aspect-oriented approach for developing self-adaptive fractal components. In Welf Löwe and Mario Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 82–97. Springer Berlin / Heidelberg, 2006. 10.1007/11821946_6.

[DLLC09] Pierre-Charles David, Thomas Ledoux, Marc Léger, and Thierry Coupaye. Fpath and fscript: Language support for navigation and reliable reconfiguration of fractal architectures. *Annals of Telecommunications*, 64:45–63, 2009. 10.1007/s12243-008-0073-y.

[(DM] Distributed Managment Task Force Inc (DMTF). Web Services Management (WS-MAN). `http://www.dmtf.org/standards/wsman`.

[DNGM+08] Elisabetta Di Nitto, Carlo Ghezzi, Andreas Metzger, Mike Papazoglou, and Klaus Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15:313–341, 2008. 10.1007/s10515-008-0032-x.

[EH07] Clement Escoffier and Richard Hall. Dynamically adaptable applications with ipojo service components. In Markus Lumpe and Wim Vanderperren, editors, *Software Composition*, volume 4829 of *Lecture Notes in Computer Science*, pages 113–128. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-77351-1_9.

[EHL07] C. Escoffier, R.S. Hall, and P. Lalanda. ipojo: an extensible service-oriented component framework. In *Services Computing, 2007. SCC 2007. IEEE International Conference on*, pages 474 –481, 2007.

[ejb] Enterprise Java Beans. `http://java.sun.com/products/ejb/`.

[Erl] Thomas Erl. Soa principles. `http://www.soaprinciples.com`.

[Esp] EsperTech. Event Stream Intelligence: Esper & NEsper. `http://esper.codehaus.org/`.

[fab] Fabric3. `http://www.fabric3.org/`.

[Fen] Jay Fenlason. The GNU profiler: gprof. `http://sourceware.org/binutils/docs-2.16/gprof/`.

[FK99] Ian Foster and Carl Kesselman, editors. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[FKNT03] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. *The Physiology of the Grid*, pages 217–249. John Wiley & Sons, Ltd, 2003.

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15:200–222, August 2001.

[Fos02] Ian Foster. What is the Grid? - a three point checklist. *GRIDtoday*, 1(6), July 2002.

[Fos06] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. *Journal of Computer Science and Technology*, 21:513–520, 2006. 10.1007/s11390-006-0513-y.

[FZRL08] I. Foster, Yong Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1 –10, 2008.

[Gan] Ganglia. Ganglia Monitoring System. `http://ganglia.info/`.

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37:46–54, 2004.

[GDA10] G. Gauvrit, E. Daubert, and F. Andre. Safdis: A framework to bring self-adaptability to service-based distributed applications. In *Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on*, pages 211 –218, 2010.

[GG07] Carlo Ghezzi and Sam Guinea. Run-time monitoring in service-oriented architectures. *Test and Analysis of Web Services*, pages 237–264, 2007.

[GMPLMT10] Carlo Ghezzi, Alfredo Motta, Valerio Panzica La Manna, and Giordano Tamburrelli. Qos driven dynamic binding in-the-many. In George Heineman, Jan Kofron, and Frantisek Plasil, editors, *Research into Practice. Reality and Gaps*, volume 6093 of *Lecture Notes in Computer Science*, pages 68–83. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-13821-8_7.

[GPD+10] Issac Garcia, Gabriel Pedraza, Bassem Debbabi, Philippe Lalanda, and Catherine Hamon. Towards a service mediation framework for dynamic applications. *Asia-Pacific Conference on Services Computing. 2006 IEEE*, 0:3–10, 2010.

[HLM+09] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: a consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 41–50, New York, NY, USA, 2009. ACM.

[Hor01] Paul Horn. Autonomic computing: Ibm's perspective on the state of information technology. *Computing Systems*, 2007(Jan):1–40, 2001.

[Hyp] Hyperic. Cloudstatus monitoring. `http://www.hyperic.com/products/cloud-status-monitoring`.

[IBM] IBM. WebSphere Application Server V7 Feature Pack for Service Component Architecture. `http://www-01.ibm.com/software/webservers/appserv/was/featurepacks/sca/`.

[IBM06] IBM. An architectural blueprint for autonomic computing. *white paper*, Fourth Edition(June), 2006.

[Int] Intel. Intel VTune Amplifier. `http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/`.

[JCL+10] Hans-Arno Jacobsen, Alex Cheung, Guoli Li, Maniymaran Balasubramaneyam, Vinod Muthusamy, and Reza-Sherafat Kazemzadeh. *The PADRES Publish/Subscribe System*, chapter 8, pages 164–205. 2010.

[Joh04] Mark W. Johnson. Monitoring and diagnosing applications with arm 4.0. In *Computer Measurement Group*, pages 473–484, 2004.

[KC03] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41 – 50, January 2003.

[KL03] Alexander Keller and Heiko Ludwig. The wsla framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11:57–81, 2003. 10.1023/A:1022445108617.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0053381.

[LDK04] Heiko Ludwig, Asit Dan, and Robert Kearney. Cremona: an architecture and library for creation and monitoring of ws-agreents. In *Proceedings of the 2nd international conference on Service oriented computing*, ICSOC '04, pages 65–74, New York, NY, USA, 2004. ACM.

[LKD+03] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P. King, and Richard Franck. Web service level agreement (WSLA) language specification, 2003.

[LLJ+05] Shulei Liu, Yunxiang Liu, Ning Jing, Guifen Tang, and Yu Tang. A dynamic web service selection strategy with qos global optimization based on multi-objective genetic algorithm. In Hai Zhuge and Geoffrey Fox, editors, *Grid and Cooperative Computing - GCC 2005*, volume 3795 of *Lecture Notes in Computer Science*, pages 84–89. Springer Berlin / Heidelberg, 2005. 10.1007/11590354_10.

[LMJ10] Guoli Li, Vinod Muthusamy, and Hans-Arno Jacobsen. A distributed service-oriented architecture for business process execution. *ACM Trans. Web*, 4:2:1–2:33, January 2010.

[LMRD10] Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar. Monitoring, prediction and prevention of sla violations in composite services. In *Proceedings of the 2010 IEEE International Conference on Web Services*, ICWS '10, pages 369–376, Washington, DC, USA, 2010. IEEE Computer Society.

[Log] LogicMonitor. Logicmonitor. `http://www.logicmonitor.com/`.

[LRD09] Philipp Leitner, Florian Rosenberg, and Schahram Dustdar. Daios: Efficient dynamic web service invocation. *IEEE Internet Computing*, 13:72–80, 2009.

[LS10] Davide Lorenzoli and George Spanoudakis. Everest+: run-time sla violations prediction. In *Proceedings of the 5th International Workshop on Middleware for Service Oriented Computing*, MW4SOC '10, pages 13–18, New York, NY, USA, 2010. ACM.

[LWR+10] Philipp Leitner, Branimir Wetzstein, Florian Rosenberg, Anton Michlmayr, Schahram Dustdar, and Frank Leymann. Runtime prediction of service level agreement violations for composite services. In Asit Dan, Frédéric Gittler, and Farouk Toumani, editors, *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*, volume 6275 of *Lecture Notes in Computer Science*, pages 176–186. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-16132-2_17.

[MBK+09] Nebil Ben Mabrouk, Sandrine Beauche, Elena Kuznetsova, Nikolaos Georgantas, and Valérie Issarny. Qos-aware service composition in dynamic service oriented

environments. In *Proceedings of the ACM/IFIP/USENIX 10th international conference on Middleware*, Middleware'09, pages 123–142, Berlin, Heidelberg, 2009. Springer-Verlag.

[MCC04] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.

[MCD10] Daniel A. Menascé, Emiliano Casalicchio, and Vinod Dubey. On optimal service selection in service oriented architectures. *Performance Evaluation*, 67:659–675, August 2010.

[MDL10] Yoann Maurel, Ada Diaconescu, and Philippe Lalanda. Ceylon: A service-oriented framework for building autonomic managers. *Engineering of Autonomic and Autonomous Systems, IEEE International Workshop on*, 0:3–11, 2010.

[MJ10] Vinod Muthusamy and Hans-Arno Jacobsen. Bpm in cloud architectures: business process management with slas and events. In *Proceedings of the 8th international conference on Business process management*, BPM'10, pages 5–10, Berlin, Heidelberg, 2010. Springer-Verlag.

[MJC+09] Vinod Muthusamy, Hans-Arno Jacobsen, Tony Chau, Allen Chan, and Phil Coulthard. Sla-driven business process management in soa. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '09, pages 86–100, New York, NY, USA, 2009. ACM.

[MKS09] Hausi A. Müller, Holger M. Kienle, and Ulrike Stege. Software engineering. chapter Autonomic Computing Now You See It, Now You Don't, pages 32–54. Springer-Verlag, Berlin, Heidelberg, 2009. ⁘

[MLBH+09] B. Morin, T. Ledoux, M. Ben Hassine, F. Chauvel, O. Barais, and J.-M. Jezequel. Unifying runtime adaptation and design evolution. In *Computer and Information Technology, 2009. CIT '09. Ninth IEEE International Conference on*, volume 1, pages 104 –109, 2009.

[MPMJS05] Graham Morgan, Simon Parkin, Carlos Molina-Jimenez, and James Skene. Monitoring middleware for service level agreements in heterogeneous environments. In Matohisa Funabashi and Adam Grzech, editors, *Challenges of Expanding Internet: E-Commerce, E-Business, and E-Government*, volume 189 of *IFIP International Federation for Information Processing*, pages 79–93. Springer Boston, 2005. 10.1007/0-387-29773-1_6.

[MR09] Jim Marino and Michael Rowley. *Understanding SCA (Service Component Architecture)*. Addison-Wesley Professional, 1st edition, 2009.

[MRD08] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 815–824, New York, NY, USA, 2008. ACM.

[MRLD08] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Advanced event processing and notifications in service runtime environments. In *Proceedings of the second international conference on Distributed event-based systems*, DEBS '08, pages 115–125, New York, NY, USA, 2008. ACM.

[MRLD09] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Comprehensive QoS monitoring of Web services and event-based SLA violation detection. In *Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing*, MWSOC '09, pages 1–6, New York, NY, USA, 2009. ACM.

[MRLD10] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. End-to-end support for qos-aware service selection, binding, and mediation in vresco. *IEEE Transactions on Services Computing*, 3:193–205, 2010.

[MRP+07] Anton Michlmayr, Florian Rosenberg, Christian Platzer, Martin Treiber, and Schahram Dustdar. Towards recovering the broken SOA triangle: a software engineering perspective. In *2nd international workshop on Service oriented software engineering: in conjunction with the 6th ESEC/FSE joint meeting*, IW-SOSWE '07, pages 22–28, New York, NY, USA, 2007. ACM.

[MS07] Khaled Mahbub and George Spanoudakis. Monitoring WS-Agreement s: An Event Calculus-based Approach. In Luciano Baresi and Elisabetta Di Nitto, editors, *Test and Analysis of Web Services*, pages 265–306. Springer Berlin Heidelberg, 2007. 10.1007/978-3-540-72912-9_10.

[nag] Nagios – Nagios - The Industry Standard In Open Source Monitoring. `http://www.nagios.org/`.

[net] .NET Specification. `http://www.microsoft.com/net/`.

[OASa] OASIS. UDDI Version 3.0.2. `http://uddi.org/pubs/uddi_v3.htm`.

[OASb] OASIS. Web Services Business Process Execution Language Version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[OASc] OASIS. Web Services Distributed Management (WSDM) Version 1.1. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsdm`.

[OASd] OASIS. Web Services Resource Framework (WSRF). `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf`.

[OAS06] OASIS. Reference Model for Service Oriented Architecture v1.0. `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf`, October 2006.

[OAS07] OASIS team and other partners in the CoreGRID Programming Model Virtual Institute. Basic features of the grid component model. Technical report, Oct. 2007. Deliverable D.PM.04, CoreGRID, Programming Model Institute.

[OGT+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14:54–62, May 1999.

[OMT08] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *Companion of the 30th international conference on Software engineering*, ICSE Companion '08, pages 899–910, New York, NY, USA, 2008. ACM.

[Ora] Oracle. JavaTM Virtual Machine Tool Interface (JVM TI). `http://download.oracle.com/javase/6/docs/technotes/guides/jvmti/`.

[Ore96] Peyman Oreizy. Issues in the runtime modification of software architectures. Technical Report UCI-ICS-TR-96-35, Dept. of Information and Computer Science, University of California, Irvine, CA 92697, August 1996.

[OSO07a] OSOA. Sca policy framework, v1.00. `http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications`, Mar 2007.

[OSO07b] OSOA. Sca service component architecture, assembly model, v1.00. `http://www.osoa.org/display/Main/Service+Component+Architecture+Home`, Mar 2007.

[Pap03] Mike P. Papazoglou. Service -oriented computing: Concepts, characteristics and directions. *Web Information Systems Engineering, International Conference on*, 0:3, 2003.

[par] The Paremus Service Fabric - A Technical Overview. `http://www.paremus.com/resources/_resources_documents/The_Paremus_Service_Fabric_-_A_Technical_Overview.html`.

[PH07] Mike P. Papazoglou and Willem-Jan Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16:389–415, July 2007.

[PTDL07] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40:38–45, November 2007.

[PWZW09] Antoine Pichot, Oliver Waldrich, Wolfgang Ziegler, and Philipp Wieder. Towards dynamic service level agreement negotiation:an approach based on ws-agreement. In Will Aalst, John Mylopoulos, Norman M. Sadeh, Michael J. Shaw, Clemens Szyperski, José Cordeiro, Slimane Hammoudi, and Joaquim Filipe, editors, *Web Information Systems and Technologies*, volume 18 of *Lecture Notes in Business Information Processing*, pages 107–119. Springer Berlin Heidelberg, 2009. 10.1007/978-3-642-01344-7_9.

[RBL+09] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Cáceres, M. Ben-Yehuda, W. Emmerich, and F. Galán. The reservoir model and architecture for open federated cloud computing. *IBM J. Res. Dev.*, 53:535–545, July 2009.

[RPD06] Florian Rosenberg, Christian Platzer, and Schahram Dustdar. Bootstrapping performance and dependability attributes ofweb services. In *Proceedings of the IEEE International Conference on Web Services*, pages 205–212, Washington, DC, USA, 2006. IEEE Computer Society.

[RRS+10] Daniel Romero, Romain Rouvoy, Lionel Seinturier, Sophie Chabridon, Denis Conan, and Nicolas Pessemier. Enabling Context-Aware Web Services: A Middleware Approach for Ubiquitous Environments. In Michael Sheng, Jian Yu, and Schahram Dustdar, editors, *Enabling Context-Aware Web Services: Methods, Architectures, and Technologies*, pages 113–135. Chapman and Hall/CRC, 05 2010.

[RS09] Pawel Rubach and Michael Sobolewski. Autonomic sla management in federated computing environments. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ICPPW '09, pages 314–321, Washington, DC, USA, 2009. IEEE Computer Society.

[RSR01] Randy L. Ribler, Huseyin Simitci, and Daniel A. Reed. The Autopilot performance-directed adaptive control system. *Future Generation Computer Systems*, 18:175–187, September 2001.

[RVSR98] R.L. Ribler, J.S. Vetter, H. Simitci, and D.A. Reed. Autopilot: adaptive control of distributed applications. In *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, pages 172 –179, July 1998.

[SDHS05] M.A. Serhani, R. Dssouli, A. Hafid, and H. Sahraoui. A qos broker based architecture for efficient web services selection. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, pages 113 – 120 vol.1, 2005.

[SDM01] Akhil Sahai, Anna Durante, and Vijay Machiraju. Towards automated sla management for web services. Hewlett-Packard Research Report HPL-2001-310 (R. 1), HP Laboratories Palo Alto, July 2001.

[Sha99] Murray Shanahan. Artificial intelligence today. chapter The event calculus explained, pages 409–430. Springer-Verlag, Berlin, Heidelberg, 1999.

[Ske07] James Skene. *Language support for service-level agreements for application-service provision*. PhD thesis, Department of Computer Science, University College London, November 2007.

[SLE04] James Skene, D. Davide Lamanna, and Wolfgang Emmerich. Precise service level agreements. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 179–188, Washington, DC, USA, 2004. IEEE Computer Society.

[SMF+09] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable sca applications with the frascati platform. *Services Computing, IEEE International Conference on*, 0:268–275, 2009.

[SMS+02] Akhil Sahai, Vijay Machiraju, Mehmet Sayal, Aad van Moorsel, and Fabio Casati. Automated sla monitoring for web services. In Metin Feridun, Peter Kropf, and Gilbert Babin, editors, *Management Technologies for E-Commerce and E-Business Applications*, volume 2506 of *Lecture Notes in Computer Science*, pages 28–41. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-36110-3_6.

[sof] SOFA Component Model. `http://sofa.ow2.org/`.

[SPM+06] Jennifer M Schopf, Laura Pearlman, Neill Miller, Carl Kesselman, Ian Foster, Mike D'Arcy, and Ann Chervenak. Monitoring the grid with the globus toolkit mds4. *Journal of Physics: Conference Series*, 46(1):521, 2006.

[SRP+06] Jennifer M Schopf, Ioan Raicu, Laura Pearlman, Neill Miller, Carl Kesselman, Ian Foster, and Mike D'Arcy. Monitoring and discovery in a web services framework: Functionality and performance of globus toolkit mds4. Technical Report ANL/MCS-P1315-0106, Argonne National Laboratory, January 2006.

[SSCE07] James Skene, Allan Skene, Jason Crampton, and Wolfgang Emmerich. The monitorability of service-level agreements for application-service provision. In *Proceedings of the 6th international workshop on Software and performance*, WOSP '07, pages 3–14, New York, NY, USA, 2007. ACM.

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4:14:1–14:42, May 2009.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[TAG+02] Brian Tierney, Ruth Aydt, Dan Gunter, Warren Smith, Valerie Taylor, Rich Wolski, and Martin Swany. A Grid Monitoring Architecture. `http://www.ogf.org/documents/GFD.7.pdf`, August 2002. Global Grid Forum, Performance Working Group.

[TBNF09] Hong-Linh Truong, Peter Brunner, Vlad Nae, and Thomas Fahringer. Dipas: A distributed performance analysis service for grid service-based workflows. *Future Generation Computer Systems*, 25(4):385 – 398, 2009.

[TF04] Hong-Linh Truong and Thomas Fahringer. Scalea-g: A unified monitoring and performance analysis system for the grid. *Sci. Program.*, 12:225–237, December 2004.

[TPP+03] Vladimir Tosic, Bernard Pagurek, Kruti Patel, Babak Esfandiari, and Wei Ma. Management applications of the web service offerings language (wsol). In Johann Eder and Michele Missikoff, editors, *Advanced Information Systems Engineering*,

volume 2681 of *Lecture Notes in Computer Science*, pages 1029–1029. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-45017-3_32.

[TSF06] Hong-Linh Truong, Robert Samborski, and Thomas Fahringer. Towards a framework for monitoring and analyzing qos metrics of grid services. In *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, E-SCIENCE '06, pages 65–, Washington, DC, USA, 2006. IEEE Computer Society.

[tus] Apache Tuscany. `http://tuscany.apache.org/`.

[vHRH+09] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. Continuous monitoring of software services: Design and application of the Kieker framework. Technical Report TR-0921, Department of Computer Science, University of Kiel, Germany, November 2009.

[W3Ca] W3C. SOAP Version 1.2. `http://www.w3.org/TR/soap/`.

[W3Cb] W3C. Web Services Architecture. `http://www.w3.org/TR/ws-arch/`.

[W3Cc] W3C. Web Services Description Language (WSDL) Version 2.0. `http://www.w3.org/TR/wsdl20/`.

[W3Cd] W3C. Web Services Policy 1.5 - Framework (WS-Policy). `http://www.w3.org/TR/ws-policy/`.

[YL05] Tao Yu and Kwei-Jay Lin. A broker-based framework for qos-aware web service composition. In *Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05) on e-Technology, e-Commerce and e-Service*, EEE '05, pages 22–29, Washington, DC, USA, 2005. IEEE Computer Society.

[YZL07] Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Trans. Web*, 1, May 2007.

[ZBHN+04] Liangzhao Zeng, Boualem Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. Qos-aware middleware for web services composition. *IEEE Trans. Softw. Eng.*, 30:311–327, May 2004.

[ZLC07] Liangzhao Zeng, Hui Lei, and Henry Chang. Monitoring the qos for web services. In *Proceedings of the 5th international conference on Service-Oriented Computing*, ICSOC '07, pages 132–144, Berlin, Heidelberg, 2007. Springer-Verlag.

[ZS05] Serafeim Zanikolas and Rizos Sakellariou. A taxonomy of grid monitoring systems. *Future Gener. Comput. Syst.*, 21:163–188, January 2005.

# Autonomic Monitoring and Management of Component-Based Services


## Abstract

Software applications have evolved from monolithic, stable, centralized and structured applications to highly decentralized, distributed and dynamic software, forcing a change in the software development process. Current attention has turned to the development of service-based applications, where the focus is in the provision of functionality that is delivered *as a service* and where independent providers offer services that can be dynamically composed and reused in order to create new added-value applications, giving rise to an ever-growing ecosystem of loosely-coupled, geographically dispersed, and rapidly evolving services.

Among the multiple advantages of the service-based approach to software development, new challenges raise. The dynamic, evolving, and heterogeneous nature of such service compositions makes the management tasks more complex as not all the services are under the control of a single entity, and the environmental changes during the execution of a service composition cannot be completely foreseen. In order to properly address this situation, a service-based application must be able to adapt itself to such conditions.

In this thesis, we present a generic framework that improves the adaptability of component service-based applications by giving a common and efficient means to introduce monitoring and management tasks at runtime into the application, and allows to provide autonomic behaviour. The framework itself follows a component-based approach where the different tasks of an autonomic control loop can be inserted and removed at runtime in order to adapt to the monitoring and managment needs.

We present the design of our framework using a generic component model (SCA), and we provide an implementation using a middleware that supports distributed component-based services (GCM/ProActive). We illustrate the feasibility of our approach through an example service-based application improved with monitoring and management features, providing an autonomic behaviour. The examples and evaluations conducted lead us to think that this approach is a feasible tool that can facilitate the insertion and development of autonomic features.


---

# Surveillance et administration autonome de services basés sur des composants


## Résumé

Les applications ont évolué, depuis des logiciels monolithiques, stables, centralisés et fortement structurés, à des logiciels fortement décentralisés, distribués et dynamiques, ce qui a provoqué un changement dans le processus de développement. Ainsi, les préoccupations actuelles sont tournées vers le développement des logiciels orientés services, où le pivot est la fourniture de toute fonctionnalité ?en tant que service?, où des fournisseurs indépendants proposent des services qui peuvent être composés de façon dynamique, et réutilisés pour créer de nouvelles applications à valeur ajoutée, ce qui donne naissance à un écosystème grandissant de services à couplage faible, géographiquement distribués, et qui évoluent rapidement.

Malgré les avantages offerts par l'approche basée services pour le développement des logiciels, il se pose aussi de nouveaux défis. La nature dynamique, évolutive, et hétérogène de ces compositions de services rend les tâches de gestion plus complexes, car les services ne sont plus contrôlés pour une seule entité, et les changements de l'environnement pendant l'exécution d'une composition des services ne peuvent pas être complètement prévus à l'avance. Pour être capable d'affronter ces types de situations, un logiciel basé sur des services doit s'adapter de manière si possible autonome à ces conditions.

Dans cette thèse, nous présentons un canevas générique qui permet d'améliorer l'adaptabilité des logiciels basés sur des services en proposant un moyen uniforme et efficace pour ajouter des tâches de surveillance et de gestion dans une application, et aussi permettre de fournir un comportement autonomique. Le canevas est lui même basé sur des composants logiciels, SCA au niveau de la conception, et GCM/ProActive au niveau de la mise en oeuvre. Nous illustrons la faisabilité de notre approche à travers un exemple d'application basée sur des services, que nous étendons avec des fonctionnalités de surveillance et d'administration autonomes. Ces exemples et leur évaluation nous laissent penser que notre approche est utilisable en pratique.