UNIVERSIDADE TÉCNICA DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Self-management of Systems Built from Adaptable Components

Liliana Wu Freitas Rosa

Supervisor: Doctor Luís Eduardo Teixeira Rodrigues
Co-Supervisor: Doctor Maria Antónia Bacelar da Costa Lopes

Thesis approved in public session to obtain the PhD degree in
Information Systems and Computer Engineering

Jury final classification: Pass with Merit

Jury

Chairperson:
Chairman of the IST Scientific Board
Members of the Committee:
Doctor José Alberto Cardoso e Cunha
Doctor Luís Eduardo Teixeira Rodrigues
Doctor Raffaela Mirandola
Doctor Maria Antónia Bacelar da Costa Lopes
Doctor Luís Manuel Antunes Veiga
Doctor João Manuel Pinheiro Cachopo

Funding Institutions

Fundação para a Ciência e Tecnologia

2012

"In our broad sweep, the city looks like a single gigantic creature - or more like a single collective entity created by many organisms. Countless arteries stretch to the ends of its elusive body, circulating a continuous supply of fresh blood cells, sending new data and collecting the old, sending out new consumables and collecting the old, sending out new contradictions and collecting the old." — Haruki Murakami, in *After Dark*

Acknowledgments

The writing of this thesis and all the research work behind it is an intensive labour of love and hatred, but mostly persistence and a few not so bad ideas. I would like to deeply thank those that shared it with me and helped me along the way. To my advisors, Luís Rodrigues and Antónia Lopes, thank you for the patience and perseverance to make this work what it has become. No words will ever be enough to properly thank you for these years. I am forever indebted to you. To Matti Hiltunen, thank you for believing in this work and let it fly to more fertile grounds. The *fresh air* you brought to my research during my stay at AT&T Labs Research was the turning point that led to the results in this thesis. Finally, I would like to thank Richard Schlichting for all the valuable input and encouraging words during and after my stay in the US.

I would also like to thank INESC-ID and AT&T Labs Research for welcoming me to their institutions. To my fellow PhD students and doctorates at GSD and INESC-ID, thank you for the companionship and all the help. To my colleagues and friends at AT&T Labs Research, thank you for making my stay in the US a great, wonderful experience. Finally, I would like to thank my family and friends for all the support and for the many times you cut me some slack because it was a hopeless case. Thank you from the bottom of my heart.

Abstract

Self-management allows a system to manage its own behavior autonomously, guided by system goals and relying on feedback of the system behavior. The main challenge to self-management is the growing complexity of software systems. Today, systems are heterogeneous collections of different services, technologies, and software components, which transforms the manual management of such systems in a complex, tiresome, and error-prone task.

This thesis proposes a conceptual framework and modeling primitives for the self-management of systems built from both distributed and non-distributed components that can be adapted. The approach relies on knowledge from both system designers and component developers to manage the system. This knowledge is the base of a goal-oriented solution to control the system behavior, according to the execution conditions and load. As a result, it becomes possible to optimize the system performance and maintain the desired behavior, while easing and automatizing the system designer task. Furthermore, the proposed approach also addresses the challenges of adapting such systems by proposing a solution that leverages on reconfiguration strategies.

The evaluation of this work in a main case study: a web based system, with both distributed and centralized components. The evaluation explores different aspects of the approach. It allows to analyze the performance and scalability of the approach, and how the distribution is handled. The obtained results show that not only the approach is able to provide the necessary adaptation support to manage the system, but it also reduces the complexity of the tasks performed by the system designer.

Keywords: Self-management, autonomic computing, self-optimization, control-loop, goal policy, adaptable components, self-adaptive systems

Resumo

Os sistemas computacionais são cada vez mais um conjunto de vários componentes, serviços, tecnologias e middleware. Garantir que os sistemas funcionam de acordo com o desejado torna-se cada vez mais uma tarefa complexa e que exige um grande esforço humano. Tornar a gestão destes sistemas autónoma, sem necessidade de intervenção de um operador humano não só a facilita, como dá garantias de correcção e maior rapidez.

Esta dissertação propõe uma moldura conceptual, constituída por vários conceitos, modelos e metodologias que permitir gerir de forma automática os sistemas compostos por vários elementos, distribuídos ou não. A principal contribuição deste trabalho consiste em tirar partido do conhecimento dos arquitectos dos sistemas e das equipas que fazem o desenvolvimentos dos vários elementos. Assim propõe uma solução baseada em objectivos para controlar e adaptar o sistema em tempo de execução de acordo com a carga e o estado do ambiente de execução. Para além disso, propõe também várias estratégias de reconfiguração para executar as alterações do sistema.

A avaliação deste trabalho utilizou um caso de estudo principal: uma sistema web-based. Os protótipos construídos para o caso de estudo permitiram analisar as questões de desempenho, escalabilidade, e suporte à distribuição. Os resultados obtidos da experimentação com os protótipos mostram que não só a abordagem é bem sucedida a gerir automaticamente o sistema, como diminui o esforço e complexidade das tarefas de gestão executadas pelo arquitecto do sistema.

Palavras-chave: adaptação, auto-optimização, políticas de adaptação, gestão autónoma, autonomic computing

Contents

The fact that our task is exactly commensurate with our life gives it the appearance of being infinite.

Franz Kafka

Chapter 1

Introduction

This thesis addresses the problem of developing self-adaptive systems composed by adaptable (distributed and non-distributed) components. The emphasis of the work is on the use of goal policies as a means to manage the inherent complexity of controlling the adaptation.

1.1 Motivation

Software is now pervasive in our daily life, and an increasing number of tasks depends on the correct and efficient execution of programs that are built by composing multiple software components or services. These programs often need to operate in dynamic and unpredictable environments, and have to cope with events such as changes in the available resources, variable workloads, shifts in user requirements, etc. This motivates the design of self-adaptive systems, i.e., systems that continuously monitor the execution environment and, without human intervention, change their behavior at runtime in response to changes that may deviate their behavior or performance from the intended.

However, the development of self-adaptive systems is far from trivial. In fact, it has been pointed out as one of the main challenges in Software Engineering [GvdHT09]. It has been recognized [CLG+09] that the proper realization of self-adaptation functionality still remains a significant intellectual challenge due to the lack of methods, techniques, and tools that enable the systematic development of this class of systems. The problem stems from the fact that

adapting a complex system, built as a collection of components is far more complex than adapting isolated components (the term component is used to name components, services, protocols, or even technologies that offer a number of functionalities and serve a purpose in the system). Many systems already offer different options for customizing their behavior at runtime, including loadable modules and other numerous configuration possibilities. Such alternatives in individual services can be used to adapt the behavior of the composed system in response to changes in its execution environment. However, in the presence of many individual components that can be adapted in different manners, the management of the global system adaptation becomes a quite complex task [Kep05]. The definition of the appropriate adaptation logic for the system, not only requires the detailed knowledge of every system component and its interactions, but also of how it can be adapted. The space of possible configurations can easily become very large and, hence, finding the right adaptation strategies becomes quite difficult, time-consuming, and error-prone.

1.2 Problem Statement

The thesis addresses the engineering of self-adaptive systems built from individual adaptable components, which may have been developed by different teams in a non-coordinated manner. The aim of the work is to develop an approach to the construction of this class of self-adaptive systems and to contribute with a set of techniques that facilitate their design and implementation, addressing the challenges of adaptation management described previously.

The work follows key principles that have emerged from previous research [CLG+09, GSC09, OMT08], namely that mechanisms supporting self-adaptation should be separated from the actual system, and clearly identified in a control loop. A key feature of the approach is the adoption of high-level forms of adaptation policies based on goals and utility functions.

1.3 Contributions

The thesis contributions are related to the specification of adaptation policies and with the mechanisms required to implement these policies. On one side, the thesis proposes:

- A conceptual framework for the self-management of systems built from multiple adaptable components, both distributed and non-distributed.

- Modeling primitives providing means for expressing the adaptation logic of a system explicitly, separated from the description of the individual services, in two complementary forms: i) in terms of action policies, defined by event-condition-action rules and ii) in terms of goal policies addressing key performance indicators.

- Techniques for runtime support of such adaptation policies, namely techniques that support the dynamic generation of action policies from high-level goals, employed whenever the system deviates from its desired behaviour.

On the other side, the thesis also addresses the implementation of the adaptation in runtime, in particular when adaptation is performed on distributed components, including:

- The re-design of a protocol composition framework to facilitate the runtime reconfiguration of protocol stacks.

To assess these contributions we use as case study a web based system that includes a combination of distributed and centralized components.

1.4 Results

Given the contributions listed above, the results of this thesis are the following:

- An engine to derive at runtime adaptation strategies from high-level policies using the representation of the system, the characteristics of each individual component and available adaptations, and context information regarding the execution environment.

- The design of R-Appia, a reconfigurable version of the Appia protocols composition and execution framework.

- The design of a set of context sensors and effectors to support the dynamic adaptation of web applications based on the Apache HTTP server and the *Infinispan* data-grid.

- A prototype and an experimental evaluation of an adaptable group communication protocol and an in-memory distributed data grid.

- A prototype and an experimental evaluation of an adaptable web application.

1.5 Research History

This work has background in the candidate's Master dissertation, which proposes a policy-oriented approach to the construction of adaptive communication protocols [RLR06]. From this work experience, it was possible to identify some of the shortcomings of self-adaptive solutions. One observation was the difficulty of describing an appropriate adaptation policy when many adaptations or communication protocols were involved. The description of the policy demanded not only a high-level comprehension of what is an adequate performance and behavior for the system, but also a detailed low-level comprehension of the protocols and how they could be adapted. Another important observation was that the design and development of the adaptation support was a demanding effort, were some aspects were tied to the managed system, while others were more generic. This was true at all levels, but specially at the planning and execution levels, despite the system being distributed.

These observations lead to an iterative effort of developing an approach that would address the challenges above, aiming at facilitating the task of developing self-adaptive systems. The thesis efforts began with studying the self-management support and how it was influenced by the distribution and composition of the system. The next step was to facilitate the description of the adaptive behavior of the system. The candidate's visit to AT&T Labs Research, and the collaboration with Matti Hiltunen and Richard Schlichting was instrumental to expand these ideas, and experiment them in a different area, of web applications. Eventually, the idea of automatically extracting the policy was first experimented in a web application with non-distributed components only and later with also distributed components, by running the web application on a cluster of nodes using an in-memory distributed data-grid.

It is worth to mention a number of works related with this thesis that have been developed in the GSD group at INESC-ID during the PhD program of the candidate. Cristina Fonseca did

a study of the advantages of using specialized switching protocols to speed up the coordination required to perform distributed reconfiguration [FRR09]. João Ferreira applied some of these ideas in the design of A-OSGi, an adaptable OSGi framework [FLR10]. Tiago Taveira has worked on the implementation of a reconfigurable group communication system based on some of the results reported here [Tav10].

1.6 Roadmap

The thesis is organized in the following manner. Chapter 2 discusses the concepts and terminology of adaptive systems. It also provides an overview of the state of the art of the many topics addressed in this work. Chapter 3 gives an overview of the entire approach, addressing the activities of adaptation support. The next three chapters focus on technical aspects of the approach. Chapter 4 covers the knowledge model. Chapter 5 addresses the rule and goal-oriented planning. Chapter 6 covers the execution of the adaptations using reconfiguration strategies. Chapter 7 presents the evaluation of the proposed approach. The thesis is concluded in Chapter 8, with some final remarks and a discussion of future work.

There are no facts, only interpretations.

Friedrich Nietzsche, in *Notebooks*

Chapter 2

Self-adaptive Systems

There are literally hundreds, if not thousands, of approaches that rely on some sort of adaptive solution to improve results when facing dynamic scenarios. They range from problem-specific to general, and come in a variety of packages, addressing different research areas. Not only much of the existing work is not of interest to this thesis, but its systematization and comparison would not bring any significant and relevant benefits. Instead, this chapter has two goals. The first goal is to provide some clarification of the terms and concepts most commonly used in this research area. The second goal is to cover the key concerns when designing and developing such systems, in the state of the art.

This chapter is divided in two main sections. The first addresses the concepts and design principles, while the second covers the design and development of adaptive systems, including a discussion of the main limitations and challenges.

2.1 Background

This section presents a concise description of the nuclear concepts commonly employed in adaptive systems literature. This literature includes mainly sources from context-awareness research, autonomic computing, and renowned conferences in software engineering, such as ICSE[1] and FSE[2], but not only. Following the introduction of elementary concepts of adaptive systems,

[1] International Conference on Software Engineering®
[2] ACM SIGSOFT Symposium on the Foundations of Software Engineering

this section also presents a brief description of two best practice design principles for software development: separation of concerns and closed control loop.

2.1.1 Key Concepts

Many concepts mentioned in this proposal are used in a number of different research areas. Therefore, the same concept can be referred under different terms, or the same term may refer to different concepts. Furthermore, many research areas employ adaptation and dynamic reconfiguration to address their particular needs. Each area focus on different aspects, thus, the use of adaptation is also different. In this section, the concepts of adaptation and context are clarified, in the scope of adaptive software systems.

2.1.1.1 Adaptation

The term *adaptation* is widely used in several domains, with slightly different meanings. In general, *software adaptation* concerns changing a software system with a purpose, in response to variations in its operational envelope, for instance, in the user needs, in the system workload or in the available resources. *Dynamic software adaptation* (also known as *runtime software adaptation*) imply that the intended changes take place during the system execution [MSKC04a]; there is no need to stop the system.

Self-adaptive systems, or simply *adaptive systems*, are systems that are able to adjust their behavior autonomously (i.e., without human intervention), in response to their perception of the environment and the system itself for a variety of goals [CLG+09]. Self-adaptive systems are also referred by more specific terms when self-adaptation exclusively addresses a specific type of goal. For instance if adaptation concerns dependability or performance, systems are said to be self-healing or self-optimizing. Systems or components are said to be *adaptable* if they provide mechanisms for being adapted during runtime but they do not monitor the environment and the control over the change in its behavior is left to other components.

Self-adaptive systems are sometimes referred to as *autonomic systems*. This is because self-adaptation is a means that has been explored to in the context of *autonomic computing* — a vision of the future in which software systems will manage themselves in accordance with

high-level objectives specified by humans [Kep05]. This self-management relies on four concerns: self-healing, self-protection, self-optimization, and self-configuration.

Two key elements in self-adaptive systems are the *context* and the *adaptation logic*. The former is the information regarding the execution environment, which has to be monitored due to its relevance to assess if the system behavior is suitable or has to be changed. The latter concerns the decisions that determine when to adapt and how.

2.1.1.2 Context

The notion of *context* and *context-awareness* was first introduced in computer science to improve the interaction between humans and computers. Context provides implicit situational information, as regular communication between humans would have and was lacking in human-computer interaction [Dey01]. In a broad sense, context can be understood as any information that characterizes the situation of an entity [Dey01]. In adaptive systems, this involves any information that can be used to detect the situations in which the system needs to adapt. For instance, context might include information that allows to evaluate if the system meets certain quality of service requirements (performance, availability, security, etc).

Context information can have different origins. Usually, it encompasses both fine-grained and coarse-grained information. The gathering of context information can be done in different ways: periodically, continuously, or on demand. Hence, key aspects in the development of *context-aware* systems [SAW94] are: (i) the identification of context sources, (ii) acquisition of information from sources, (iii) aggregation of context information, and (iv) analysis and publication of the analysis results.

2.1.1.3 Adaptation Logic

The *adaptation logic* of an adaptive system is what defines its adaptive behavior. This includes, on one hand, the characterization of the situations in which changes are needed and, on the other hand, the definition of what type of adaptation is required in each of these situations. In component-based software systems, adaptation is typically classified as *behavioral adaptation* or *structural adaptation* [MSKC04b]. Behavioral adaptation takes place if the system dynamically

changes its behavior without changing its structure. For instance, adjusts the value of some parameters to tune the system behavior or change some variable that determines the algorithm that is used. Structural adaptation occurs if the system's structure is modified at runtime (for instance, a software component is replaced with another component that has the same interface or, more generally, components and connectors are arbitrarily added or removed). While the variability supported by behavioral adaptation is fixed (in the sense that the number of variants becomes fixed at the time of the system construction), structural adaptation supports an unbounded number of variants because adaptation may introduce an element made available subsequently.

2.1.2 Design Principles for Self-adaptive Systems

Self-adaptation can be handled within the system, at the code level, for instance making use of programming language features such as exceptions, reflection, and aspects. However, the complexity and cost of developing and maintaining adaptive systems can become extremely high when ad-hoc solutions, defined on a per-system basis, are adopted. Considerable research has been done in the area of dynamic adaptation and self-adaptation in order to establish approaches that allow the construction and maintenance of adaptive systems in a cost-effective way. Among the key design principles that have emerged from this research, the most significant are: (i) the separation of the concerns of system functionality from the concerns of self-adaptation and (ii) explicit representation of a feedback control loop.

2.1.2.1 Separating the Adaptive Behavior

When the system's adaptation logic is scattered by several components, it becomes challenging to understand the outcome of adapting in a particular scenario, and reuse the components or the adaptations. Altogether, this scattering makes it hard to maintain and costly to modify.

The separation of concerns (SoC) paradigm [HL95] is a widely accepted solution to these issues, allowing a cost-effective construction of self-adaptive systems. The solution requires that all aspects that concern adaptation are extracted from the base system and treated separately from the system [CGS05]. To some extent, self-adaptation is made external to the base system,

which, for this reason, is also referred to as the managed system. By applying this principle it is possible to: (i) decrease the complexity of the software development, (ii) facilitate comprehension, and (iii) promote reuse of adaptation solutions.

2.1.2.2 Closed Control Loop Systems

A number of recently developed approaches implement the separation and externalization of the adaptive behavior in terms of a control layer on top of the managed system. This layer monitors the base system, possibly maintaining an explicit model of the system, and, relying on a set of high-level goals, adapts the behavior or structure of the system. In many approaches it requires the insertion of probes in the base system (for instance, to detect specific system events) and effectors that can perform a specific set of adjustments. This technique can be applied to both recently built or legacy systems, and further facilitates the reuse across different systems, reducing the cost of developing new self-adaptive systems.

Many approaches rely on a particularly popular external control mechanism reminiscent from the classical control theory: the closed control loop [SEM03], depicted in Figure 2.1. In a closed control loop system, a sensor monitors the output and feeds the data to a controller which continuously adjusts the input as necessary to keep the error to a minimum or to maintain a goal. Feedback on how the system is actually performing allows the controller to dynamically compensate for disturbances to the system. An ideal feedback control system cancels out all errors.

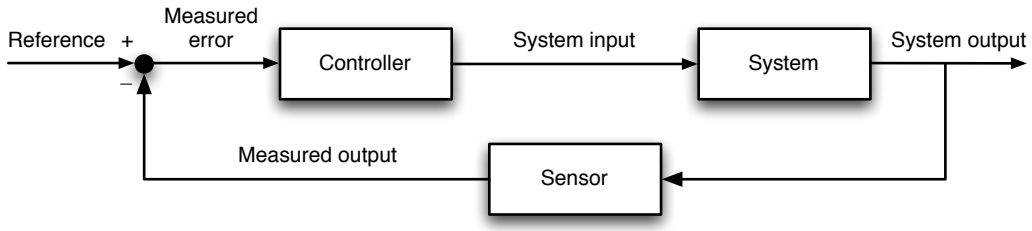


Figure 2.1: Closed control loop system

In adaptive systems, the control loop consists of mechanisms that monitor the system, reflect on observations for problems (check if the observed behavior meets the system requirements), and control the system to maintain it within acceptable bounds of behavior [SEM03]. Furthermore,

to reflect on observations, the external control system requires an explicit model of the system being maintained, to be used as a basis for adapting the system [OGT+99].

A principle that has been strongly advocated is that control loops are elevated to first-class entities in the development of self-adaptive systems [CLG+09]. That is to say, the feedback control loop needs to be an explicit and visible element of the system in its modeling, design, and implementation.

From adaptive systems, several computation paradigms emerged, using specific variations of the closed control control. From these paradigms, one of the most recent is *autonomic computing* [KC03b]. The goal of autonomic computing is to develop computer systems that are capable of self-management, in an attempt to address the growing complexity of software systems and the dynamism of the environment and load. An autonomic system monitors itself and decides the best adaptations to the current conditions, thus being able of four self-management tasks: self-configuration, self-healing, self-optimization, and self-protection. Self-configuration automatically adapts the system and components configuration according to the changing conditions. The self-healing task aims at the automatic discovery and correction of faults. The self-optimization task adapts resources to ensure optimal functioning and performance according to the defined requirements. Finally, the self-protection task is concerned with the proactive identification and protection from arbitrary attacks.

Autonomic systems employ a variation of the closed control loop with four activities as depicted in Figure 2.2. This control loop is often called MAPE-K (monitor, analyze, plan, execute, and knowledge) loop [HM08]. The *monitoring activity* (M) is responsible for gathering the context information regarding the execution environment, the system itself, and any other information considered relevant for adaptation. This is done relying on sensors. The *analysis activity* (A) analyzes and interprets the collected information according to the system model, in order to detect deviations from the desired system behavior. The *planning activity* (P) uses the information provided by the monitoring and analysis activities to decide which actions must be performed to return the system back to an acceptable state. Finally, the *execution activity* (E) performs the system adaptation, applying the actions decided in the previous phase through effectors. The division in several activities enables the independent modification and extension

of each one. The MAPE-K loop relies on knowledge (K) regarding the system and the system adaptation to perform all these activities.

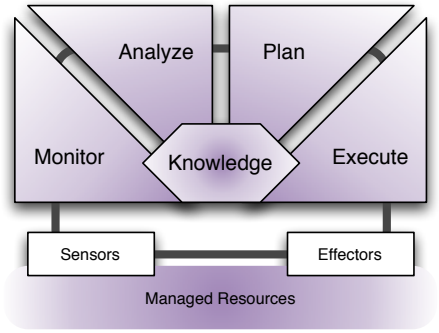


Figure 2.2: Autonomic manager control loop

Overall, employing a feedback control system as an external control mechanism to support adaptation results in an infrastructure that is highly reusable and easy to modify, whenever changes to the adaptation support are necessary. These characteristics are paramount when designing and implementing adaptive systems.

2.2 State of the Art

This section gives an overview of the state of the art for adaptive systems. Due to the sheer volume of work that employs dynamic reconfiguration, self-adaptation, and other self-adaptive solutions, we will mainly focus on approaches in which self-adaption is built externally in a control layer, separated from the other concerns, and relies on an explicit closed control loop.

In this class of approaches there are several critical design issues. One is the choice of type of system model. The system model is used by the external control loop system to detect changes in the context and select adaptation strategies. Another is the choice of model for the adaptive behavior, which allows to express the adaptation logic. Another critical choice is the type of adaptation support offered to the control layer, responsible for executing adaptation and providing updated information on the system and execution environment. Finally, the last critical choice is the support for executing adaptation.

2.2.1 System Model

As mentioned previously, in approaches to self-adaptation that rely on feedback control loops, systems maintain a model of themselves, useful to provide information to detect changes in the behavior, but also indispensable to help select an adequate adaptation strategy. The choice of model type is intrinsically connected with the type of adaptations that are expected to be supported. For example, a component-based system may be adapted by adding or removing components, and by changing links between components. To perform such adaptations, the system model must capture the components that are currently in use in the system and how they are connected.

This type of system model is known as an *architectural model*: it represents an abstract view of the system as a composition of computational elements and their interconnections [SG96]. *Architecture-based approaches* to self-adaptation rely on this type of system model. Adaptation actions in these approaches are typically restricted to operations that can change the structure of the system architecture: additions, removals, replacements, and (dis)connection of architectural elements. When behavioral adaptation is needed, for instance, tuning core parameters of the system, the system model must capture which parameters are available and their current value.

Taylor and colleagues [OMT98, OGT+99][3] demonstrated the beneficial role of an explicit architectural model fielded with the system and used as the basis for runtime change. In a chapter devoted to self-managed systems in *Future of Software Engineering 2007*, Kramer and Magee also state that architectural models seem to provide the required level of abstraction and generality to deal with the challenges posed by self-adaptation [KM07]. In the last years, several architectural-based approaches to self-adaptation have been proposed. Some more prominent examples are briefly described below.

- *Rainbow framework* [GCH+04] — it offers structural self-adaptation, maintaining an architectural model of the system and mechanisms to update the model according to changes to the system. The approach relies on a mapping between model elements and system-level elements. A detailed description of this framework is given in Section 2.2.3.2.

[3] The first work was considered the most influential paper of ICSE98

- *Mobility and Adaptation Enabling Middleware (MADAM)* [FHS+06] — a reference architecture and middleware for mobile computing applications that supports flexible context monitoring, adaptation planning and dynamic reconfiguration through the change between different variants for the same variation point.

- *Three-layer reference model* [KM07, SHMK08] — a reference architecture that relies on a three-layer model for adaptable software architectures and task synthesis from high-level goals. The *top layer* generates plans[4] from high-level goals. The *middle layer* constructs component configurations from plans and executes those configurations. Finally, the *bottom layer* includes the components implementation.

- *Policy-based architectural management (PBAAM)* [GT09, GvdHT09] — an approach to self-adaptation that uses architectural models as runtime artifacts, adopts a policy-based model of adaptive behavior that can be modified at runtime and provides explicit support for the recording and visualization of adaptations in order to help the operators and designers.

.

2.2.2 Models of Adaptive Behavior

In approaches to self-adaptation that separate the adaptation behavior from non-adaptive behavior, an important decision is the choice of the type of model of adaptive behavior that is considered. Approaches that focus on structural adaptation often opt for graph-based models. In this case, adaptation boils down to graph rewriting, and can be expressed, for instance, using graph transformation rules [BLMT08, TGM99, HIM00], graph grammars [LM98] or simply programming scripts for graph manipulation [WLF01, GCH+04] (also called repair strategies). There is also a number of approaches that use process-based models [ADG98, Oqu04] and others that use *state-based models* [ZC06, DHPB03]. For instance, the *AutoTune* [DHPB03] agent framework employs state-based models to uniquely adapt a set of system parameters. These models are obtained from equations that relate system properties with its parameters and that

[4] A plan specifies which actions will lead the current state to the goal state, in the form of condition-action rules.

allow to determine which parameters' values are suited to a particular situation. The equations are determined in an experimental manner, before system execution. The authors illustrate the approach using a case study of web applications performance. The goal is to maintain two metrics, the CPU use and the memory use, below certain thresholds. The adaptable parameters are the *timeout* and the *keepalive* of Apache web server. By adapting these parameters, it is possible to increase/decrease the use of CPU and memory. The system monitors continuously both metrics and they are input to the equations, which output the parameters' values.

In the aforementioned approaches, the modeling of adaptive behavior is conducted at a relatively low-level of abstraction. In particular, when the aim is to use self-adaptation in the context of autonomic computing, as pointed out in [Kep05], it is essential that humans can express their goals to the systems in an easy way, at an adequate level of abstraction. as pointed out in [Slo94], policies are considered a better choice than the models referred previously.

The advantages of policy-based adaptations result from their declarative nature (in contrast with operational nature of the other referred models). This facilitates the understanding of the adaptation logic of the system by an operator, without requiring fully detailed knowledge of implementations. The use of policies allows to achieve independence from the current system state, facilitating and reducing development effort and subsequent tuning of the system's adaptive behavior. Next, are addressed the two main types of policies used for expressing adaptation: *action-based* and *goal-based*.

2.2.2.1 Action Policies

Action policies are declarative situation-action rules that express the actions that should be performed when given conditions are satisfied. In the context of self-adaptive systems, they are used to govern the adaptation of the managed system. Each rule indicates the actions that should be taken in response to events (or in particular states) that indicate the need of adaptation. Due to the simplicity of specification it is common for approaches to rely on their own format to specify action policies. The structure of these policies ranges from a simple if-then-else format [KC03a, MB11] to a more sophisticated event-condition-action (ECA) rules [MD89, KW04, AST09].

There were several approaches to define policy languages [KKK96, LCJS01, DDLS01, Ant06]. Among existing policy languages, the Ponder [DDLS01] languages has achieved a renowned position by its completeness and comprehensive support. The language, primarily offering different formats to describe access control rules, has been extensively used for adaptation purposes [MLS04, FLR10]. The language allows to specify different behaviors for each system object. A rule determines how to choose the best behavior for the current scenario. Each rule is triggered by a specific event and includes a condition clause that must be respected and the respective action that must be performed. Events signal new circumstances, changes, that can be either internal to the system or external, which must be addressed. For example, in a website, if a login attempt fails the user/password tuple three times, an event is generated. Conditions are predicates that must be evaluated to determine if the rule applies or not, and which action will be chosen. The actions describe the adaptations that will change the behavior. Other policy languages exist that target specific adaptation goals, for example, targeting network management [SK05] or dynamic provisioning of resources [ZJY+09].

PBAAM, already mentioned, is an example of an approach to self-adaptation that uses action policies to specify how the structure changes. They consist of a set of observations and a list of responses; when the list of observations is fully satisfied, the entire set of responses is enacted. To facilitate the understanding, they may contain a human readable textual description that indicates the purpose of the policy to operators.

Action-based policies present some drawbacks that can be hard to tackle. The major drawback is probably the amount of knowledge that an operator has to gather before being able to describe the system adaptive behavior in terms of situation-action rules. This requires policy makers to be intimately familiar with low-level details of system function. Also, the larger the number of adaptations, the more complex and error-prone becomes the task of specifying the policy. Another drawback is that, if there are many elements of the context that can change independently, the number of rules necessary to describe the adaptive behavior can quickly become very large and complex to be manipulated by humans. Namely, it may become impossible for a human to predict the combined effect of potential conflicting rules. Finally, if the adaptive behavior changes or further context information is added, the policy may have to be written

from scratch, with few reuse options.

2.2.2.2 Goal Policies

To address the drawbacks of action policies in some contexts, some approaches employ *high-level* or *abstract* goals [AHW04, BCGZ06, SHMK08, HSMK09]. The idea is to support the definition of how the system should behave without full understanding of what the system can do, leaving to the approach to determine the actions required to achieve the goals.

Goal policies can be specified in numerous ways. In the three-layer reference model [SHMK08, HSMK09], a goal policy is expressed as a set of formulas, using temporal logic. These goals, together with a description of the system capabilities, are used to generate action policies to enforce the goals. This generation relies on identifying all the states from which it is possible to lead the system to a correct state, thus creating a rule for each undesired but amendable state. Other approaches maintain action policies and rely on a mechanism to map goal policies to action policies [BLMR04]. A number of approaches rely on the adaptation impact estimation to generate action policies[BB08]. Reinforcement learning and heavy computations allow to estimate the results of executing an adaptation in a certain state, thus selecting or not an adaptation to address a state. Reinforcement learning allows the adaptation logic to gather feedback on the impacts of performing an adaptation, and use that same information in future estimations.

Another approach to the specification of goal policies is to use utility functions [TK04, FHS$^+$06, CGS06]. The idea in this case is to define the utility (a scalar value) of each possible system configuration as a function of specific data available in the context (e.g., memory and bandwidth available). The aim is to assemble a configuration tailored to the current situation. For instance, in MADAM, a goal policy is expressed in terms of an utility function that assigns a scalar value to each possible system variant, as a function of the system properties in a given context. The choice of a system variant when the system needs to be adapted relies on property predictor functions over that system properties that are associated to each system variant.

Recent work by Salehie and Tahvildari [ST12, ST07] proposes an approach that employs a variation of utility functions. The proposed approach uses a weighted voting mechanism to

select the adaptations. The mechanism relies in the priority of the goals in the policy, given by a priority vector, and in a voting schema. The goal policy encompasses both high-level goals and more low-level goals, which are organized in a hierarchy. The leaf goals of the policy are mapped directly to adaptation actions. When a goal or several goals are violated, the voting mechanism receives the lists of preferred adaptation actions for each goal and aggregates and weights them according to the priority. This process allows to filter the lists and select the adaptations that will be used.

2.2.3 Adaptation Support

The adaptation support assists the adaptation process, by providing necessary input and by carrying out the output of the decision making. The input is a number of informations necessary to assess, according to the model of the adaptive behavior, which adaptation is necessary, if any. The output is the selected adaptation to correct the system behavior. The input provisioning overlaps with a monitoring and analysis task; the output implementation is addressed separately in Section 2.2.4.

The implementation of the adaptation support depends on many system aspects, namely, the components, the system model, the adaptations, and the monitored context information. There are many approaches that made contributions to the adaptation support. There are two frameworks of particular interest: Cactus and Rainbow. Cactus, while not as recent as Rainbow or other frameworks, is one that provides a very complete adaptation support, from monitoring to adaptation execution. Rainbow is one of the most well-known frameworks for self-management in software engineering. In the next sections, we cover the two frameworks and analyze how they provide adaptation support, and their main concerns while doing it.

2.2.3.1 The Cactus Framework

The *Cactus* framework [HSUW00] is a toolkit for the development of services and network protocols that can be adapted during runtime. This framework focus on fault-tolerance and survivability[5].

[5]A system's ability to continue regular execution even in face of attacks or failures.

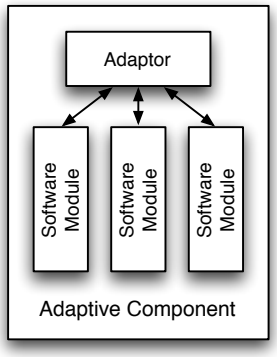


Figure 2.3: Adaptive Component in Cactus

The framework does not have an explicit system model. The system's building blocks are adaptable and distributed services and protocols. These building blocks are compositions of software modules, each providing a different property or function. The composition can be adapted by changing the execution parameters of a software module or exchange the software modules altogether, thus changing the service composition.

In terms of adaptation support, Cactus offers monitoring and analysis support and adaptation execution. This support relies on making the building blocks into adaptive services. An adaptive service is a set of adaptive components (ACs) (and occasional static components). An AC, depicted in Figure 2.3, is a collection of software modules, and a component adaptor that coordinates adaptations between the software modules. Each software module provides a different implementation of the component functionality, with the component adaptor switching between the alternative modules or changing the execution parameters of the component.

In terms of monitoring, Cactus allows the construction of monitors, as a collection of micro-protocols, to capture relevant context information, such as resource consumption, and report that information. The information can be handed over to component adaptors or to the user through a GUI.

In terms of execution, Cactus has to address several issues related with the *state* and the *distribution*. When switching software modules, it may be necessary to transfer state from the old module to the new one. Therefore, using service variables, Cactus allows to transfer state easily, since the service variables are available to any software module in the composition. Furthermore, if the service is distributed, adapting it may require further care. Cactus relies

on a three phase process to ensure the coordination of the different sites [CHS01]. The process begins with the *detection* phase where the component adaptor determines if the current module is the best, and, if not, which one is. This is done using fitness functions that take the current system state as input. The next phase is the *agreement* between sites to adapt, where all hosts reach a conclusion if it is necessary to adapt and how. This is achieve using consensus. Finally, the last phase is *action*, where a graceful adaptation from one module to another is performed. These concerns may not apply when changing the service parameters.

Overall, the Cactus framework provides a rich adaptation support for distributed services and network protocols change. The system model represents the system in terms of its components and allows to change parameters and exchange modules. The framework employs a control loop where a number of monitors control the context information and component adaptors perform adaptations, taking care of state transfer and coordination. However, although there is a separation of concerns between the building block functionality and the adaptation concerns, the adaptation logic is together with the adaptation execution, in the component adaptors. This may raise a number of difficulties when it is necessary to change the adaptive behavior. Furthermore, Cactus also suffers from the issues related with scattering, discussed in Section 2.1.2.1.

2.2.3.2 The Rainbow Framework

The *Rainbow* framework [GCH+04] provides support to the self-adaptation of software systems. This framework focus on providing a reusable infrastructure and low maintenance and development costs. The framework follows an abstract architectural system model and the supported adaptations are of the entire responsibility of the system developer.

In terms of adaptation support, Rainbow implements a closed control loop system. To make this control loop reusable by different software systems, the entire adaptation support is external to the system. This degree of separation is far greater in comparison with the Cactus framework.

The adaptation infrastructure is divided in three different elements, as depicted in Figure 2.4. The *system* layer establishes the connection between the infrastructure and the system, in other words, it is the system access interface. This layer encompasses sensors (probes) to monitor the

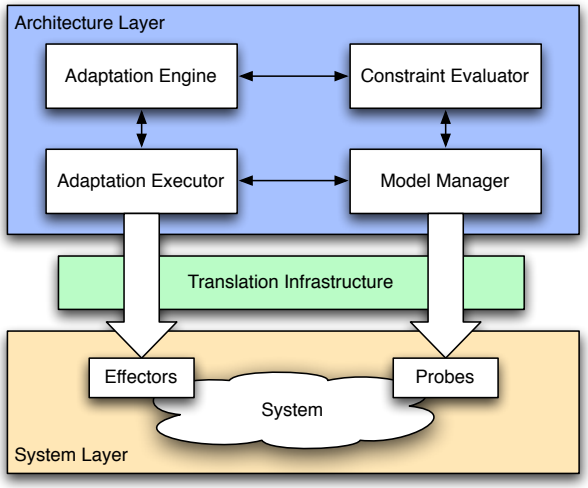


Figure 2.4: Adaptation infrastructure in Rainbow

system and effectors to execute adaptations to the system. The *architecture* layer is responsible for the analysis and planning tasks. This layer has a model manager that gathers information from all sensors and interprets it and updates the system model representation. Furthermore, a constraint evaluator checks the system state to detect anomalies. When anomalies are detected the adaptation engine determines the adaptation action and the conducts its implementation. Finally, the *translation* infrastructure mediates the mapping of information across the abstraction gap from the system to the model. Furthermore, the infrastructure maintains a repository of the mappings from architectural-level elements to system-level elements and vice-versa.

The adaptation infrastructure depends on system-specific adaptation knowledge. The adaptation knowledge is in fact the adaptation logic, which includes the system operational model. The operational model imposes explicit constraints to the system behavior, defining, among other elements, adaptation strategies. Adaptation strategies define which actions are possible and how they should be implemented for particular system concerns. Rainbow relies on architectural styles to improve the reusability of this knowledge. Architectural styles can be used by different systems that share the same structural and semantic properties. Allied with adaptation strategies, it is possible to define an adaptation style which allows to adapt any similar systems that share the same system concerns.

Overall, the Rainbow framework provides adaptation support that can be reused by different systems to attain self-adaptation. The entire support is external to the system, requiring that the

system provides an access interface. There is a clear separation between the adaptation logic and the adaptation support. The description of adaptation actions also includes its implementation, since the framework does not support generic adaptations.

2.2.4 Adaptation Execution

There are many concerns affecting the execution of adaptations. Some refer to the component and system consistency and disruption, and adaptation cost. The execution of an adaptation also depends on several aspects. The type of adaptation may make a difference on how an adaptation is performed in system. The target component is also important, as the execution of the same adaptation may be different for distinct components. The operation and guarantees that a component or system provide may also determine how an adaptation is executed. Other components that are co-dependent of the component being adapted, or that are affected by the adaptation may also be critical for the execution. In the remaining of this section, we will refer to the execution of an adaptation as a reconfiguration, to distinguish it from the actual adaptation.

Early work in architecture-based adaptation [MK96, ADG98] executed structural adaptations simply by directly changing the system configuration. This was achieved by adding new components, connecting or disconnecting ports, without any concerns regarding the component and the system. However, this may not be adequate when we remove a component that is currently interacting with another component, the entire interaction is compromised and the component left in the system may become inconsistent.

Further work by Kramer and Magee [KM98] addressed consistency when reconfiguring components. Their work demands *quiescent* components before executing a removal (or an exchange) adaptation. Quiescent components must not be engaged in any ongoing interaction with other components and that will not start any new interaction. Many times, a component to become quiescent depends on other components that communicate with it. In order to achieve a quiescent state, the target node needs these components to stop communicating with it. These components are called *passive*, and only when they stop communicating with the target node it can reach quiescence and be disconnected and removed. The passive components are also up-

dated in terms of connections, removing all the connections to the removed component. However, the assumptions they rely on, make the approach non-viable when the target component needs to perform a service request to one of the passive components to become quiescent. The work proposed by Moazami-Goudarzi [MGK96] makes the set of passive components grow dynamically, by blocking the passive components only when no service is required from them. Thus, gradually but definitely, quiescence is obtained. The work by Vandewoude *et al* [VEBD07], also based on not as strong assumptions to achieve quiescence, provides another solution if the removal is actually part of replacing a component. The proposed *tranquility* (instead of quiescence) consists on stopping only the component to be removed and the passive components can continue operation. In this solution, the component is removed when all passive nodes commit on not using the target component, despite continuing operation. However, there in some cases the quiescence will never be reached.

One of the problems with the proposed solutions is that they do not address distributed components. The distribution elevates the complexity of achieving a quiescent state because all the instances of the component must becomes quiescent. Another issue is that the simple solutions used to tackle consistency come at the expense of a high disruption level, namely when replacing components. In systems with both strong consistency needs and very high throughputs, such disruptions may cause the system to fail. Communication protocols are examples of scenarios where the consistency guarantees are strong to avoid message lost but at the same time, any service disruption causes messages to be delayed. It is important to note that the lowest disruption solutions are achieved with when tailored for the particular services.

The switching of protocols may be more or less complex, depending on the guarantees offered by the protocols. Protocols that execute in several different nodes are, often, more complex to switch, since (commonly) all the nodes have to agree on switching at the same time. Protocols that have state are also more complex to switch, because a quiescent state may be necessary before the switch, as well as state transfer from the old protocol to the new one may be required. A protocol that has state and executes in several nodes is the total order, and is frequently used as a case study for switching mechanisms [CHS01, MR06, LvRB$^+$01]. A total ordering protocol ensures that all members of a group receive all messages sent to the group in the same (total)

order. The total order case study allows to verify if the guarantees are maintained before, during and after the switching. In the next section we describe in more detail a the mechanisms used to replace communication protocols: one for any communication protocol and the other tailored for a total order guarantee that has both high consistency and minimum disruption demands.

2.2.4.1 Cactus Switching Mechanism

The Cactus framework, addressed in Section 2.2.3.1, includes a mechanism to switch between micro-protocols during runtime [CHS01]. This mechanism intends to be valid for any micro-protocol substitution.

The adaptation process is a three phase model. The first phase relies on detecting changes in the execution environment, and if it would be beneficial to adapt or not and how. The responsibility for detecting a change depends on the implementation and type of application. Any micro-protocol can detect changes. The adaptation decision is made using information regarding the state and the fitness functions associated to each adaptation-aware algorithm module (AAM). The AAM determines which is the best behavior for the current scenario. The second phase is the agreement process in which all the adaptive components reach consensus regarding the need to adapt or not. Finally, the third phase is the adaptation action itself, implemented through a graceful adaptation process that implement barriers to achieve synchronism.

Figure 2.5 illustrates the elements involved in the switching mechanism. The entire mechanism is controlled by an adaptive component (AC) that includes an adaptor module, which includes three micro-protocols specifically for controlling the adaptation (*adaptor*), agreeing when to adapt (*consensus*), and executing the adaptation (*barrier*). The AC also includes several AAMs.

To illustrate how the switching mechanism operates, let us consider the exchange of different total order micro-protocols: $TO1$ to $TO2$ at the same time in all the adaptive components. In the given example, $TO1$ is a sequencer-based total order protocol [KT91], while $TO2$ is a token-based total order protocol [CM84]. The group member holding the sequencer/token starts the switching process.

The sequencer holder sends a message to all participating ACs, so that they are prepared

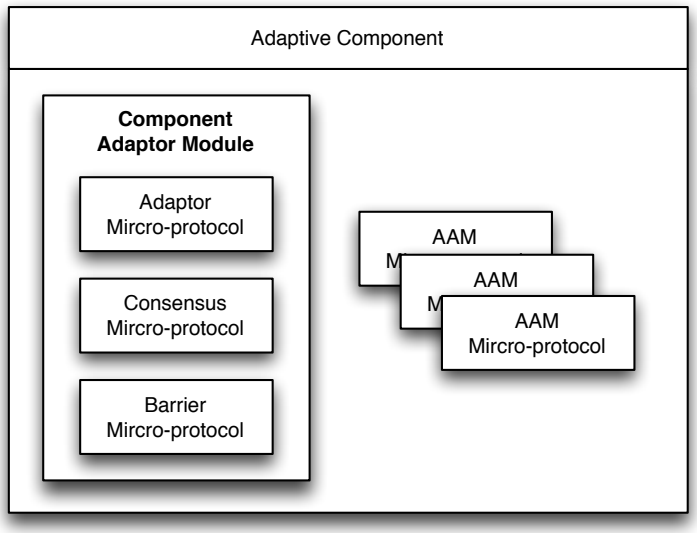


Figure 2.5: The Cactus switching mechanism

to receive incoming messages in $TO2$, besides control messages sent through $TO1$ that help the switching process. After all the ACs are prepared, the sequencer sends a control message, so that each AC allows $TO2$ to start processing outgoing messages (messages sent to the group). At this point, messages from $TO1$ and $TO2$ are flowing at the same time, and they are delivered to the corresponding protocol. After a while, only the $TO2$ processes incoming messages (messages received).

The mechanism also allows to transfer state after the $TO2$ initial preparation. The state may include attribute values, unprocessed messages, history of processed messages, among many others.

Overall, the Cactus switching mechanism is managed by a coordinator. The mechanism offers non-disruptive switching that relies on Cactus components and shared information between micro-protocols in the same adaptive component.

2.2.4.2 Appia Switching Mechanism

This mechanism [MR06] targets the *Appia* protocol composition framework [MPR01]. The *Appia* framework allows to develop protocols in a modular manner. Different protocols are then organized in a stack to offer tailored communication services. The mechanism relies on a

monolithic multiplexer tailored to total order switching. This multiplexer is located above the protocols to switch and depends on Appia ability to handle multiple channels at the same time, with shared protocols, as depicted in Figure 2.6. This work switches between a sequencer-site total order protocol ($TO1$) [KT91] and a symmetric total order protocol ($TO2$) [MR06].

The process begins with a synchronization point, when all the nodes start sending messages in $TO2$. The first message sent through $TO2$ is flagged, or in the absence of a message, a null message is sent. At this point there are messages being sent and received from $TO1$ and $TO2$, being delivered to the corresponding protocols. The messages delivered to $TO2$ are buffered by the protocol, while the ones delivered to $TO1$ are processed immediately. When the flagged messages from all nodes are received, the buffered messages in $TO2$ are delivered (if they were not already delivered by $TO1$), and all messages received by $TO1$ are discarded. After this point, all the messages are processed by $TO2$.

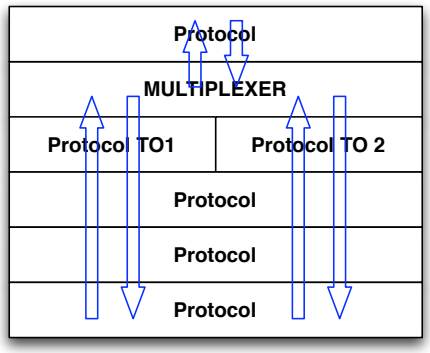


Figure 2.6: The Appia switching mechanism

Overall, the Appia switching mechanism does not require a coordinator, only someone to initiate the process. The mechanism offers a non-disruptive approach that relies on the buffering ability of protocols and shared sessions in different channels in Appia.

2.2.4.3 Generic Switching Protocol

In [LvRB+01] a different switching protocol is proposed. This mechanism aims at protocol composition frameworks, such as Ensemble [vRBH+98] or Appia [MPR01], where there is a vertical composition of protocols in stacks. The communication relies on events that flow in up

and down directions in the stack and between the different nodes with stacks. The mechanism is generic, thus, it can be used for any protocol. The switching relies on two specialized protocols, a *switch* and a *multiplexer*. The first is located above the protocols to switch, while the second sits below, encapsulating both the switching protocols, as well as the protocols that will be exchanged, as depicted in Figure 2.7. Both *switch* and *multiplexer* control the event flow, they determine to which protocol the events are delivered, and maintain transparency to the remaining protocols in the stack. The description of this switching protocol will use the total order case study, where a $TO1$ protocol is switched by a $TO2$ protocol.

The switching process relies on an elected coordinator that sends the control messages to the group. It starts with a *prepare* message sent to all members. Every member replies with an *okay* message that carries the member identification and the number of messages sent so far by $TO1$. From this point on, every new message is sent using $TO2$. If a message is received for $TO2$, it is buffered during this phase. When a member starts sending messages in $TO2$, a reply is sent to the coordinator. After collecting the acks from every member, the coordinator sends a new message *switch* that carries a vector containing the number of messages sent by each member. This information allows a member to know when all the messages from all the nodes sent through $TO1$ are received. At this point, the member can deliver all the buffered messages to $TO2$ and stop the $TO1$.

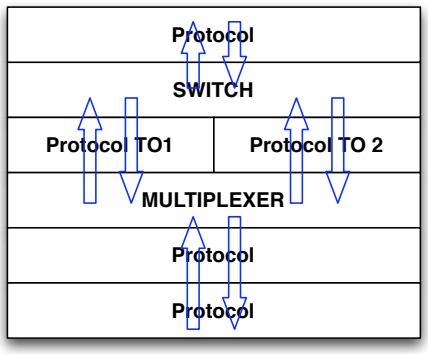


Figure 2.7: Protocol exchange using a switch and a multiplexer

Overall, this mechanism allows to extend any protocol composition framework, by adding a switch and a multiplexer to hide the switching from the remaining protocols. A buffering mechanism is necessary, although it is separated from the protocols, and can be the multiplexer itself.

This mechanism also relies on a coordinator. The end of the switching process is determined by each member on its own. This mechanism can perform worse than the previous one, if one of the nodes suffers from a slow connection or other type of delay. This results in delaying the entire process, because all nodes have to wait also. On the other hand, the messages are sent in duplicate by both channels in the previous mechanism.

2.2.5 Limitations and Challenges

The state of art discusses three main concerns when developing self-adaptive systems. In software systems built from adaptable components, these concerns raise some challenges and point out the limitations of some design choices. In terms of system model, the choice is determined by the type of adaptation that must be supported. In the literature, one can find a clear distinction between structural and behavioral adaptations. However, in the systems targeted by this work, both types of adaptation are complementary. Whilst system models continue to be developed for only one type of behavior, it is becomes necessary to use two distinct models in not the most efficient of manners. Therefore, the challenge here is to make both architectural and behavioral models coexist harmoniously, in a single model that allows both structural and behavioral adaptations.

The second choice is the type of model used to describe the system's adaptive behavior. In the targeted systems, the complexity resulting from the large number of components, their adaptations and the impacts of the later on the global system make a strong push towards the separation of systems' behavior, namely, the adaptive behavior. Action policy models allow the necessary separation but their low-level declarative nature may be a constraint in the targeted system. With a large number of components, it is expectable that the number of possible adaptations also rises. The larger is the number of rules, the harder it becomes to declare new rules and identify any conflicts between rules. It also becomes more complex and difficult to guarantee that the specified rules in fact translate the desired adaptive behavior. On the other hand, the goal policy models in the literature may help with the complexity of the system but they still demands some sort of mapping between the high-level goals and the adaptations that must be performed. This mapping is done manually, thus, it can also be a tiresome and

error-prone task due to the large number of adaptations. The challenge here is to automate the error-prone and complex manual tasks, allowing the policy model to handle more components. This may be more feasible for the goal policies, as the manual mapping could be automated.

Finally, the third choice addresses the adaptation support. A fairly generic adaptation support, such as the closed control loop, depends on a series of sensors and effectors that must be tailored to each component and adaptation. In the target systems, the large number of adaptable components demands an also large number of sensors and effectors. Furthermore, due to the distributed nature of many components, the monitor phase must be able to gather information from different nodes to characterize a component. The same happens with the execute phase that must support coordination mechanisms to perform the adaptation simultaneous in several component instances.

Overall, the current state of the art lacks mechanisms and methodologies to address the greater complexity of these systems. This is the key issue that needs to be overcame to efficiently support self-adaptation in these systems, as many of the approaches still heavily depend on the human operator to describe and manage all the aspects of the adaptation logic and its support.

Summary

This chapter introduces the concepts used in adaptive systems and provides an overview of the state of the art. In terms of concepts, it clarifies the terms *context*, *adaptation*, and *adaptation logic* terms, and covers some design principles for adaptive systems. In terms of state of the art, the overview addresses three key aspects: the system model, models of adaptive behavior and the execution of adaptations.

There's only one corner of the universe you can be certain of improving, and that's your own self.

Aldous Huxley, in *Time Must Have a Stop*

Chapter 3

Approach Overview

This chapter provides an overview of the conceptual frameworks proposed in this thesis. It begins with the description of a concrete example, that will motivate and illustrate the approach, and later will be used for its evaluation. After, the chapter provides a high level introduction to how to model the system and discusses the organization of the adaptive elements in different abstraction layers.

3.1 Running Example

The example application consists of a website that is subject to dynamic and unpredictable load, often facing high traffic. The website hosts an online shopping store, that allows users to register for an account, browse the products catalog, and perform online orders. The users can either be *home* or *business* clients, with distinct contents being served. The website is deployed in a clustered-based architecture, i.e., a web cluster, as shown in Figure 3.1.

In general, the clients make requests to a virtual IP, served by a front-end that acts as a load balancer, distributing the requests among the available servers. Cluster nodes may receive and process any request from any client because they are not specialized. Each server node of the cluster has its own web server. Web servers do not interact directly with each other. Instead, they have access to shared state using a distributed in-memory caching service, maintained collectively by all server nodes. The distributed cache is used to store results from

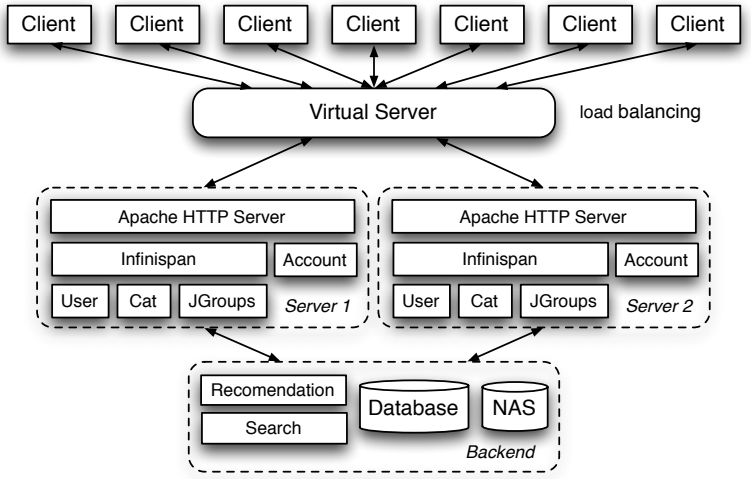


Figure 3.1: The running example

recent requests. Therefore, when a server receives a request, it first checks if the request can be served from the distributed cache. If this is not the case, then it forwards the request to the local software component that is capable of retrieving or generate the webpage. The cache is never used to store sensitive information, therefore, some requests are forwarded directly to the corresponding software component. Finally, all nodes have access to a shared persistent store, implemented by a database and additional network attached storage (NAS). These are part of the *backend*.

The website content is served by three main software components installed in the web servers deployed at each cluster node. The *Catalog* component (abbreviated by *Cat* in Figure 3.1) handles static content, such as product webpages. The *Account* component handles sensitive content, such as credit card information or the user's account password. Finally, the *User* component handles dynamically generated content, which is customized to the user, such as product recommendations and customized searches. For that purpose, it relies on two centralized components. The *Recommendation* engine generates recommendations for a particular user and the *Search* engine gathers the results of a search also considering the user. These two components operate in the backend and can be accessed by any server. Furthermore, the *Catalog*, *Account*, and *User* components are separated in *business* and *home* to cater to the different types of users.

The system runs in several nodes. Each client executes in a different node. The virtual

server executes in another node. A pool of server nodes execute the web server, an instance of the in-memory caching service, the multicast communication service required by the cache, and the *Catalog*, *User* and *Account* components. Each server runs Linux OS and Apache HTTP server [Apa] and uses the RedHat *Infinispan* [Inf] as the distributed in-memory caching tool and *JGroups* [JGr] as the multicast service. The backend executes in a separate node. Finally, there is a pool of free nodes that can be used to dynamically increase the number of servers.

The case study offers many opportunities for self-adaptation. The system performance is tied to the workload, and is severely affected by overload. The most obvious adaptation is to have an elastic number of active servers, in response to changes in the workload. In periods of more load, more free servers can be activated, while in periods of less load some servers can be assigned to other tasks or switched off for power saving. When the pool of free nodes is depleted, alternative configurations of the software components can be used to further increase the system capacity and avoid overload. For instance, reducing the resolution of any media content in webpages helps down-sizing the webpages, allowing gains in terms of resource consumption and processing time. Another possibility is to reduce the freshness of the recommendations and search results, which can be achieved by fetching the last generated recommendations/search results instead of fresh ones. The system performance may also be negatively affected by update hot-spots, when several users try to concurrently access and update the same content, which may result in conflicts when accessing the cache entries. One solution is the activation (or inactivation) of a mechanism able to totally order concurrent requests to the cache, preventing deadlocks at the expense of higher network utilization.

3.2 Components, Nodes, and Instances

The proposed approach targets distributed systems composed of multiple nodes, executing multiple software components. The key elements of the system are the *components*, which interact with each other in a manner determined by the underlying architectural style. Components can be *non-distributed* or *distributed*. Non-distributed components have a single *instance* and execute in a single *node*. Distributed components have multiple instances, each executing in a different node. When a component is distributed, its service is provided cooperatively by the

multiple instances, that interact among each other; such interactions are internal to the distributed component. Components are typed and multiple components of the same type may execute in the system. The approach imposes the following restriction: two or more components of the same type may execute simultaneously in the system only if they are non-distributed and deployed on different nodes. As illustrated in Figure 3.2, a system is composed of several nodes, each node may include both distributed and non-distributed elements. A distributed component is a set of instances of the same component type.

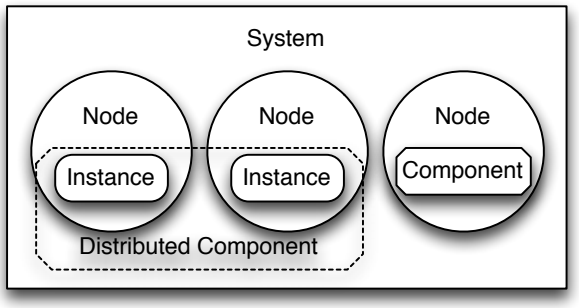


Figure 3.2: Distribution taxonomy

Consider the running example of Section 3.1. Each instance of the Apache HTTP server is independent of every other instance; these instances do not communicate directly and are not aware of each other, therefore, they are modeled as different non-distributed components (of the same type). Contrasting, all instances of *Infinispan* cooperate to maintain shared state. When an item is written or read from the cache, the instances engage in some protocol to distribute and replicate data items among them, while ensuring some target consistency criteria. Thus, all instances are considered to be part of a distributed component. In the running example, *Infinispan* together with *JGroups* are distributed components, while all the remaining components are non-distributed.

The components in the example interact with each other according to a fixed architectural style. Each Apache HTTP server interacts with an instance of *Infinispan* and the *Account* component. *Infinispan* also interacts with the *User* and *Catalog* components, and *JGroups*. The *User* component interacts with the *Recommendation* and *Search* engines. The instances of the distributed components interact internally. This is not shown in the graphical representation

of the system architecture.

3.3 System Adaptation

As mentioned in the previous chapter, a system can be adapted in many ways. Considering the high level model described above, the approach assumes that a system may be adapted as follows:

- *Addition of a component*: Components may be added to the system. If a component is non-distributed, the component needs to be created at a given target node (where no instance of that type already exists). If the component is distributed, one needs to specify the number of initial instances and the nodes where each instance is deployed.
- *Addition of an instance*: When a component is distributed, it may be possible to add new instances, by creating an instance in a node where no such instance exists.
- *Removal of a component*: Components may be removed from the system. When a distributed component is removed, all instances are removed.
- *Removal of an instance*: When a component is distributed, it is possible to remove only a designated instance or instances, instead of all. When the last instance is removed, the component is implicitly removed.
- *Exchange a component for another compatible component*: It is possible to have multiple components that provide the same service, each corresponding to a different implementation of that service. In some cases, one implementation may be obviously better than another (an upgrade or a bug fix), but it is also possible to have implementations that materialize different tradeoffs between service quality and resource consumption. In such cases, one may adapt the system by replacing one component for another component, in responses to changes in the workload or in the available resources. If the component is distributed, all the instances will be exchanged.
- *Exchange an instance for another compatible instance*: When a component is distributed, it is possible to exchange only a specific instance. This assumes that all the instances

continue to be compatible and able to execute together.

- *Configure parameters*: We assume that each component, node, or the system may have parameters that control its behavior. Furthermore, the values of these parameters can be changed during runtime. When it is a distributed component's parameter, we further distinguish parameters that are *local*, affecting a single instance, and *global*, affecting all instances of the component.

3.4 System Representation

To adapt a system it is necessary some representation of the system to reflect on. The system representation must have all the necessary information for the self-management but abstract the unnecessary information. Given the assumptions discussed in the previous sections, the proposed approach depends on information regarding the system configuration. The relevant aspects of the system configuration to represent the system are the following:

- Which components are executing at a given time, and in which nodes they are deployed. For distributed components, how many instances exist and in which nodes these instances are deployed;
- The current values of the configurable parameters of all deployed components, nodes and the system.

Thus, the system representation must include information regarding the nodes, components and the system. The representation of a node includes the node type and the value of any configurable properties. It also contains the types of all components deployed in that node. The representation of a component includes the component type and the values of any configurable parameters. The system representation is concluded in the values of any system parameters and properties. While the representation of components often includes the interactions with other components, the proposed approach does not require this information because we do not adapt connectors between components. Finally, the system configuration only includes active elements. A node is active if it has at least one active component. A component is active if it

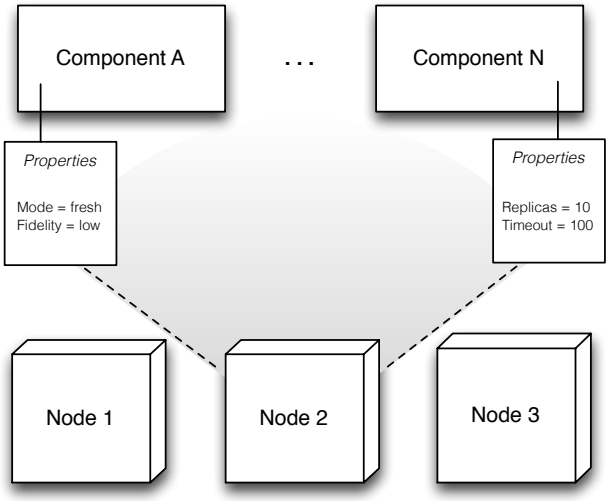


Figure 3.3: Representation of the system configuration

is ready for execution.

3.5 Adaptation Logic and Adaptation Support

The proposed approach relies on adaptation policies to describe the adaptation logic of the system. The rationale for this decision is as follows. First, adaptation policies offer better usability to the system designer, when compared to other solutions; their declarative nature allows easier description and comprehension. Second, a number of systems have been built resorting to policies to specify adaptations [GCH+04, KC03a] and we leverage on their experience. Finally, adaptation policies allow to specify both simple adaptation logics as well as sophisticated adaptive behaviors. Thus, the approach supports two types of planning: a *rule-oriented planning* for simple adaptation logics and a *goal-oriented planning* for more complex logics.

Our approach follows the closed control loop introduced in Chapter 2. We considered it to be the most adequate type of support for two main reasons. One is the clear separation between the monitoring and adaptation, making it possible to support different adaptive behaviors using the same infra-structure and elements. The other reason is that it promotes the reuse of several adaptation mechanisms for different managed systems (in the best case, only the sensors and effectors need to be specifically designed and developed for each system). Also, by modeling

the control component as a logically centralized entity, the management of distribution becomes simpler.

As described in Chapter 2, the control loop consists in the *monitoring and analysis*, *planning*, and *execution* activities. Each activity is carried out by one or more elements that, for their operation, use static and dynamic knowledge about the system and its state. Not all elements operate at the same abstraction level; some are tied to the managed system and execution environment, while others are built on top of the latter, and address the requirements specified by human operators. To provide a better understanding of the self-management support, the elements are organized in different layers according to their level of abstraction and the information they require from other elements. These layers are addressed next.

3.6 Abstraction Layers

The self-management elements are organized in four distinct layers, as depicted in Figure 3.4. The proposed layering draws a clear separation between the managed system and the self-management support, creating two abstraction levels: the managed system level and the self-management level. The self-management level, in turn, is internally structured into multiple layers: the change management layer and two planning layers, namely the action policy and the goal policy layer. In summary, the four abstraction layers are the: *managed system*, *change management*, *action policy*, and *goal policy*. Figure 3.4 also depicts the static and dynamic knowledge necessary for the self-management. The following notation is used in the diagram:

- Each layer is depicted by a rectangular frame.
- Boxes inside the frame represent elements that execute at the abstraction level that corresponds to that layer.
- An upwards arrow leaving the frame represents information that is provided in runtime by the elements of the layer to the layer above.
- A downwards arrow entering the frame represents directives received from the layer above.
- In the managed system, the S stands for sensors and E stands for effectors.

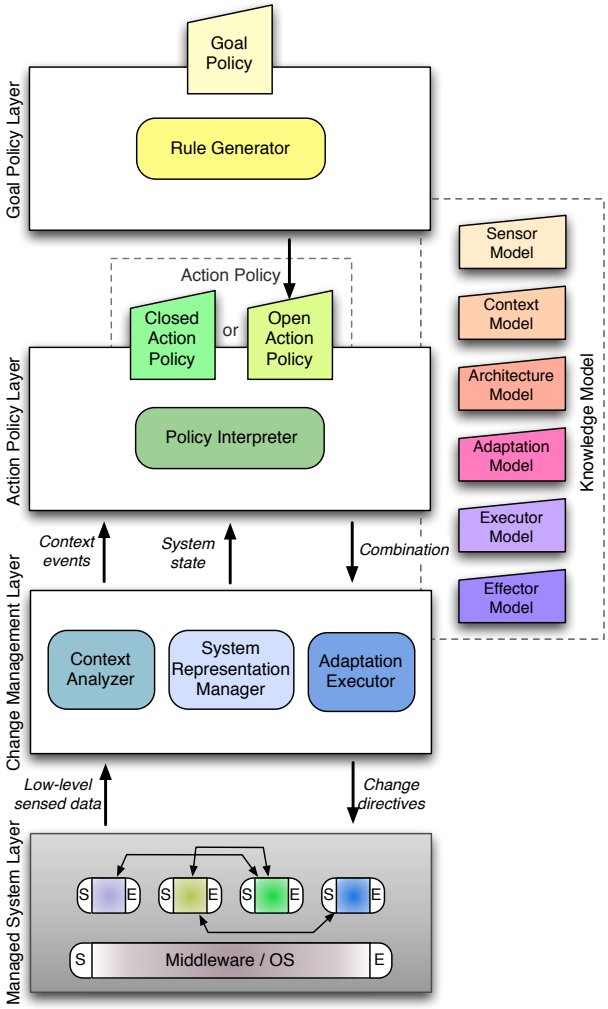


Figure 3.4: The approach's elements and levels of abstraction

Next, we provide a more detailed description of each layer.

3.6.1 Managed System Layer

The managed system layer corresponds to the lowest abstraction level. This layer includes the managed system and the executing environment, as well as their *sensors* and *effectors*.

Sensors are hooks introduced in the managed system for monitoring purposes, thus, capturing information regarding the system execution or the environment. Sensors can be associated with a component, a node, or a system property. They can capture different types of information, such as the system load, the CPU use, or the throughput. The raw information may

be captured periodically, at pre-determined time intervals, or, asynchronously through exceptions and traps. These asynchronous events and the raw information collected periodically are transmitted to the layer above. Depending on the component being monitored, the sensor may use standardized interfaces (for instance, SNMP [CFSD90]), be custom made for the specific interfaces of the component, or require the component to be extended in order to export the needed information. In some cases, components may register information in the form of logs, which can be monitored and parsed by the sensor to obtain the required data.

Effectors are also hooks in the managed system that allow to perform changes to the system. An effector can be associated to a component, a node, or to the system. If associated to a component, it is prepared to change the component configuration and control its operation; if it is associated to a node, it can change the node configuration and properties; if associated to the system, it can change the nodes that constitute the system, as well as any system parameters. Effectors may also perform several operations to prepare the element for adaptation, such as forcing the component to a quiescent state, or retrieving/setting the component state. Effectors may be in one of two operational states: *waiting* or *busy*. In the waiting state, an effector is waiting to receive control messages that trigger reconfigurations. The effector then switches to busy state and remains in that state until it has completed all the required reconfigurations.

3.6.2 Change Management Layer

The change management layer contains three elements: the *context analyzer*, the *adaptation executor*, and the *system representation manager*. These elements support the monitoring, analysis, and execution activities.

The context analyzer is responsible for the monitoring and analysis activity with the assistance of sensors. It receives all the raw information provided by the sensors and interprets this information to capture the system state. While the distribution of components is transparent to sensors, it is not to the analyzer. As a result, the context analyzer is aware of the system configuration and is prepared to interpret information for both non and distributed components, and to integrate system-wide information that comes from several nodes. The analyzer may also store a history of the information collected in the past. This is useful for smoothing algorithms

used in the monitoring, that clean momentary peaks. Finally, the context analyzer is responsible for detecting when the system state becomes undesirable and may require adaptation. When such an occurrence is identified, a context event is sent to the layer above.

The adaptation executor determines how an adaptation is achieved in the managed system. For this purpose, the adaptation executor takes into consideration the system state, specific requirements associated with the adaptation target (such as the need to put a component into a quiescent state), and the type of adaptation to be performed. The distribution of components is a critical aspect to consider, as it may require coordination of the different instances to execute the adaptation. The executor operation starts when it receives a *combination* (a set of adaptations to be performed) from the layer above. Then, it selects the appropriated *reconfiguration strategy* to execute the adaptation. A strategy is a sequence of commands that are exchanged with effectors and that execute a particular adaptation.

The system representation manager has the sole responsibility of maintaing the part of the system representation that is dynamic. The system representation is bootstrapped with the initial system configuration. Changes to the system representation may have two different causes. They can be caused by an environmental phenomena, such as a fault, or may be triggered internally, as the result of self-adaptation. If the source of change is external, the context analyzer will signal the manager, who finds the new architecture for the new system state. If the source is internal, it is the adaptation executor that signals the manager. In summary, the representation manager depends on information provided by the context analyzer and the adaptation executor and provides the updated information to the layer above, so that it can be used in the planning activity.

3.6.3 Action Policy Layer

The action policy layer can be used to control directly the adaptation, or be used in combination with a goal policy layer (above). The layer includes only one element: the *Policy Interpreter* that decides which adaptation(s) should be performed. The decision is based on the evaluation of an action policy according to the current system state. Information regarding the current system state is provided by the system representation manager, or obtained by per-

forming queries to the context analyzer. The evaluation mechanism is prepared to handle two different action policies. A *closed action policy* is employed if the rule-oriented planning is used. This type of action policy consists in a set of rules and is provided by the system designer. Each rule states which adaptation or adaptations should be performed when a specific undesirable system state occurs. An *open action policy* is used in combination with goal-oriented planning. This type of action policy is generated automatically in the layer above (goal policy layer). The evaluation mechanism depends on the type of action policy that is used and requires information that is available at the knowledge model and in the system representation.

3.6.4 Goal Policy Layer

The goal policy layer corresponds to the highest abstraction level. There is a single element in this layer: the *rule generator* that is responsible for generating an open action policy from a *goal policy*; the policy consists of a collection of goals that the system must satisfy. The goals may refer to particular performance indicators, system properties, or other non-functional properties. Some examples are the throughput, the memory use, the redundancy, and the power consumption, among many others. Each goal may refer to one or more aspects of the system behavior, establishing specific thresholds or best-effort demands. The rule generator identifies the system states that are undesirable and, for each state, it filters the set of adaptations so that all the adaptations that are not helpful are eliminated (the entire set of possible adaptations are described in the knowledge model). Then, the set of viable combinations of adaptations is computed. The pairing of a given state with the set of combinations of adaptations that may be useful to meet the target goals given that state creates a rule; the set of all rules created this way forms the open action policy.

3.7 Self-management Knowledge and Activities

The previous section overviewed the approach's elements, according to the abstraction level, interaction, and organization in the proposed approach. In this section, we address the same elements and self-management knowledge from the perspective of the closed control loop activities. As described in Section 2.1.2.2, the loop consists in the four activities already mentioned

and based on some knowledge. In our approach, we opt to join the monitoring and analysis in a single activity, as they are related. Next, we address the knowledge and the three activities of our approach.

3.7.1 Knowledge

Knowledge refers to the information necessary to monitor and adapt the system. Knowledge can be static or dynamic. The *static* knowledge refers to all the information necessary for the self-management activities that do not evolve with the system execution, while the *dynamic* knowledge contains all the information regarding the evolving system execution. The static knowledge is captured in the *knowledge model*. The dynamic knowledge is captured in the *system representation* (see Sections 3.4 and 3.6.2).

As depicted in Figure 3.4, the static knowledge model captures the necessary information for the operation of the elements in the goal policy, action policy, and change management layers. This model includes the configurations and properties of the system elements, such as the sensors, and effectors, but also knowledge regarding how the system can be adapted. The latter includes information regarding the type and properties of each component, information regarding the available nodes, the available adaptations to change the system behavior and, finally, information regarding how adaptations can be executed. This knowledge is organized into several sub-models: sensor, context, architecture, adaptation, executor, and effector models.

The sensor and context models are tied with the monitoring and analysis activity. The *sensor model* describes the sensors available in the system, namely, the raw data that they capture and how often they capture that data. The *context model* describes composite sensors that specify how raw context information is interpreted to obtain a global view of the system. The *architecture model* describes the components, nodes and the system, their properties and initial configuration. The adaptation, executor, and effector models are tied with the planning and execution activities. The *adaptation model* describes the adaptations available to change the behavior of components, nodes, and the system. The description may also include some estimate of how they affect the system behavior. The *executor model* describes the reconfiguration strategies available to perform adaptations. Finally, the *effector model* describes the commands

that each effector is able to perform.

3.7.2 Monitoring and Analysis

This activity is performed by the sensors and the context analyzer. The context analyzer gathers all the raw information captured by the sensors and interprets this information to characterize the system behavior. The analyzer operates in the change management layer.

We assume that all sensors have configurable *time periods* that control the time interval at which the monitored data must be captured and sent to the context analyzer. The configuration of this parameter is made before runtime, by using information that is provided in the sensor model, as depicted in Figure 3.5. The choice of monitoring interval is a decision tied to the nature of the information and the resources required to monitor it. Another aspect is that sensors must be able to reply to a query sent by the context analyzer. Furthermore, a sensor is only aware of itself, thus, it has no knowledge of other sensors, nor if the component is distributed.

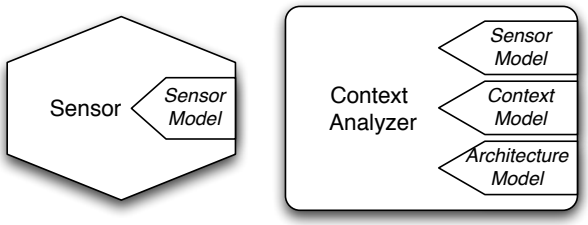


Figure 3.5: Models used by sensor and context analyzer elements

The context analyzer creates a representation of the global system state from the partial data provided by the individual sensors. The context analyzer uses information from the static knowledge model to operate, as depicted in Figure 3.5. The architecture model provides information about the nodes, components and the system being monitored. The sensor model describes the available context information, captured by sensors. The context model describes how the raw information collected by the sensors must be analyzed and combined to build all the information regarding the system state and the context. When any relevant change occurs, the context analyzer will signal the policy interpreter at the action policy layer, so that change can be addressed. The description of the relevant changes is also in the context model, made in the form of events. The operation of the policy interpreter may demand information from

the context analyzer. For this purpose, it is prepared to accept queries, that demand fresh or interpreted information regarding the system.

3.7.3 Planning

The planning activity determines the self-management of the system. As noted before, there are two reifications of the planning: rule and goal-oriented. The choice of reification has impact on the approach operation. If the rule-oriented planning is selected, the approach runs entirely online, as depicted in Figure 3.6. Instead, if the goal-oriented planning is selected, the approach also includes a preparation phase, consisting in an *offline support*. The choice of reification also affects the execution of the policy interpreter. Next, we address the difference between both reifications.

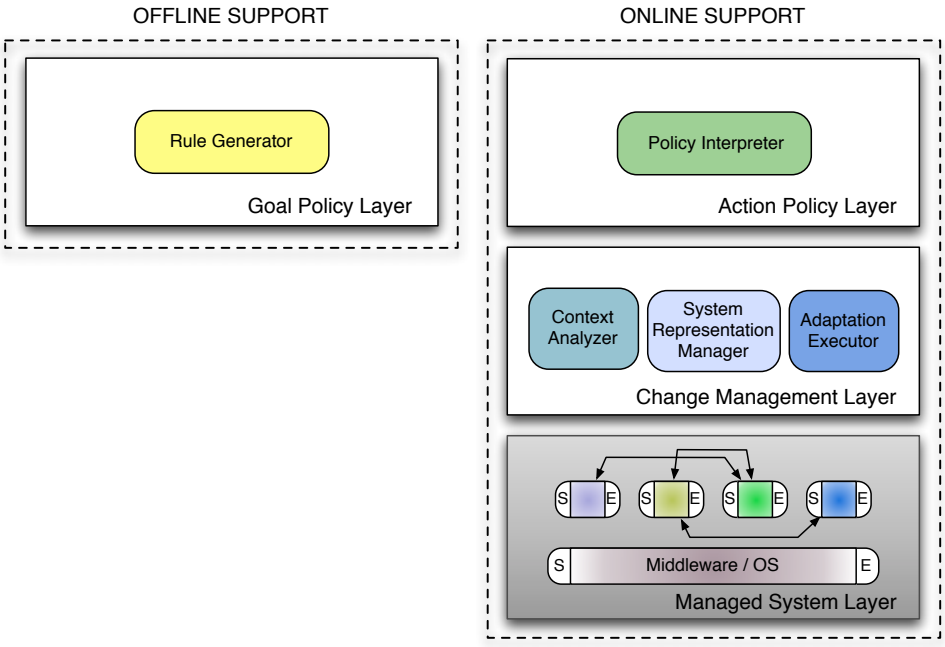


Figure 3.6: The offline and online support

3.7.3.1 Rule and Goal-oriented Planning

The *rule-oriented* planning is intended for systems where the adaptive behavior is simple, despite the amount of components or nodes involved in the system. An adaptive behavior is

simple if it considers a small amount of adaptations or if the adaptations address very specific scenarios, with little overlapping. This type of planning also allows to reuse an existing action policy. The self-management support for the rule-oriented planning is run exclusively *online* and relies only on the policy interpreter, at the action policy layer. The change management and managed system layers are also part of the online support. This planning relies on an action policy to describe the possible system adaptations. Figure 3.7 depicts the models necessary for the operation of the policy interpreter in the rule-oriented planning.

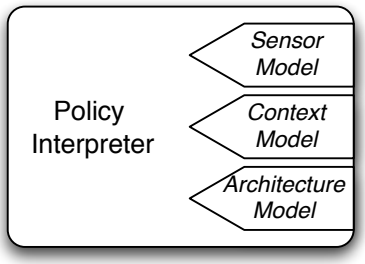


Figure 3.7: Models employed in the rule-oriented planning

The *goal-oriented* planning targets systems with complex adaptive behaviors. An adaptive behavior is considered complex if it includes a large number of components, adaptations, trade-offs, and conflicts. A complex adaptive behavior usually tries to achieve several different goals for the system; because it is concerned with more than one indicator of the system performance. This type of planning receives as input a high-level goal policy, that establishes the goals for specific system indicators, related to performance or non-functional properties. The self-management support for goal-oriented planning consists in *offline* and *online* phases. During the offline phase a closed action policy is generated from the goal policy. During the online phase an action policy (either generated or provided by the designer) is evaluated by the policy interpreter. The models employed are depicted in Figure 3.8. The two types of planning will be discussed in detail in Chapter 5.

3.7.4 Adaptation Execution

The execution of the adaptations selected by the planning activity is the final step of the control loop. The adaptation executor, in the change management layer, and the effectors,

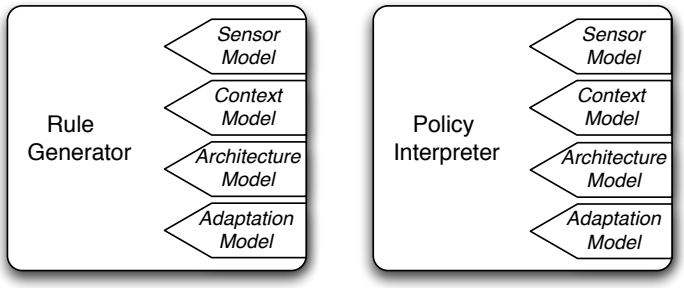


Figure 3.8: Models employed in the goal-oriented planning

in the managed system layer, are the elements involved in the execution of adaptations. The executor is responsible for determining how the selected adaptation(s) should be applied to the system. This depends on the selected adaptations and the current system state. As depicted in Figure 3.9, the executor, effector, and architecture models provide the necessary knowledge regarding the adaptations and their requirements, while the system representation provides the system state. This information allows the executor to pinpoint the exact adaptation targets. For each adaptation it is necessary to find a *reconfiguration strategy* to execute it. These strategies dictate the sequence of commands that the executor must exchange with the effectors to perform the adaptation (the commands accepted by the effectors are described in the effector model, see Figure 3.9).

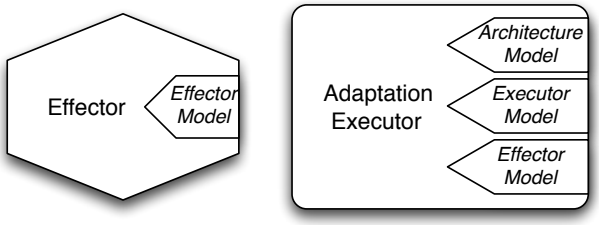


Figure 3.9: Models used by adaptation executor and effector elements

Summary

This chapter described the conceptual framework used in the thesis. It presented the elements according to its organization in different abstraction levels and in terms of self-

management activities. The remainder of this thesis will describe the contributions made at some of these activities. Chapter 5 address the planning activity, both rule-oriented and goal-oriented. Chapter 6 addresses the execution activity, namely the reconfiguration strategies and execution support.

A model is just an imitation of the real thing.

Mae West

Chapter 4

Knowledge Model

The knowledge model captures the information necessary for the operation of the elements in the different layers. We recall that, as introduced in Chapter 3, the knowledge is organized into several models, which are depicted in Figure 4.1. The figure also shows the dependencies between the different models, with the architecture model being central to other models. This chapter addresses in detail each of the models of the knowledge model.

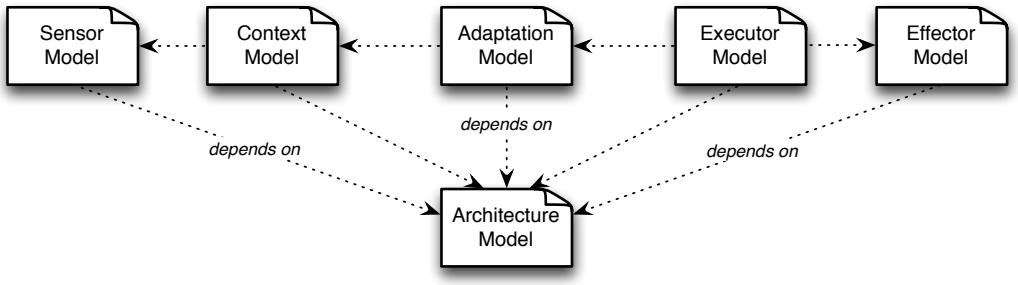


Figure 4.1: The knowledge model

4.1 Architecture Model

The architecture model describes the system elements and its architecture, as well as its initial configuration. The model consists in a global system description and initial configuration, and the definition of the components and nodes that compose the system. The system description covers the nodes that can be deployed in the system and any global parameters. The initial

configuration describes how the elements are deployed. The description of components relies on a component type hierarchy, where all types are sub-types of a root *ComponentT* type. The description of nodes relies on a node type hierarchy, where all types are sub-types of a *NodeT* type as root. The notion of sub-typing subsumes the *is a* relationship. Hence, all the super-type characteristics are inherited by the sub-type. In both hierarchies, types can be *abstract* or *concrete*. Abstract types allow to organize components or nodes according to the characteristics and features they provide. The concrete component and node types sub-type abstract types, which are those that can be used to create instances of components and nodes. The two type hierarchies are presented in Listing 4.1.

In the component type hierarchy, a component type can be marked as *abstract* and/or *distributed*. In the absence of an *abstract* marker, the component type is concrete, and in the absence of a *distributed* marker it is non-distributed. The description of a component type also includes the component types from which it is a sub-type and the adaptable parameters. The parameters of distributed components can be *local* (to each instance) or *global* (that apply to all instances of the same type). The later are identified with the marker *global*. Non-distributed components can only specify local parameters. A component type inherits the parameters from the component types that it sub-types. These parameters cannot be redefined nor eliminated.

In the node type hierarchy, a node type can be marked as *abstract*, otherwise being concrete. The description also includes the node types from which it sub-types, any adaptable parameters, and the component types that can be deployed in the node. All parameters are local to the node. In a similar manner to component types, the inherited parameters cannot be redefined or eliminated.

The definition of the system builds on the node type hierarchy, which is also presented in Listing 4.1. The system designer is required to define the maximum number of nodes that the system can have, together with any adaptable parameters the system may have. Also, she must provide the list of node types that can be deployed in the system.

Listing 4.2 illustrates the description of component and node types, and the system in the architectural model. They refer to the running example introduced in Section 3.1. The first example defines an abstract *Catalog* component type, which is non-distributed and can

Listing 4.1: Architecture model: components, nodes, and system

```
{Abstract} {Distributed} Component componentType
 subtype [componentType]*
  Parameters
   ([Global][parameter:type])*

{Abstract} Node nodeType
 subtype [nodeType]*
  Parameters
   [parameter:type]*
  Components
   [componentType]*

System
 Parameters
  max_nodes = number
  [parameter:type]*
 Nodes
  [nodeType]*
```

be configured to provide images with *regular* or *low* fidelity. This type is the super-type of *BusinessCatalog* and *HomeCatalog* component types, which reflect the type of customers that the component serves. The example that follows simply describes a *WebServer* component type. Next, the component type *Infinispan* is defined, which is distributed and has both local and global parameters. In this example, the multicast property of an *Infinispan* component is a global parameter, since all instances of *Infinispan* must use the same primitive to inter-operate. The number of threads of each instance is a local parameter, that can be configured independently for each replica. In terms of node of types, the *ServerNode* is a concrete node which has an *is_active* parameter and allows the component types *Infinispan*, *WebServer* and *Catalog* to be deployed. Finally, the system is described as having a maximum of 10 nodes, which are all of type *ServerNode*.

The architecture model also includes the initial configuration of the system and the default configuration for new instances. The initial configuration of the system, presented in Listing 4.3, describes how many nodes of a node type should be created, their configuration in terms of parameters and instances of concrete component types. The configuration of the component instances is also specified. The description of how component and nodes instances should be

Listing 4.2: Excerpt of an architecture model

```
Abstract Component Catalog
 Parameters
  fidelity:{regular,low}

Component BusinessCatalog
 subtype Catalog

Component HomeCatalog
 subtype Catalog

Component WebServer

Distributed Component Infinispan
 Parameters
  Global multicast_properties:{causal,total}
  nb_threads:int

Node ServerNode
 Parameters
  is_active: boolean
 Components
  WebServer
  Infinispan
  Catalog

System
 Parameters
  max_nodes = 10
 Nodes
  ServerNode
```

Listing 4.3: Architecture model: initial system configuration

```
Configuration
 System
  [nodeType = number]*
 Nodes
  [nodeType:{[parameter=value]*}:{[componentType]*}]*
 Components
  [componentType:{[parameter=value]*}]*

Configuration
 System
  ServerNode = 3
 Nodes
  ServerNode:{is_active=true}:{WebServer, Infinispan, HomeCatalog, BusinessCatalog}
 Components
  Infinispan:{multicast_properties=causal, nb_threads=3}
  Catalog:{fidelity=regular}
```

configured is used as the default configuration to deploy new instances. When the system starts, each instance is created from a concrete type and set up according to the information on the initial configuration. Listing 4.3 also presents an example of an initial configuration for the system.

The architecture model provides information that is fundamental for other elements of the knowledge model, but also for the system representation. The system representation describes the system current configuration, which consists on the following information:

- the values of system parameters;
- node instances and the values of node parameters;
- the component instances deployed in each node;
- the values of component parameters (if the parameter is global, then the same value is used for all instances).

The system representation evolves as a result of changes in the execution environment and adaptations of the system. Thus, it always provides information on the latest, current configuration of the system. When describing elements in the knowledge model, often the

system designers or component developers have to refer to a particular instance of the system or a parameter. However, since this information is dynamic, references or *architecture operators* are used. References are used to refer to a specific parameter, which is made in the form: *instance.parameter_name*, over a component or node instance. If the human operator needs to refer to the system parameters, then this is made in the form: *system.parameter_name*. The architecture operators reflect over the current system configuration, providing updated information on the component and nodes instances running. The use of these operators will become more clear in the upcoming description of the other models of the knowledge model. The architecture operators available are the following:

- *getComponentInstances(CT,NT)* returns all the instances of a component type CT that are deployed in nodes of type NT. If the node type is not important, then passing the $NodeT$ type will select all the component instances of type CT.
- *getComponentInstance(CT,NI)* returns the instance of component type CT that is running in the node instance NI.
- *getNodeInstances(NT,CT)* returns all the instances of a node type NT that have instances of component type CT running. If the component type is not important, then passing the $ComponentT$ type will select all the node instances of type NT.
- *getNodeInstance(CI)* returns the node instance where the component instance CI is running.

It is important to note that all operators that accept a component type or node type, can accept the root abstract types $ComponentT$ and $NodeT$, thus referring to all component types or all node types.

4.2 Sensor Model

In the proposed approach, we assume that each component or node instance has one sensor or none. The same applies to the system. The sensor model describes the information that these sensors provide. It defines for each sensor the type of element that is being sensed and

the *observables* and *events* that will be provided to the context analyzer. Observables model the context information that is periodically captured by sensors and monitored by the context analyzer, while events are indications of relevant asynchronous changes.

As presented in Listing 4.4, each sensor description has a *target* element which can be the system, a component type, or a node type. In both cases, each deployed instance will have a sensor instance that captures and provides the information in the sensor description. This also means that if a component type is distributed, then all instances of that component type will have its own sensor instance. The information that is captured and provided to the context analyzer is defined in the form of *observables* and *events*. Observables have a name, the accepted parameters, the return type and the relevance margin (RMargin). The relevance margin is used to determine if two readings are equivalent or not. If the readings' values are separated by less than the margin value, then they are considered equivalent. The description of an observable is concluded with the sampling interval between readings (in seconds). Events have a name and, depending on the type of event, they will have a triggering condition and/or a time interval. *Sporadic* events are triggered when a condition is verified, while *periodic* events are triggered when a certain amount of time has elapsed. Periodic events may also define a condition that controls the triggering.

If the target of a composite sensor is a distributed component type, observables and events can be marked as *global*. A global observable is an observable that is shared by all the instances of a distributed component type and whose value is the same for all instances. A global event is an event that is triggered only once, for the component, instead of being triggered on a per-instance basis.

Listing 4.5 provides some concrete examples of sensors for the running example. The first example describes a sensor for the *WebServer* component type and has two observables: *requests_per_second* and *cpu_use*, sampled every ten seconds. The second example describes a sensor for the *Infinispan* component type. The sensor has two observables: *cpu_use* and *total_cache_size*. The *cpu_use* is per instance, while the *total_cache_size* is component wide. The third example describes a sensor that any component of type *Catalog* will have. For instance, instances of component types *BusinessCatalog* and *HomeCatalog*. The fourth example describes

Listing 4.4: Sensor model

```
Sensor
 Target:
  componentType|nodeType|system
 [Observable
  [[Global] returnType obs_name([parameter]*): RMargin number
  periodically: number]]*
 [Event
  [Global] eventName
   [with [attribute_name: type]*]*
   [periodically: number]]*
```

a sensor for the node type *ServerNode*, which means that any node of this type will have its own sensor. Finally, the last example describes the sensor for the system, which provides an observable regarding the number of *active_nodes* and two events. The first event is a *sporadic* event that signals that there is a failed node, while the second event is a *periodic* event, triggered every sixty seconds.

As shown by the third example in Listing 4.5, one may define sensors whose targets are abstract component types. When there are more than one sensor description for a type, the instance of that type will have the observables and events that result from the union of all descriptions. This is the case of the sensor for a *HomeCatalog* instance, that will have two observables: *cpu_use* and *home_processing_time*.

The sensor instances deployed at a given time depend on the components and nodes that are running. Thus, similarly to architecture operators, we provide *sensor operators* that allow to reflect on the sensor instances that are running. The operators return a set with the existing sensor instances according to some characteristics, such as the type, observable or event name, among other options. If there are no sensor instances that fit the characteristics, an empty set is returned. Below, we present a (non-exhaustive) list of operators:

- sensorSystem() returns the sensor instance attached to the system.
- sensorsComponentType(CT) returns the sensor instances that are attached to components of type CT.
- sensorsNodeType(NT) returns the sensor instances attached to nodes of type NT.

Listing 4.5: Excerpt of a sensor model

```
Sensor
 Target: WebServer
 Observable int requests_per_second(): RMargin 10 periodically: 10
 Observable  double cpu_use(): RMargin 0.01 periodically: 10

Sensor
 Target: Infinispan
 Observable double cpu_use(): RMargin 0.01 periodically: 10
 Observable Global int total_cache_size(): RMargin 0 periodically: 60

Sensor
 Target: Catalog
 Observable  double cpu_use(): RMargin 0.01 periodically: 10

Sensor
 Target: HomeCatalog
 Observable int home_processing_time(): RMargin 0.2 periodically: 30

Sensor
 Target: ServerNode
 Observable int power_consumption(): RMargin 1 periodically: 60

Sensor
 Target: system
 Observable int active_nodes(): RMargin 0
 Event node_failed with failed_node: ServerNode
 Event beacon periodically: 60
```

- sensorsDeployedInNode(NI) returns the sensor instances that are deployed in a node instance NI.
- sensorsObservable(ON) returns the sensor instances that sense an observable named ON.
- sensorsEvent(EN) returns the sensor instances that sense an event named EN.
- sensorInstance(I) returns the sensor instance associated to the component or node instance I.
- sensorNode(CI) returns the sensor instance of the node where the component instance CI is deployed.
- sensorsSelectT(CT,NT,ON) returns all sensor instances attached to a component of type CT, as long as they are in node instances of type NT, and that sense observables named ON. This operator is a shorthand for the intersection of the operators sensorsComponentType(CT), sensorsNodeType(NT), and sensorsObservable(ON).
- sensorsSelectI(CT,NI,ON) returns the sensor instance attached to a component of type CT in the node instance NI, and that sense observables named ON. This operator is a shorthand for the intersection of the operators sensorsComponentType(CT), sensorsDeployedInNode(NI), and sensorsObservable(ON).

Similarly to operators introduced in the previous section, the operators accept the root abstract types *ComponentT* and *NodeT*, which allow to refer to all component types or all node types. In the same manner, the observable name can be *undefined*, so that the selection is made independently of the observables name.

There is one last sensor operator that instead of giving a set of sensor instances, provides the system or the component or node instance that is being sensed by a particular sensor instance. This operator does not seem important on its own, but it allows to refer to a system instance in conjunction with other operators and in other models. The operator is the following:

- getTarget(SI) returns the system, or component or node instance to which a given sensor instance SI is attached to.

4.3 Context Model

The context model describes the context information used to characterize the system behavior, or that is relevant for the system adaptation. Naturally, all the raw information that is produced by sensors is implicitly part of the context that is available to manage the system. However, in general, it may be easier to express policies by defining observables and events which are computed/triggered from information provided by multiple sources. Therefore, the context model not only includes the context information captured by sensors, but it augments it with the interpretation, filtering, and processing of the captured information through the definition of composite sensors. The sensor instances, corresponding to the sensors described in the context model, can be handled using the same sensor operators described in Section 4.2. Thus, a sensor operator can return a basic or a composite sensor, or both if it returns a set.

4.3.1 Composite Sensors

A composite sensor is an abstraction defined at the level of the context model, that provides new observables and events from the information provided by the basic sensors in the sensor model. Composite sensors can aggregate information from different sensors, or perform some operation on that information. The definition of a composite sensor is different from a basic sensor mainly in two aspects:

- The specification of a composite observable includes an *observableExpression*, that specifies how it is computed from other observables (which either are provided by basic sensors or by composite sensors).

- The specification of a composite event includes an event *condition*, that specifies when the event should be triggered, and the information that is carried by the event.

The differences above are reflected in the description of composite sensors, presented in Listing 4.6. Similarly to basic sensors, a composite sensor has a target, observables, and events. The target is a component type, a node type, or the system. At runtime, the target becomes the instance to which the sensor instance is associated to. Thus, *target* can be used in the description

Listing 4.6: Context model

```
Composite Sensor
 Target:
  componentType|nodeType|system
 [Observable
  [Global] returnType obs_name([parameter]*) [: RMargin number]
  [periodically:number]
  observableExpression]*
   [Event
  [Global] eventName
  [when condition]
  [with [attributeName=expressionOfType]*]
  [periodically: number]]*
```

of observables and events to refer to the target instance. The description of observables has some aspects in common with basic sensors. Observables can be marked as global, being available for the component type, and not on a instance basis. Observables in composite sensors that employ global observables may be global or not. The system designer is responsible for marking them as global when it is the case. The most significant difference from basic sensors, is that the system designer can describe an expression to calculate the observable. This expression may employ other observables or events, the target, architecture and sensor operators, component parameters, among others.

The description of events in composite sensors is similar to basic events, with two exceptions. One is the when statement that describes a triggering condition for the event. This condition may refer to other events or to the state of an observable. If the when and periodically statements are used together in the description of an event, that means that every *number* of seconds the condition is evaluated and, if satisfied, the event is triggered. The other exception is the with statement that describes the information that will be carried by the event. The description of this statement includes the name of the attribute and how its value is obtained. Similarly to basic sensors, events can also be global, thus a single event is triggered by distributed component.

Listing 4.7 provides several examples of observables and events of a composite sensor attached to the system. The first observable, average_workload, calculates the average of the number of requests per second at the web servers, using the values sensed at each different server. In this case, the *sensorsSelectT(...)* returns the sensors attached to all components of

Listing 4.7: Excerpt of a context model

```
Composite Sensor
 Target: system
 Observable
  double average_workload(): RMargin 20
  periodically:30
  Avg s.requests_per_second() | s: sensorsSelectT(WebServer,NodeT,
      requests_per_second)

 Observable
  int servers_above_avg(): RMargin 3
  periodically:30
  Count (s.requests_per_second() > (target.average_workload()+LIMIT)) | s:
      sensorsSelectT(WebServer,NodeT,requests_per_second)

 Event
  WebclusterOverload
  when target.average_workad() > THRESHOLD
  with current_load = target.average_workload()
```

type *WebServer*. The second observable, servers_above_avg, describes how many web servers are subject to a higher workload than the average. The expression relies on the *Count* function and in the average_workload observable. The listing also shows the description of an event that is triggered when the average_workload exceeds a predefined threshold.

4.3.2 Combination and Aggregation Functions

The description of composite sensors provided in the previous section allows the designer to cover the monitoring needs of systems in general. However, we have identified some recurring patterns during the specification of the context model. Thus, we have extended the specification of composite sensors so that these patterns can be expressed in a simple manner. Next, we address in detail the patterns and how the composite sensor specification was extended to support them.

One pattern is to have information regarding a node as a whole, that is obtained from all the instances running on that node. For instance, consider the observable cpu_use, captured for all the instances of different component types that run in a given node. If the designer wants to monitor the total CPU consumed by the node, these individual measures will have

Listing 4.8: Node-aggregated observables

```
Composite Sensor
 Target: ServerNode
 Observable
  double node_cpu_use(): RMargin 0.01
  periodically:10
  Sum s.cpu_use() | s: sensorsSelectI(ComponentT,target,cpu_use)

Composite Sensor
 Target: ServerNode
 Node-aggregated Observable
  double node_cpu_use(): RMargin 0.01
  periodically:10
  from cpu_use with AF: Sum
```

to be combined. As presented in Listing 4.8, the observable *node_cpu_use* of the first sensor resorts to the use of several operators to calculate this information. The computation of the observable is the sum of the *cpu_use* of all component instances deployed in the node to which the sensor is attached. To simplify the description of the pattern, *node-aggregated* observables were introduced in the specification. These observables rely on an *aggregation function*(AF) supplied by the designer to calculate the aggregated value. The description of the observable *node_cpu_use* using this pattern is also presented in Listing 4.8, using the *sum* as the aggregation function to join the CPU use of all instances running in the node.

Another pattern is to have information regarding a component type, that is obtained from all the instances of that component type — both distributed and non-distributed. For instance, the system designer may need to monitor the average CPU consumption of a distributed component, as shown by the first *avg_infinispan_cpu_use* observable in Listing 4.9. To simplify the description of this pattern, we augmented the description of composite sensors with *component-combined* observables, which rely on a *combination function*(CF) to calculate the final value. The description using this pattern is shown by the second observable in Listing 4.9, using the *average* as a combination function.

The last pattern is a combination of both previous patterns. This pattern is to have the information system wide. For instance, continuing with the CPU consumption example, the system designer would like to monitor the average CPU use of the entire system. We augmented

Listing 4.9: Component-combined observables

```
Composite Sensor
 Target: Infinispan
 Observable
  double avg_infinispan_cpu_use(): RMargin 0.01
  periodically:10
  Avg s.cpu_use() | s: sensorSelectT(Infinispan,NodeT,cpu_use)

Composite Sensor
 Target: Infinispan
 Component-combined Observable
  double avg_infinispan_cpu_use(): RMargin 0.01
  periodically:10
  from cpu_use with CF: Avg
```

Listing 4.10: System-wide observables

```
Composite Sensor
 Target: system
 Observable
  double avg_global_cpu_use(): RMargin 0.01
  periodically:10
  Avg (Sum s.cpu_use | s: sensorsSelectI(ComponentT,x,cpu_use)) | x:
      getNodeInstances(ServerNode,ComponentT,cpu_use)

Composite Sensor
 Target: system
 System-wide Observable
  double avg_global_cpu_use(): RMargin 0.01
  periodically:10
  for ServerNode from cpu_use with AF: Sum CF: Avg
```

the description of composite sensors with *system-wide* observables that employ both combination and aggregation functions to calculate the final value. Thus, instead of describing a *node_cpu_use* observable and an *avg_global_cpu_use*, as presented in Listing 4.10 (first observable), the system designer may describe a single system-wide observable as illustrated by the second observable in the same listing.

A significant advantage of considering node-aggregated, component-combined, and system-wide observables in the context model is that the context specification becomes much more synthetic and simple. Listing 4.11 summarizes how these observables can be specified. System-wide observables play an important role in the goal-oriented planning. The ones that are relevant

Listing 4.11: Specification of combined and/or aggregated observables

```
Composite Sensor
 Target: nodeType
 Node-aggregated Observable
  returnType obs_name([parameter]*) [: RMargin number]
  [periodically:number]
  from from_obs_name with AF: function

Composite Sensor
 Target: componentType
 Component-combined Observable
  returnType obs_name([parameter]*) [: RMargin number]
  [periodically:number]
  from from_obs_name with CF: function

Composite Sensor
 Target: system
 [KPI] System-wide Observable
  returnType obs_name([parameter]*) [: RMargin number]
  [periodically:number]
  [for:[componentType|nodeType]*] from from_obs_name with AF: function CF:
      function
```

for that type of planning must be marked as KPI. Also, in the context of goal-oriented planning, the system-wide observables can be composed to form Composed-System-wide observables. These observables are calculated from other system-wide observables, according to a join function (JF), as presented in Listing 4.12. The join function can employ other system-wide observables. The composed-system-wide observables relevant for the goal-oriented planning must also be marked as KPI. The use of these observables in the goal-oriented planning is discussed in detail in Section 5.2.1.

4.4 Adaptation Model

The adaptation model describes the adaptations that are available to adapt the system. The description of an adaptation starts with a name and has a target, a list of requirements, a list of impacts, and a stabilization period, as presented in Listing 4.13. The name identifies the adaptations and thus is unique. The target of an adaptation can be a component type, a node type, or the system. At runtime, the target is an instance of that type or the system. The list

Listing 4.12: Specification of combined and/or aggregated observables

```
Composite Sensor
 Target: system
 [KPI] Composed-System-wide Observable
  returnType obs_name([parameter]*) [: RMargin number]
  [periodically:number]
  with JF: function

Composite Sensor
 Target: system
 KPI Composed-System-wide Observable
  double service_ratio(): RMargin 0.01
  with JF: throughput ÷ load
```

of requirements narrows the application of the adaptation to a set of system states. The list of impacts captures the known impacts of the adaptation in the system configuration and behavior. The impacts on the configuration of the system, nodes and components are expressed through a number of actions, while the impact on the behavior is expressed using functions that estimate how the affected observables will change. The stabilization period describes the time needed for the system to stabilize after the adaptation, so that it can eligible again for adaptation.

Listing 4.13: Adaptation model

```
Adaptation adaptationName
 Target:
  componentType|nodeType|system
 [Requires:
  [condition]*]
 Impacts:
  [impact]*
 Stabilization:
  integer
```

As mentioned above, two different types of impacts can be described. The first type of impacts is used to describe how the system configuration changes and is required in any description of an adaptation. Since the impacts refer to the adaptation target, different impacts will be available depending on the adaptation target:

- system.nodeAdded(NT) this impact can only be used in adaptations whose target is the system. It indicates that an empty node of type NT has been added to the system when

the adaptation concludes. It returns the node instance that was added.

- system.nodeRemoved(NI) this impact can only be used in adaptations whose target is the system. It indicates that a node instance NI has been removed from the system when the adaptation finishes.

- nodeInstance.componentAdded(CT) this impact can only be used in adaptations whose target is a node instance. It indicates that a new instance of a component type CT has been deployed in the node instance and returns the added component instance.

- nodeInstance.componentRemoved(CI) this impact can only be used in adaptations whose target is a node instance. It indicates that the component instance CI has been removed from the node instance. If the component is distributed, this impact only describes the removal of that instance.

- nodeInstance.componentReplaced(CT) this impact can only be used in adaptations whose target is a node instance. It indicates that the component instance *target* will be replaced by a new instance of the component type CT. If the instance is part of a distributed component, this impact affects only that instance.

- componentInstance.parameterChanged(P,NV) this impact can only be used in adaptations whose target is a component instance. It indicates that the parameter P in the component instance will be changed to a new value NV. If the component instance is part of a distributed component and P is global, all the instances of the component are affected. If P is local, then only this instance is affected.

The second type of impacts describes how the system behavior and performance are expected to change. This estimation is made for an affected observable and is provided as a function, which may depend on any observable in the context model or component/node parameters. The description may also employ architecture and sensor operators, as well as the *forall* keyword to iterate the set of sensors or elements returned by the operators. The description of these impacts assumes that all *known impacts* will be declared. This type of impacts is not required if the system designer decides by a rule-oriented planning.

- observable := F(observable1,...) describes the estimated value of an observable after the adaptation. This value is obtained using function F that may employ the current values of other observables and/or component or node parameters.

Listing 4.14 describes a number of adaptations. The first adaptation changes the *Infinispan*'s parameter that controls the multicast_properties from causal to total. Since the parameter is global, it needs to be invoked only in one instance. However, the impact on the *cpu_use* is for all instances, thus, the impact description needs to iterate over all instances of *Infinispan*. The CPU use of all instances is expected to increase by 10%. The second adaptation adds a new node empty to the system, as long as the maximum number of nodes has not been reached yet. The third adaptation deploys all the component instances necessary in the new empty node. The last adaptation activates an inactive node that is deployed in the system and ready to operate. As a result the number of active nodes of the system increases by one.

The adaptation model also includes a list of dependencies and conflicts between different adaptations. While some adaptations are considered to be conflicting by the system, such as having the same target or conflicting impacts (add a component and then remove it), the system designer can describe other conflicts. The description of explicit conflicts and dependencies is as presented in Listing 4.15. Conflicts are declared as pairs of adaptations. This assumes that despite the target instance, the adaptations are always conflicting. Dependencies have a more complex description. A dependency consists of a main adaptation and a collection of other, dependent, adaptations. The dependent adaptations can have the same target of the main adaptation or one that is given by the impacts. If it is the same, then the adaptation target is passed to the dependent adaptation. If it is not the same, then it is necessary to specify which impact returns the target. This is exemplified in Listing 4.16.

4.5 Effector Model

The effector model describes the control interface of each effector in the system. It lists the commands that can be invoked in component, node, and system effectors to perform changes. These commands address pre and post adaptation concerns, or even concerns directly tied to the

Listing 4.14: Example adaptations

```
Adaptation activateTotalOrder:
 Target:
  Infinispan
 Requires:
  target.multicast_properties == causal
 Impacts:
  target.parameterChanged (multicast_properties,total)
  forall s: sensorsComponentType(Infinispan) s.cpu_use *= 1.1
 Stabilization:
  60

Adaptation addNode:
 Target:
  system
 Requires:
  sensorSystem().active_nodes() < target.max_nodes
 Impacts:
  target.nodeAdded(ServerNode)
 Stabilization:
  60

Adaptation deployComponents:
 Target:
  ServerNode
 Impacts:
  target.componentAdded(WebServer)
  target.componentAdded(Infinispan)
  target.componentAdded(HomeCatalog)
  target.componentAdded(BusinessCatalog)
 Stabilization:
  60

Adaptation activateNode:
 Target:
  ServerNode
 Requires:
  target.is_active == false
 Impacts:
  target.parameterChanged(is_active,true)
  sensorSystem().active_nodes() += 1
 Stabilization:
  60
```

Listing 4.15: Adaptation model: conflicts and dependencies

```
Conflicts
 Adaptations [(adaptation_name1, adaptation_name2)]*

Dependencies
 If adaptation_name1 Apply
  [adaptation_name2 with Target adaptation_name1.target|adaptation_name1.Impacts.
      impact]*
```

Listing 4.16: Example of conflicts and dependencies

```
Conflicts
 Adaptations (deactivateNode, addNode)
 Adaptations (activateNode, removeNode)

Dependencies
 If addNode Apply
  deployComponents with Target target.NodeAdded(ServerNode)
  activateNode with Target target.NodeAdded(ServerNode)
```

adaptation. They may refer to state concerns, such as quiescence or state transfer, or controlling the operation of the elements to which the effector is attached. These concerns will be discussed in more detail in Section 6.1; for now we present the list of commands that can be supported by effectors:

- makeQuiescent: this command places the component or node instance in a quiescent state, i.e. a stable state where execution is halted.
- getState: this command retrieves the state of a component instance.
- putState: it command sets the state of a component instance.
- setParameter: this command sets a parameter of a component instance.
- start: this command starts the component instance operation.
- stop: it stops the component instance operation.
- pause: it halts the execution of a component instance.
- resume: it resumes the execution of a component instance.

- addComponent: this command is strictly for node effectors and cause a new component of a specific type to be added.
- removeComponent: this command causes a node effector to remove a component.
- addNode: this command causes a system effector to add a new node to the system.
- removeNode: it causes the removal of a node from the system.

The description of an effector involves not only the commands above but also the definition of to which type of element the effector will be associated with. Thus, the description also includes the component type, node type, or the system as targets. The description of the different effectors is presented in Listing 4.17, where the differences in terms of accepted commands is according to the type of target.

Listing 4.18 gives an example of an effector for *Catalog* components. The effector accepts all commands applicable to component types, allowing to set the *fidelity* parameters also.

4.5.1 Effector Operators

We provide *effector operators* to obtain the effector instances attached to component or node instances or to the system. These operators are only employed in the executor model, to define strategies. The following list of effectors is not exhaustive but shows the the most common effector operators:

- effectorSystem() returns the effector instance attached to the system.
- effectorsComponentType(CT) returns all effector instances that are attached to components of type CT.
- effectorsNodeType(NT) returns all effector instances attached to nodes of type NT.
- effectorsDeployedInNode(NI) returns all effector instances that are deployed in a node instance NI.

Listing 4.17: Effector model

```
Effector
 Target:
   componentType
 Commands
  [boolean makeQuiescent()]
  [stateType getState ()]
  [boolean putState (state: StateType)]
  [boolean setParameter (parameter_name, value: parameterType)]
  [boolean start()]
  [boolean stop()]
  [boolean pause()]
  [boolean resume()]

Effector
 Target:
   nodeType
 Commands
  [boolean makeQuiescent()]
  [boolean addComponent (c: componentType)]
  [boolean removeComponent (c: componentType)]

Effector
 Target:
  System
 Commands
  [boolean setParameter (parameter_name, value: parameterType)]
  [boolean addNode (n: nodeType)]
  [boolean removeNode (n: NodeType)]
```

Listing 4.18: Example of an effector

```
Effector
 Target:
  Catalog
 Commands
  makeQuiescent()
  getState()
  putState(state)
  setParameter(fidelity,{low,regular})
  start()
  stop()
  pause()
  resume()
```

- effectorsSelect(CT,NT) returns all effector instances attached to a component of type CT, as long as they are in node instances of type NT. This operator combines the effectorsComponentType and effectorsNodeType operators.

- effectorSelect(CT,NI) returns the effector instance attached to a component of type CT in the node instance NI. This operator combines the effectorsComponentType and effectorsDeployedInNode operators.

- effectorNode(CI) returns the effector instance attached to the node instance where a component instance CI is deployed.

- effectorInstance(I) returns the effector instance associated to the a component or node instance I.

It is important to note that all operators that accept a component type or node type, can accept the root abstract types *ComponentT* and *NodeT*, thus referring to all component types or all node types.

As in the sensor operators, there is one effector operator that provides the system or the component or node instance that is being adapted by a particular effector instance. The operator is the following:

- getTarget(EI) returns the system, or component or node instance to which a given effector instance EI is attached to.

4.6 Executor Model

The executor model specifies how adaptations described in the adaptation model are executed, using the commands defined in the effector model. The sequence of these commands forms a Strategy, and each adaptation has at least one strategy associated. If more than one strategy is available, the designer may state in the which strategy is the default strategy. The description of strategies relies on several statements that can employ architecture and effector operators to obtain elements of the system and their effectors, *forall* keywords to iterate the sets returned by operators, and any effector commands allowed. These statements can be grouped in

Listing 4.19: Executor model

```
[default] adaptationName Strategy strategyName [Parallel]
 [Step:
  [statement]*]*
```

Listing 4.20: Executor model: a strategy

```
default activateTotalOrder Strategy stopAndGo
 Step:
  forall e: effectorsComponentType(Infinispan)
   e.pause()

 Step:
  forall e: effectorsComponentType(Infinispan)
   e.setParameter(multicast_properties,total)

 Step:
  forall e: effectorsComponentType(Infinispan)
   e.resume()
```

a sequence of Steps that are executed serially. As a result, no command of a given step may be performed before all the commands of the previous step have been concluded. However, commands executed in the context of a forall statement in a given step may be executed in parallel. Listing 4.19 shows how a strategy is specified in the executor model. A strategy consists of the name of the adaptation and the strategy name, wether or not it is the default strategy, and a number of steps, each consisting in a collection of statements.

Consider the strategy for activating the total order in the *Infinispan* component presented in Listing 4.20. The *stopAndGo* strategy consists in three steps. In the first step, all instances of the *Infinispan* component are paused in parallel. Once all instances have been paused, in the second step, the value of the parameter multicast_properties is changed to total. As soon as all replicas are configured, in the third step, their execution is resumed in parallel, concluding the execution. On all steps, the commands are invoked over effectors, obtained using the operators described in the beginning of this section.

The executor model includes also information necessary to parallelize the execution of strategies. A strategy can be marked as Parallel or different strategies can be described as parallel, as presented in Listing 4.21. In the former case, it means that if the adaptation is selected for

Listing 4.21: Executor model: parallel strategies

```
Parallel
 (adaptationName1:strategyName1) [AND (adaptationName2:strategyName2)]*
```

several targets, its execution in different targets can be performed in parallel. In the latter case, it means that the any adaptations and corresponding strategies in the set can be performed at the same time, in parallel. The parallelism of strategies will be addressed in detail in Section 6.3.

Summary

This chapter describes the knowledge model in which the proposed approach relies. It describes each of the sub-models in detail: architecture, sensor, context, adaptation, effector, and executor. It also describes the operators used in the models and gives extensive examples of how describe the elements of each sub-model.

People's behavior makes sense if you think about it in terms of their goals, needs, and motives.

Thomas Mann

Chapter 5

Planning

The proposed approach relies on a planning activity that can be guided either by rules or by goals, described in a policy. The use of one or the other has different benefits and disadvantages, as previously discussed in Chapter 3. The system designer must decide which type of planning is best suited for the system and provide the corresponding policy. In the *rule-oriented* planning the system designer needs to manually describe an action policy that lists all the states where the system must be adapted. In the *goal-oriented* planning the designer manually describes a goal policy that lists the behavior goals for the system that must be fulfilled during runtime. In this type of planning, the high-level goal policy is then automatically transformed in an action policy that can be used by the policy interpreter. In this chapter, we address both the *rule-oriented* and the *goal-oriented* planning, discussing the scenarios where they may yield the best results and minimize the designer's effort. For each type of planning, we address its support and elements involved, namely the offline and online support of the different policies.

5.1 Rule-oriented Planning

The rule-oriented planning relies on an action policy composed by rules. Each rule characterizes an unwanted system state and lists the adaptations needed to return the system to a desired state. As previously mentioned in Section 3.7.3.1, this type of planning requires an in-depth knowledge of the system elements and the understanding of how changes in their behavior

impacts the overall system. This may be a reasonable demand when the adaptive behavior is simple enough, the number of system components is not very large, or the adaptations available are sparse. This type of planning may also be effective in cases where the adaptive behavior is straightforward, without complex trade-offs. The framing of the rule-oriented planning in the context of the approach is depicted in Figure 5.1.

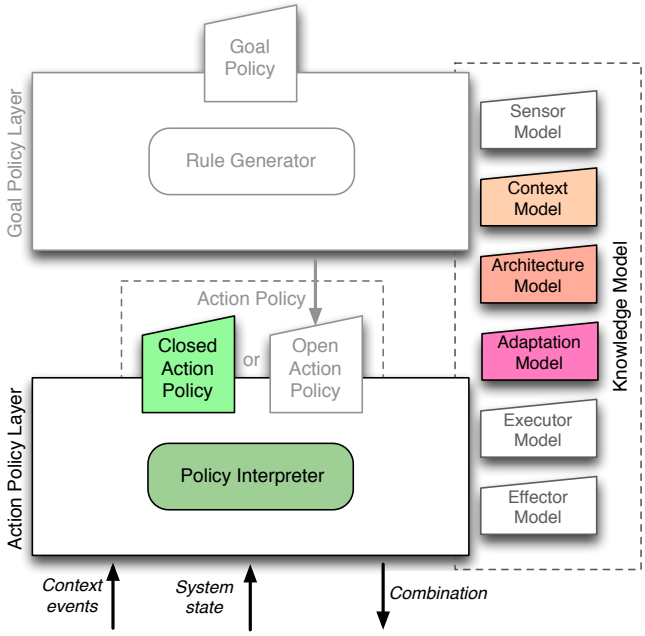


Figure 5.1: Rule-oriented planning in context

Rule-oriented policies are employed in this type of planning to describe the adaptation logic that determines how the system changes. They are called *closed action policies* in the context of the approach. The approach supports a single policy at a time, that is input to the policy interpreter in the action policy layer. The description of the policy used in this type of planning relies on information available in the knowledge model. Next, we address *closed action policies* and the operation of the *policy interpreter* in the context of rule-oriented planning. We conclude this subsection with a discussion of the tradeoffs involved with the use of this type of planning.

5.1.1 Closed Action Policies

An action policy is a collection of event-condition-action (ECA) rules [MD89] describing changes in the system configuration. Each rule features an *event* that triggers the evaluation of

the rule, a *condition* that describes the system state where the rule can be fired, and an *action*, one or multiple adaptations that should be applied if the condition is satisfied. In our approach, *closed* action policies are action policies that describe exactly how the system should be adapted and the strategy used to execute the adaptation.

The description of rules in closed action policies is as presented in Listing 5.1. The description relies on five elements: when, with, do, where and for. The When describes the triggering *event*, which can be a raw event provided directly by a sensor defined in the sensor model or a composite event defined in the context model. The With is used to state the *condition*: a boolean expression that may refer to parameters of the event (if any), the system configuration, or any observables from the knowledge model. The description of conditions may rely in any of the operators used in the definition of the composite sensors. The Do statement is used to describe the adaptations that must be applied to the system, by defining their names previously described in the adaptation model. The Strategy is the name of an adaptation strategy defined in the executor model. Recall that strategies are explicitly associated with adaptations, therefore the Strategy statement describes the strategy that will be used to execute the adaptation. If a strategy is not explicitly specified, the default strategy is selected.

The adaptations declared in a rule have a target. When the target of the adaptation is a component or node type, all existing instances of the defined type will be targets. This is called the *scope* of the adaptation. The scope of an adaptation can be specified using the Where and For statements. The Where allow to narrow down the target set of nodes from *all* (the default) to a smaller set, such as a specific type of node or to a node that is carried by the event. Similarly, the For narrows the target components from *all* component instances to only those of a specific type (as long as it is a sub-type of the component type specified in the adaptation). Depending on the adaptation and scenario, the target defined in the adaptation may be specific enough. However, adaptations whose target is less specific, such as an abstract type, may declare a scope to narrow down the target instances. The scope is also useful when the target is a specific node or component instance carried by the event.

There is an important aspect that should be considered when describing the policy rules. The order by which the rules are listed in the policy is not arbitrary, as it will be followed at

Listing 5.1: Closed action policy: the rules

```
When event
[With stateCondition]
 Do { adaptation [Strategy strategy]
   [Where nodeType]
   [For componentType]}+
```

execution time. This is true for both several rules triggered by the same event, as well as for several adaptations described in the same rule. In the first case, if several rules are triggered by the same event and their conditions are satisfied, the selected adaptations of the first rule will be executed before the adaptations in the second rule. In the second case, if a rule declares several adaptations, they will be executed by their order.

Listing 5.2 provides several examples of closed action rules, in the context of the running example. All examples target scenarios where the web servers are over their capacity, and are triggered by the WebclusterOverload event described in Listing 4.7. In the first rule, the adaptation adds a new node to the system, if the system has not reach the maximum number of servers. In this case, there is no need to describe a scope, because it is unambiguously the system. The second adaptation is to change the *fidelity* to low of all the *HomeCatalog* components in the system. This happens when the maximum number of servers has already been reached and the *BusinessCatalog* fidelity is already *low* in all instances. The last rule changes the multicast properties of the *Infinispan* components, because none of the previously described adaptations is possible. In this case, the scope is not necessary because the adaptation already targets all the instances of a concrete component type.

Listing 5.2: Rule examples

```
When WebclusterOverload
With sensorSystem().active_nodes() < system.max_nodes
 Do addNode

When WebclusterOverload
With sensorSystem().active_nodes() == system.max_nodes
   AND forall bc.fidelity == low | bc: getComponentInstances(BusinessCatalog,
       NodeT)
   AND forall hc.fidelity != low | hc: getComponentInstances(HomeCatalog,NodeT)
 Do toLowFidelity Strategy flash
  Where all
```

```
   For HomeCatalog

When WebclusterOverload
With sensorSystem().active_nodes() == system.max_nodes
   AND forall bc.fidelity == low | bc: getComponentInstances(BusinessCatalog,
       NodeT)
   AND forall hc.fidelity == low | hc: getComponentInstances(HomeCatalog,NodeT)
 Do activateTotalOrder Strategy stop-n-go
```

5.1.2 Policy Interpreter

The policy interpreter determines how the system will be adapted. For that purpose, it evaluates an action policy and has two operation modes, one for each type of planning. In this section, we cover the operation of the interpreter under the rule-oriented planning, where closed action policies are evaluated. The other operation mode is addressed in Section 5.2.5. The interpreter executes exclusively at runtime, as part of the online support.

As previously depicted in Figure 3.6, the general operation of the interpreter relies on two tasks: *preparation* and *selection*. However, in the rule-oriented planning, the preparation task is not necessary, thus, bypassed. The selection task takes place every time an event from the context analyzer is received. The interpreter goes through the entire policy, rule by rule, and evaluates the rules that are triggered by the event. When a rule is triggered by the received event, any conditions present are evaluated. The conditions may refer to the system state, such as components configurations, context information, among others. Thus, the policy interpreter relies on the system representation and context analyzer to obtain the required information and determine whether the conditions are met or not. When the conditions are satisfied, the adaptations are selected ant it is necessary to check the dependencies in the adaptation model. If the selected adaptation has any dependencies, then those adaptations are also selected. After evaluating the policy rules and checking the dependencies, it is necessary to check if all the adaptations are compatible. The conflicts description in the adaptation model is used for this purpose. When two adaptations are conflicting, we remove the one that comes second according to the rule order. Any adaptations that depend on the removed adaptation are also removed. The list of selected adaptations, as well as their scope and strategy, is sent to the adaptation executor for the next activity.

5.1.3 Tradeoffs

A significant advantage of the rule-oriented planning is tied to the expressiveness of the closed action policies. In conjunction with the knowledge model, the closed action policies allow to characterize virtually any system state and its adaptation. This fine-grain control of the adaptations that are applied to the system is possible because the knowledge model allows to tailor composite events that are triggered in specific situations. While policies cannot refer to instances because they are not know at design time, tailored events allows to overcome this issue, allowing pinpoint instances to adapt, that are carried by the event. These situations can be further filtered by defining conditions that allow the designer to define different responses to the same event, based on the system state. Furthermore, based on her know-how about the system operation, the designer may select the adaptations and strategies that should be used in each case.

Unfortunately, rule-oriented planning demands that the system designer gathers profound knowledge on the impact of each adaptation. This can be achieved either through past experience or by extensive experimentation. Only with this knowledge, the designer is able to select the most appropriate adaptation for each case and specify its target. This can be a complex task in systems where the number of adaptations is very large. For instance, even in the simple example above, it is possible to react to an overload in the cluster by adding new nodes or by changing the fidelity of components. The best adaptation depends on a number of conditions that characterize the system state.

Furthermore, in a policy composed by several rules, the goals that the designer is trying to enforce to the system are captured implicitly and not explicitly. This makes the understanding of the policy difficult and can make it extremely hard to maintain, change, or extend. Consider again the example above, if one opts to react to the system overload by increasing a server the designer implicitly gives preference to customer satisfaction detriment of resource consumption; on the other hand, if the policy opts to degrade the fidelity even in a scenario where it would be possible to increase the number of servers, one is implicitly preferring to save resources in detriment to customer satisfaction. Both policies may be valid in different business models, but such choices need to be inferred from the actions that have been selected and are not explicitly

reflected in the policy.

5.2 Goal-oriented Planning

The goal-oriented planning relies on policies composed by goals, specified in order of decreasing importance. Each goal describes an objective for the system behavior, which may refer to performance, service quality, resource provisioning, among other properties. While the rule-oriented planning offered great expressiveness at the expense of a profound understanding of the system, the goal-oriented planning aims at simplifying the task of managing the system. The system designer is expected to provide the goal policy, but its description does not require that the designer is knowledgeable of the available adaptations and how they are executed. Instead of explicitly specifying the adaptations that should take place, the system designer expresses goals that must be met by the system. The adaptations are selected automatically, using the information provided by the component developers and captured in the knowledge model. While goal-oriented policies allow for a shorter and more intuitive specification of the adaptive behavior, they require that component developers make available information about the impacts of each possible adaptation, such that adaptations may be selected in an automated manner. This assumes that we are able to assess the impact on the system from local impacts on instances.

The framing of the goal-oriented planning in the approach context is depicted in Figure 5.2. This type of planning encompasses both *goal policy* and *action policy* layers, with two main elements: the rule generator and the policy interpreter. The rule generator operates before runtime, and constitutes the offline support because its operation does not require information from the runtime. The policy interpreter operates at runtime, depending on information regarding the system state. The separation of the goal-oriented planning in online and offline support is depicted in Figure 5.3.

In this section, we address several aspects of the goal-oriented planning. First, we address the goal policy specification, which includes the key performance indicators used to describe goals and the actual goal policy. After, we address the elements that support the goal-oriented planning, namely the policy generator, the *open* action policies, and we revisit the policy interpreter.

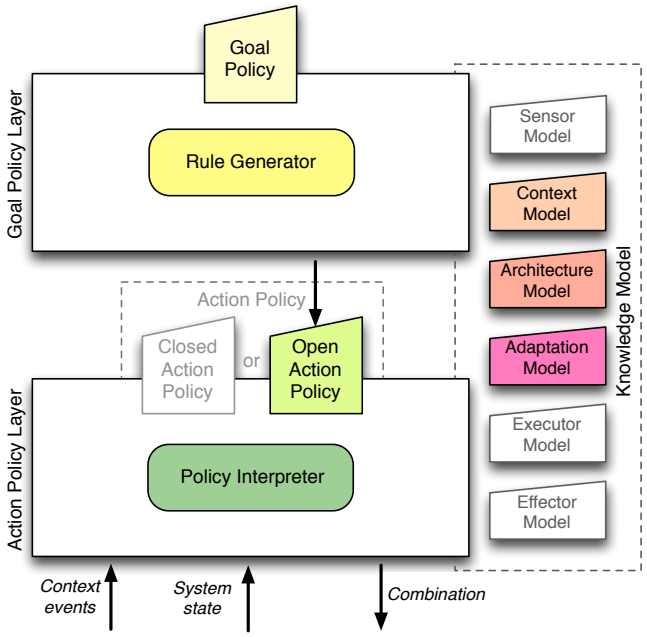


Figure 5.2: Goal-oriented planning in context

5.2.1 Key Performance Indicators

The *Key Performance Indicators*, KPIs for short, are observables from the context model, relevant to describe the goals for the system behavior. The system designer can mark an observable from the context model as a KPI only if that observable is *system-wide*. This is done using the marker with the same name. A KPI also demands that the aggregation and combination functions are limited to *monotonically non-decreasing* functions. A function is monotonically non-decreasing if *for all* $x < y, f(x) \leq f(y)$. In other words, by increasing x, $f(x)$ also increases or remains the same; conversely, by decreasing x, $f(x)$ also decreases or remains the same. This restriction is tied to the policy interpreter and will be discussed in Section 5.2.3.2. The sum, average, and maximum are three common monotonic non-decreasing functions used in the definition of system-wide observables.

The system designer may also describe *composite KPIs* (CKPIs for short) computed from previously declared KPIs. CKPIs are particularly useful to interpret new performance indicators from the basic KPIs or relate different KPIs using utility functions. The latter allows to attribute a specific weight to each KPI, thus, relating the concerned KPIs among themselves. This is

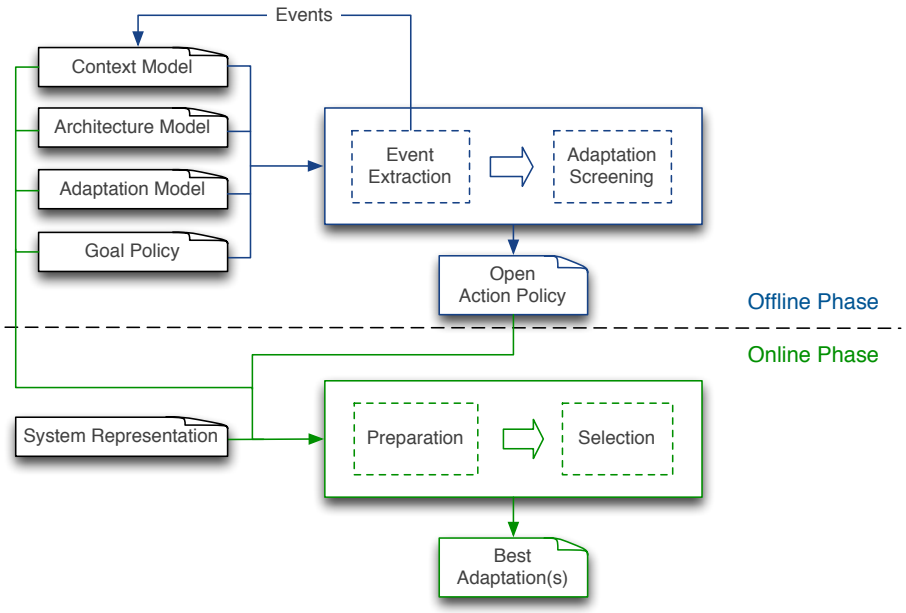


Figure 5.3: The offline and online support

useful to express trade-offs more complex than a simple rank order allows, and is discussed in more detail in the next section. CKPIs are *composed-system-wide* observables in the context model marked as KPI. A composed-system-wide observable relies in a join function that relates different system-wide observables (already marked as KPIs). Contrarily to the aggregation and combination functions, the join function does not face any restrictions, so that it can express utility functions. From this point onward, we will use KPI to refer to both KPIs and CKPIs.

All the observables that are of interest to describe the goal policy need to described as system-wide observables and marked as KPIs. This applies not only to properties of the system behavior and performance, but also to the system configuration. For instance, if the system designer needs to describe a goal that refers to the configuration of a component, such as the fidelity of *Catalog* components, she must describe a system-wide observable first, so that it can be marked as a KPI.

5.2.2 Goal Policy

The goal policy guides the planning, by describing the desired behavior for the system. As mentioned previously, the use of a goal policy may be an adequate solution in several scenarios.

One scenario is when the system designer does not have detailed knowledge of the system components and their adaptations, but has a global view of the system and its behavior. This makes it possible for the system designer to express what she considers the desired behavior and performance for the system, just by describing and marking the KPIs and specifying a goal policy. Another scenario is when there is a large number of components and it becomes very hard to manage the adaptive behavior. The larger the number of components, the more likely is the number of adaptations to grow. In cases where the number of adaptations is very large, describing the rules for a closed action policy becomes an error-prone task. Describing a much smaller and contained goal policy demands less effort and concentration from the system designer. The other scenario where a goal-oriented planning may be an adequate solution is when it is not possible to predict all scenarios where adaptation may be necessary, due to frequent changes during runtime. In this case, establishing more general guidelines for the system behavior in a goal policy allows the planning activity to be more equipped to deal and react in unexpected scenarios.

The goals in the policy establish the acceptable or desired state for a KPI. There are two groups of goals: *exact* and *optimization* goals.

Listing 5.3: Exact goals

```
Goal goal_name: kpi_name Above threshold_lower
Goal goal_name: kpi_name Below threshold_upper
Goal goal_name: kpi_name Between threshold_lower threshold_upper
```

Exact goals separate the values of a KPI in two disjoint sets: *acceptable* and *not acceptable*. There are three different types of exact goals and their description is presented in Listing 5.3. An Above goal states that the value of the KPI should be kept above the stated threshold, a Below goal that the value should be kept below the threshold, and a Between goal that the value should be kept within the stated lower and upper thresholds.

The optimization goals, instead of simply classifying the values of a KPI as good or bad, specify a total order between these values. That is, for any two values, which one is better, as long as their distance is larger than the relevance margin. We consider three different types of optimization goals and their description is presented in Listing 5.4.

Listing 5.4: Optimization goals

```
Goal goal_name: kpi_name Close target MinGain mgvalue Every period
Goal goal_name: Minimize kpi_name  MinGain mgvalue Every period
Goal goal_name: Maximize kpi_name MinGain mgvalue Every period
```

A Close goal states that the KPI value should be kept as close as possible to the *target* value, a Minimize goal states that the KPI value should be as small as possible, and a Maximize goal states that it should be as large as possible. The description of optimization goals includes other statements: the minimum gain and the periodicity. The minimum gain (MinGain) allows the system designer to describe when the adaptation is worth it. In other words, the designer demands that only when the gain of the applicable adaptations is above *mgvalue* they are selected. As any two values within the relevance margin specified for the target KPI are considered indistinguishable, the *mgvalue* should always be larger than the relevance margin. In the end, this statement allows the designer to narrow down the pool of adaptations to those with a significant impact on the KPI. The periodicity (Every) allows the system designer to describe how often the planning activity should try to improve a particular KPI. The system designer provides a time *period* in seconds, that when elapsed, an attempt to optimize the KPI will be made.

As mentioned above, the policy consists on several goals of the previous types, organized in *rank* order, from the most important to the less important. This rank allows a graceful degradation of the system behavior when it is not possible to fulfill all the goals. How this degradation is achieved depends on the evaluation criterion used. For rank-based policies, there are several criteria available and they are discussed in more detail in Section 5.2.5.

Below, we present two examples of goals for the running example. The first aims at keeping the system's redundancy level, by maintaining at least three active servers. The second goal states that, every 300 seconds, the self-management should attempt to maximize the service ratio. If the service ratio can be improved, this is only done if the gain is superior to 5%.

```
Goal preserve_redundancy: active_nodes Above 3
Goal maximize_service_ratio:Maximize service_ratio MinGain 0.05 Every 300
```

5.2.3 Offline Support: Rule Generator

The rule generator is an element exclusive to the goal-oriented planning. It parses a goal policy and automatically generates an *open* action policy, that will be used by the policy interpreter to manage the system. The rule generator operation consists in two steps: *event extraction* and *adaptation screening*. In this section, we address the two steps and the open action policies.

5.2.3.1 Event Extraction

The purpose of the first step of the rule generator operation is to identify the undesirable system states and generate the description of the events corresponding to those states. The events description is extracted automatically from the goal policy, using information from the knowledge model. At the end of this step, the event descriptions are added to a composite sensor that targets the system, and is included in the context model. From this point on, we will refer to the description of events simply as *events* to facilitate the reading.

The events are extracted from the goals in the policy. Exact goals result in sporadic events, while optimization goals result in periodic events. Table 5.1 summarizes the events generated for each type of goal. Sporadic events have a name and the triggering condition. The name of an event is extracted automatically from the goal, with a prefix. The prefix depends on the type of goal. For exact goals, there are two possible prefixes: *Above* and *Below*. For a Below goal, an *Above* prefix is added to the name of the goal and used as the name of the event. This indicates the states when the goal is not satisfied: when the KPI value is above the goal threshold plus the relevance margin. Similarly, for an Above goal, a *Below* prefix is added to the name. For a Between goal, two events are generated, one whose name has an *Above* prefix and another with a *Below* prefix. These two events are generated because the goal is a combination of Above and Below goals. The condition that triggers the event covers the undesired states, with the threshold considering the relevance margin.

Listing 5.5 provides examples of exact goals and the corresponding events description. From the exact Below goal named *cpu_reserve*, a composite event named *Above_cpu_reserve* is generated. The threshold value of the observable used to generate the event is automatically derived from the value of the goal and from the corresponding margin (in this case 0.75 plus the rele-

Listing 5.5: Examples of sporadic events extracted from exact goals

```
Goal cpu_reserve: avg_global_cpu_use Below 0.75

Composite Sensor
 Target: System
 Event
  Above_cpu_reserve
  when avg_global_cpu_use() > 0.76
  with current_cpu_use = avg_global_cpu_use()

Goal target_cpu: avg_global_cpu_use Between 0.4 0.6

Composite Sensor
 Target: System
 Event
  Below_target_cpu
  when avg_global_cpu_use() < 0.39
  with current_cpu_use = avg_global_cpu_use()
 Event
  Above_target_cpu
   when avg_global_cpu_use() < 0.61
   with current_cpu_use = avg_global_cpu_use()
```

vance margin, 0.01, thus 0.76). In a similar fashion, from the exact goal named *target_cpu*, two events are automatically generated, one is triggered when the observable goes below the lower threshold (*Below_target_cpu*) and the other when the observable goes above the higher threshold (*Above_target_cpu*).

Periodic events are tied to *optimization* goals and they are triggered every time the specified period of time elapses. These events have a mandatory name and a time period. In certain cases, the description may also include a triggering condition. The name of the event is generated by adding a prefix to the goal name. Two prefixes can be used: *Increase* and *Decrease*. For a Maximize goal, an *Increase* prefix is used, and for a *Minimize* goal a *Decrease* prefix is used. A *Close* goal generates two events. The name of one of the events has a *Decrease* prefix and cover all the states where the KPI value is above the target. Thus, the event also has a condition associated, so that the event is only triggered for those KPI values. The other event has a name that starts with the *Increase* prefix and covers all the states where the KPI value is below the target. It also has a condition associated. Any of the events have a time period associated that

Listing 5.6: Examples of periodic events extracted from optimization goals

```
Goal minimize_cpu: Minimize node_cpu_use MinGain 0.05 Every 100

Composite Sensor
 Target: System
 Event
  Decrease_minimize_cpu
   with current_cpu_use = avg_global_cpu_use()
   periodically 150

Goal target_cpu: node_cpu_use Close 0.3 MinGain 0.05 Every 200

Composite Sensor
 Target: System
 Event
  Decrease_target_cpu
   when avg_global_cpu_use() > 0.31
   with current_cpu_use = avg_global_cpu_use()
   periodically 200
 Event
  Increase_target_cpu
   when avg_global_cpu_use() < 0.29
   with current_cpu_use = avg_global_cpu_use()
   periodically 200
```

is extracted from the goal.

Listing 5.6 provides, for two different goals, the generated events. The first example describes a periodic event in a composite sensor, triggered every 150 seconds. The second example adds two events to a composite sensor, triggered every 200 seconds, but only if the observable *avg_global_cpu_use* is above 0.31 or below 0.29.

Goal Type	**Goal**	**Event A**	**Event B**	**Trigger**
Exact	Above	*Below*	-	lower threshold violated
Exact	Below	*Above*	-	upper threshold violated
Exact	Between	*Below*	*Above*	upper or lower threshold violated
Optimization	Maximize	*Increase*	-	periodic
Optimization	Minimize	*Decrease*	-	periodic
Optimization	Close	*Increase*	*Decrease*	periodic

Table 5.1: Event(s) generated for each type of goal

5.2.3.2 Adaptation Screening

The screening of adaptations is the second step of the rule generator operation. In this step, the adaptations in the adaptation model are analyzed to determine if they may contribute to achieve the goal in hand — the goal from which the event under consideration was extracted. The analysis consists in three distinct phases:

1. *Impact*: all the adaptations that do not have impact on the KPIs employed by the event are discarded.

2. *Usefulness*: all the adaptations that do not improve the KPIs employed by the event are discarded.

3. *Combinations*: all the compatible combinations of the screened out adaptations are generated.

The outcome of this step is a set of *viable* combinations of adaptations per event. Viable combinations are all the possible combinations of adaptations that may help achieve the goal, respecting any conflicts or dependencies. This also includes the empty combination that represents keeping the system as it is. Next, we address the three phases in more detail.

Impact: The first phase of the adaptation screening identifies the adaptations that are of interest to the goal in hand. We consider that an adaptation is of interest if it has impact on the KPI that is used in the goal. If the goal addresses a CKPI, then the adaptation has impact on at least one of the KPIs used in the CKPI description. In terms of adaptation description this translates in the following. All adaptations that declare any of the following impacts are selected for the next phase:

- has impact on the KPI, CKPI, or one of the KPIs (if the goal deals with a CKPI);

- has impact on any observable that is employed to build the KPI.

The first statement selects all the adaptations that declare a direct impact on any KPI of the goal in hand. The second statement covers the observables that are used, both directly and

indirectly, to build any of the KPIs of the goal in hand. The second statement is necessary because the human operator is only expected to declare the *known impacts* of an adaptation. This means that if the adaptation affects a KPI, then, there will be at least one impact that declares how the KPI, or any observable related to it, is expected to change. For instance, if we have a KPI A, built from B, who, in turn, is built from C; to address an event on KPI A, adaptations that have impact on A, B and C are selected. There are other cases, where the impact on an unrelated observable may denote an impact. For instance, let us assume an observable D that is built using C; an adaptation with impact on D, means that probably C also changes. Thus, the adaptation eventually affects KPI A and, thus, should be selected. However, despite an apparent impact, these adaptations are not selected due to implicit conflicts. These decisions regarding the second statement are discussed in more detail in the next paragraphs.

Usefulness: The second phase of the adaptation screening assesses the usefulness of the adaptations selected in the previous phase. We consider that an adaptation is useful to achieve the goal in hand, if the adaptation improves the KPI. The usefulness translates in selecting the adaptations whose impact on the KPI changes it in the same direction as necessary to achieve the goal. Thus, having a *positive* impact. For instance, the event extracted from the goal *preserve_redundancy* is triggered whenever the number of *active_nodes* is below 3. The adaptations that increase the number of *active_nodes* are the useful ones. Similarly, if the aim is to decrease a KPI, all the adaptations whose impact decrease the KPI are useful.

Assessing the usefulness of an adaptation offline solely relies on the described impacts of an adaptation. When the impact is declared regarding the target KPI, static analysis of the specified impact function can help to determine if the impact is positive or not. If it is impossible to automatically determine the direction of change for an impact function, this information can be provided by the component developer.

As previously addressed, the description of impacts is not exhaustive, thus, we cannot limit the adaptations of interest to those that declare a direct impact on the target KPI. As a result, the analysis must also look at adaptations that declare impacts on any observables used to build the KPI. However, we must be able to estimate the global impact on the KPI from the local impacts on the observables, described in the adaptation. The functions used in

the description of KPIs already cover this requirement, as they are limited to *monotonically non-decreasing* functions (as discussed in Section 5.2.1). Regarding the other observables, either they are basic sensed observables, or they are composite observables whose expression has to respect the properties of monotonically non-decreasing functions.

In all scenarios where the usefulness of the adaptation cannot be assessed, for instance, if the impact functions depend on context information, the impact is marked as *undetermined*. In these cases, the adaptation is always selected to avoid eliminating potentially useful adaptations. This guarantees that only adaptations known not to help achieve the goal are discarded.

For instance, consider the event *Above_cpu_reserve* from the previous section, where the goal is to decrease the value of the *cpu_use*. Consider also the adaptations listed in Section 4.4. The adaptation *toLowFidelty* declares an impact function that decreases the amount of current CPU use by 30%, $cpu\_use* = 0.7$. By calculating the derivative of the function $f(KPI) - KPI$, it is possible to see if the impact is positive or not. For instance the derivative of $x * 0.7 - x$ is always negative, thus the impact always decreases.

Combinations: The third phase of the adaptation screening is the generation of the viable combinations of the useful adaptations. While this phase could be performed online, it takes place offline to speed up the operation of the policy interpreter, by providing a set with only the viable combinations. A combination is viable if it respects the dependencies and explicit conflicts declared in the adaptation model, as well as the implicit conflicts, inherent to the approach.

To assess if a combination is viable it is necessary to *unfold* the adaptations first. The unfolding of an adaptation relies in predicting all the possible targets of that adaptation. The adaptations that need unfolding are those that can have more than one target. This applies to adaptations that target node and non-distributed component types. Adaptations that target distributed component types only need unfolding if they adapt local parameters, otherwise the target is only one. The prediction of the targets is tied to the number of nodes that the system may have at the time. The unfolding process uses the *max_nodes* as the number of nodes, thus, covering all the nodes that might be running. Thus, an adaptation that targets a node type will have as many variants as the number of *max_nodes* multiplied by the number of concrete node types to which the target type corresponds. The target of each variant is

different and is an instance of the node type, identified by the node type and an *id*, with $id = 1, ..., max\_nodes$. If the adaptation targets a component type, then there will be as many variants as the number of *max_nodes* multiplied by the number of concrete node types accepted in the system description and where that component type can run. The target of each variant is given by *getComponentInstance(CT,NI)*, where CT is the target component type and NI is one of the node instances that the system may have.

The unfolding process allows the generator to find all the possible variants of each adaptation, thus, it becomes possible to calculate all the combinations. The dependencies are important in this process to assess if a combination is viable. So are the conflicts, both the explicit (described in the adaptation model) and the implicit. The implicit conflicts allow to reject all adaptations or combinations where we cannot estimate the impact of applying the adaptation. This is the case of adaptations in the situation of the example of observable D, previously addressed in the discussion of the impacts. This is also the case of pairs of adaptations whose impacts overlap and thus it becomes impossible to determine the impact on the KPI. Two adaptations are implicitly conflicting if both:

- have impact on a common KPI;
- have the same target and impact over a common observable, which is used to build a KPI.

Adaptation requirements determine if an adaptation can be applied or not. However, these requirements refer to runtime, such as the configuration of some component, node, or system. Thus, they cannot be assessed offline and do not determine if a combination is viable or not.

The set of viable combinations will have the empty combination and combinations with size that can range from one to up to as many adaptations described in the adaptation model. At the end of this process, the rule generator has generated a set of combinations for each extracted event. The assembly of an event and all the combinations of adaptations are addressed next.

5.2.4 Open Action Policies

An open action policy is similar to a closed action policy with the one exception: the Do statement. In the closed action policy, the *Do* statement accepted adaptations with a strategy

Listing 5.7: Open action rule and example

```
When event
[With stateCondition]
 Do
   Select   {AC1 , AC2 , ...}

When Decrease_minimize_cpu
 Do
   Select   { [toLowFidelity , addNode] , [toLowFidelity] , [addNode] ,  ... }
```

and scope. In open action policies, the Do statement accepts a Select function that has all the viable combinations of adaptations $\{AC_1, AC_2, ...AC_m\}$ associated to the event. The event is declared in the When clause. The With clause is empty.

The Select function states that only one combination is to be selected from the set and the choice of the most appropriate combination is delegated to a runtime procedure. An open action rule has the format specified in Listing 5.7. The reader may wish to compare the format of an open action rule with that of a closed action rule introduced in Section 5.1.1. The listing also provides an example of a generated rule, despite not being complete.

5.2.5 Online Support: Policy Interpreter

We have just described how the offline support generates the open action policy. The open action policy is processed by the runtime support in a manner similar to that of closed action policies, with the obvious exception that, with the open action policy the Select function is invoked in runtime, to select the required adaptations from the set of viable combinations provided as input.

The interpreter starts the evaluation of the open action policy when it receives an event. The interpreter finds the only rule triggered by the event and starts the preparation task over the set of viable combinations. It begins by trimming the set of combinations, so that only the combinations with adaptations that can be applied to the current system state are kept. For this purpose, the preparation task starts by checking the target of the adaptation. If the target is the id of a node, it searches the system representation for it. If the target is a *getComponentInstance* operator, it executes the operator. The adaptations whose target does

not correspond to a instance in the system or whose set returned by the operator is empty are eliminated. It also eliminates adaptations targeting distributed component types that do not have at least one instance running in the system. If the target of an adaptation exists, the interpreter verifies the adaptation requirements and eliminates all the adaptations whose requirements are not satisfied. All the information necessary to perform the preparation task is available in the system representation and in the context analyzer. Any repeated combinations are eliminated. The preparation task is concluded with the calculation of the impacts on the KPIs involved in the goals of the policy. First, the estimated value for the KPIs are calculated per adaptation and, only after that, for the combination.

After the preparation task, the combinations in the rule are ready to be evaluated by the Select function during the selection task. The selection of a combination subsumes an optimization criterion and there are different reasonable choices. For instance, if we restrict our attention to policies with exact goals only, one possibility is to consider that being optimal means to satisfy as many goals as possible. In this case, the selection process must pick a combination of adaptations AC_i that maximizes the number of goals that are expected to be satisfied if those adaptations in AC_i are performed (the ranking order of the goals is ignored in this case). A different optimization criterion, which we call *ranked-eager*, is to take into account the rank of goals in policies and satisfy more important goals first. In this case, the selection process starts by picking the combinations that are expected to satisfy the highest ranked goal k, then among those are selected the combinations that are expected to satisfy the second ranked goal $k+1$, and so on. There is one exception: If all combinations selected in step k violate the $(k+1)^{st}$ goal, all combinations are selected. In this manner, if it is not possible to satisfy the $(k+1)^{st}$ goal without violating the k^{th} goal, the remaining goals are still used for tie-breaking. This selection mechanism is illustrated in Figure 5.4.

We have opted for an optimization criterion that is an extension of the *ranked-eager* criterion, which is also applicable to policies with optimization goals. Given that an optimization goal specifies a total order between the possible values of a KPI, when several combinations are evaluated against an optimization goal, are selected those that put the KPI close to the target specified in the goal. This is illustrated by Figure 5.5, where the best combination among those

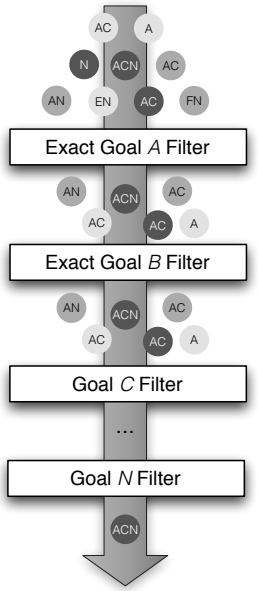


Figure 5.4: The ranked-eager criterion

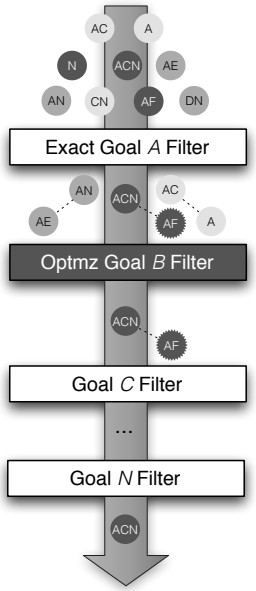


Figure 5.5: Evaluation of optimization goals

evaluated against the optimization goal B is represented by a multi-point star shape—the AF combination. Dashed lines are used to link combinations that are considered equivalent in what concerns goal B (the difference between their estimated impacts on the KPI of goal B is smaller than the relevance margin). In the example, when the optimization goal B is evaluated, only two combinations are selected. One is the combination that puts the KPI of B close to the target, and the other combination is an equivalent one. The next paragraphs present a more detailed and rigorous description of the whole selection process.

The process starts with a search space $\mathcal{SS} = \{AC_1, AC_2, ...AC_m\}$ of the viable combinations of adaptations. The search involves analyzing the estimated effects of the different combinations on the KPIs addressed by the goals and deducing which ones best fit these goals. More precisely, recall that policies define a set of ranked goals $\{G_1, ..., G_n\}$, where G_1 is the goal with the highest rank. The comparison between different combinations of adaptations relies on their evaluation against these goals, starting with G_1. In other words, each goal works as a filter that allows only a number of AC_i to pass. The filter of a goal G_i, which performs the evaluation of a combination of adaptations C against a goal G_i, depends on the type of goal (exact or optimization) and the estimated impact of the adaptations on the KPI_i associated with the goal. This value, which we denote by KPI_i^C, is calculated as follows:

- if KPI_i is not composite and employs both an AF and a CF, then KPI_i^C is the result of first applying CF to each target node of type *nodeType* (as long as the component instances offer the observable *from_obs_name*), and then apply AF to all nodes.
- if KPI_i is not composite and employs only an AF, then KPI_i^C is the result of applying AF to all the nodes of type *nodeType* (as long as the instances offer the observable *from_obs_name*).
- if KPI_i is not composite and employs only a CF, then KPI_i^C is the result of applying CF to all the instances of type *componentType* (as long as the instances offer the observable *from_obs_name*).
- if KPI_i is composite (a CKPI), then KPI_i is defined by a join function JF involving non-composite KPIs and hence, the value of KPI_i^C is obtained after calculating the estimated impact of C in these KPIs.

Note that if C is the empty combination, then KPI_i^C is just the current value of KPI_i.

The definition of C *matches* $\{G_1, ..., G_i\}$ is given by the *conjunction* of the three following conditions:

1. if $i > 1$, C matches $\{G_1, ..., G_{i-1}\}$
2. if G_i is an exact goal: KPI_i^C satisfies G_i or, for all other combinations C^+ in $\mathcal{SS}$ that match $\{G_1, ..., G_{i-1}\}$, $\text{KPI}_i^{C^+}$ also violates G_i.
3. if G_i is an optimization goal: $|KPI_i^C - KPI_i^{C^*}| < relevance\_margin^{KPI_i}$ where C^* is, among the combinations in $\mathcal{SS}$ that match $\{G_1, ..., G_{i-1}\}$, the one that puts the KPI_i closer to the target specified in G_i.

C *best matches* $\{G_1, ..., G_n\}$ only if *i)* C *matches* $\{G_1, ..., G_n\}$ and *ii)* for some $1 \leq i \leq n$, either G_i is an exact goal that is currently violated and KPI_i^C satisfies it; or G_i is an optimization goal and, compared to the current value of KPI_i, there is a gain that exceeds the specified minimum gain.

To illustrate the selection process, consider a goal policy whose highest ranked goal is *global_mem_use Below 0.58* and the second highest ranked goal is *avg_global_cpu_use Below*

0.36. Let's assume that the current *avg_global_cpu_use* value is 0.58 (the second goal is currently violated) and the rule available to deal with the violation of this goal is When *Above_avg_global_cpu_use* Select $\{AC_0, AC_1, AC_2, AC_3\}$, where AC_0 is the empty combination and all adaptations included in the others combinations AC_i decrease *avg_global_cpu_use* at the expense of increasing *global_mem_use*. During the evaluation of the rule triggered by the *Above_avg_global_cpu_use* event, the selection process starts by selecting the combinations whose estimated effects do not bring *global_mem_use* above 0.58. If the value of *global_mem_use* is already close to the limit and the estimated effect of all three non-empty combinations is to bring its value above 0.58, then the result of the selection process is to leave the system as it is (the selected combination is the empty one). On the contrary, if the value of *global_mem_use* is far from the limit and the estimated effects of, say, all but AC_3 keep *global_mem_use* value below 0.58, then the process would continue by selecting among AC_0, AC_1, AC_2, the combinations that are able to bring *cpu_use* below 0.36. Suppose this is the case with the two non-empty combinations. Then, the next ranked goal in the policy would be used to tie-break among them.

The rule evaluation mechanism has two implicit advantages. First, even if a goal is violated and cannot be satisfied, the evaluation process will continue to see if it can improve the system by satisfying other goals, such as optimization goals. Second, when it is not possible to satisfy all goals, the proposed approach provides graceful degradation according to the rank. As a result, the goals with lower rank will be violated first to maintain the more important goals.

Note that if an optimization goal ranks first in the policy, the rule evaluation mechanism will treat it in a greedy manner. This makes it possible to describe scenarios where we want to give preference to an optimization goal, hence, the system adapts mainly focused on that optimization.

5.2.6 Discussion

Besides the obvious advantage of handling complex adaptation logics, the goal-oriented planning facilitates several aspects tied to the human part of managing the system. Humans play two distinct roles in this approach: component developers and system designers. The component developers share their experience with the system, providing intel on how the system can be

adapt and of the benefits and trade-offs that can expect from a certain adaptation. The system designer or designers describe the expected behavior for the system, and what is considered desirable, by characterizing that behavior through a goal policy. This separation reduces the human effort in managing the system, allowing a more flexible handling of both the adaptations pool and the goal policy. The component developers can easily add new adaptations without being concerned with the existing ones. The system designer can easily change the goal policy without being concerned with the adaptations.

The key to using knowledge from both groups of human operators is the collection of KPIs. The goal policy is specified in function of these KPIs, and the adaptations describe impacts on the base observables of these KPIs. While for some systems, the necessary KPIs are obvious, for others several iterations may be necessary to reach an adequate set of KPIs. By describing the adaptation impacts over observables, it is possible to bypass this issues. While adding new KPIs do not necessarily affect existing adaptations, changing a KPI or the set altogether, will demand that the goal policy is re-specified.

If the KPIs bridge the gap between developers' knowledge and system designer's aims, the ranks make the specification of a goal policy a high-level task. At first glance, the expressiveness of rank-based policies might seem not ample enough for expressing more complex scenarios such as $G_1 : k_1$ *Below* v_1 is more important than $G_2 : k_2$ *Below* v_2 and G_2 is more important than $G_3 : k_3$ *Below* v_3 but we prefer to have G_2 and G_3 both satisfied than to have just G_1. However, with the definition of appropriate CKPIs, we can easily address this type of scenarios. Specifically, we would just have to define a CKPI $k_{23} = (v_2 \dot{-} k_2) * (v_3 \dot{-} k_3)$, a goal $G_4 : k_{23}$ *Above* 0, and a policy with the goals G_4, G_1, G_2, G_3, in this order. Note also that CKPIs can be used to define weight-based policies. For instance, one could also represent that we prefer the conjunction of G_2 and G_3 to G_1 by defining the following CKPI:

$$k_w = w_1 \cdot \text{signal}(v_1 - k_1) + w_2 \cdot \text{signal}(v_2 - k_2) + w_3 \cdot \text{signal}(v_3 - k_3)$$

where $signal(x) = 1$ if $x > 0$ and 0 if $x \leq 0$, and then setting the weights as $w_1 = 0.4$, $w_2 = 0.35$, and $w_3 = 0.25$, respectively. This together with the goal:

Goal G_w: **Maximize** k_w **MinGain** 0.01 **Every** 100

shows how we can describe goals with weights in the policy. We opted by having rank-based goal policy because our aim is to make the policy definition a high-level task. Using ranks is far simpler to specify and be sure of what the policy does, than if one needs to rely exclusively on weights. A weight-based approach demands that the operator decide exactly the weight of each goal compared to others—a task that is far more complex than simply identifying which goals are more important than others. Furthermore, it is harder for the operator to verify if the given weights have the effect she expects in terms of system behavior. Finally, the extension of a weight-based policy with an additional goal also demands more effort than simply adding the new goal somewhere in the policy.

Ultimately, the success of the goal-oriented planning is tied to a sensible goal policy, but, above all, to efficient descriptions of adaptations. When describing an adaptation, the component developer must effectively identify which KPIs will be affected by the adaptation. Failing to do so, may cause unexpected behaviors and the self-management will fail. Another aspect that is crucial is an accurate measure of the impact of the adaptation in the KPIs, or in other words, the impact functions. The impact functions are obtained through experience, either when developing the component or after testing and benchmarking the component. The more accurate is an impact function, the more effort and time it demands. It is important that impact functions are accurate enough so that the self-management is not misguided. The necessary level of accuracy will depend on the KPI and the adaptation. This is the part of an adaptation description that will take the most effort. However, any self-adaptive approach also spends the majority of its time/effort in trying to understand the effects, benefits and trade-offs of an adaptation to build the action policy.

Summary

This chapter describes how the proposed approach tackles the planning activity. It presents the two reifications: rule-oriented and goal oriented. For each reification, it covers the elements that intervene and how they operate in detail. The chapter also includes a discussion of both reifications, namely their limitations and challenges.

Change is not merely necessary to life; it is life.

Alvin Toffler, in *Future Shock*

Chapter 6

Adaptation Execution

There are many aspects that need to be considered when executing an adaptation. They range from how the execution should be performed to the actual implementation of the changes. This chapter discusses the aspects and considerations necessary to describe how an adaptation is executed. We begin by discussing the concerns and needs that motivated the decisions behind the execution support and the design of the effector and executor models. Next, we address in more detail the specification of strategies, namely, we present some of the existing strategies in the literature and how they can be described in the executor model. We conclude this chapter with the application of multiple strategies at one time, when a set of adaptations is the output of the planning activity.

Figure 6.1 shows how the execution support fits in the approach context. The main elements engaged in this activity are the effectors and the adaptation executor. However, the system representation manager is also tied to this activity as the system representation is necessary to execute the adaptations and the manager updates the representation after the execution.

6.1 Execution Needs and Runtime Support

The execution of an adaptation must be tailored to the change being made but also consider the target or targets and the system. The needs to perform a structural adaptation are often different from the ones of a behavioral adaptation, as the adaptation of a distributed component

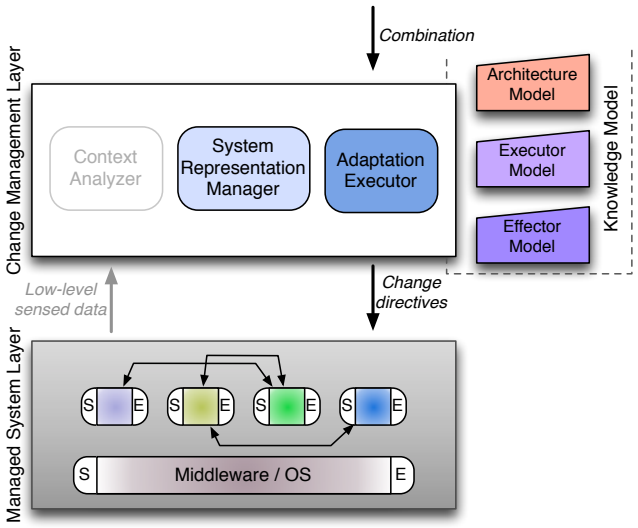


Figure 6.1: The adaptation execution in context

has needs that adapting a non-distributed component does not have. These needs vary greatly according to the adaptation, the target, and the system. Thus, there are several approaches that provide solutions for specific components or scenarios, as analyzed in Section 2.2.4.

The adaptation needs can be placed in three different categories according to the phase of the execution: before, during, and after the adaptation. The pre-adaptation needs refer to the preparation of a target or targets for adaptation. Depending on the component, this may involve stopping or pausing a component instance so that the adaptation can be safely performed, or reaching a quiescent state. If it is necessary to exchange a component instance by another, it may be useful to retrieve the state of the old component instance and use it to initialize the new one. In terms of needs during the actual adaptation, they are essentially the adaptations themselves. Finally, in terms of post-adaptation needs, it may involve resuming or starting a component instance, or, initialize a new one with previously captured state. These needs were already identified and translated into a set of commands described in Section 4.5, where the effector model is covered. Table 6.1 shows the commands and relates them to the type of concerns they address.

The design of the runtime support reflects not only the needs of behavioral and structural adaptations in distributed system, but also reflects these pre and post adaptation concerns. As depicted in Figure 6.2, the support consists in the adaptation executor, reconfiguration

Command	Pre	Adaptation	Post
addComponent		x	
removeComponent		x	
setParameter		x	
stop	x		
start			x
pause	x		
resume			x
makeQuiescence	x		
getState	x		
stateSet			x

Table 6.1: Adaptations associated with each concern

agents, and different types of effectors. As described in Section 3.6.2, the *adaptation executor* is responsible for executing the adaptations selected by the planning activity. The executor performs two tasks. One task is to combine different strategies, when more than one adaptation is selected during the planning. The other task is to control and coordinate the reconfiguration agents to execute the adaptations. Each node instance has a *reconfiguration agent* that works as an intermediary between the executor and the effectors. There are three types of effectors: system, node, and component. There is only one *system* effector that performs changes in terms of setting the value of a system parameter and adding or removing nodes. For each node instance, there is a *node* effector that is prepared to add and remove components from that node. Finally, each component instance has a *component* effector that is prepared to set the value of the parameters of the component and execute any pre and post-adaptation concerns. All effectors are oblivious to each other, even if they are tied to instances of distributed components. However, each effector registers the reconfiguration agent that controls it, and is aware and prepared to receive commands from the agent.

The reconfiguration agent directly controls the effectors in that node. The agent is prepared to receive a batch of commands from the executor and send each command to the target effector. For instance, if the agent receive a batch of two commands: *addComponent* and *start*; it will send the *addComponent* to the node effector, and the *start* command to the new component effector. The reconfiguration agent also handles the state transfer that may be required when one component is replaced by another component that provides the same service (for instance, to

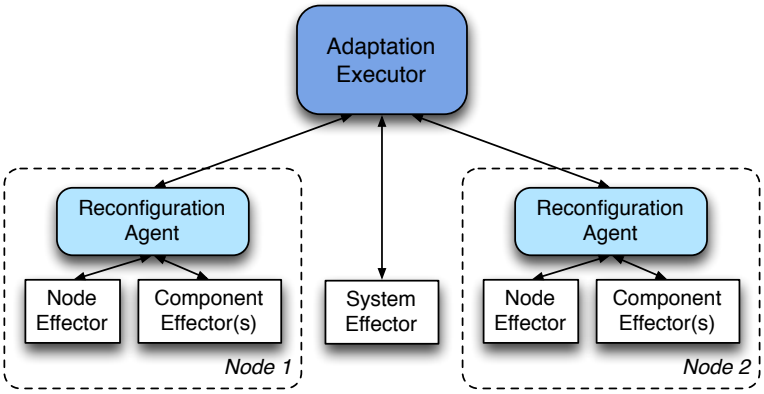


Figure 6.2: The interaction between executor, reconfiguration agents and effectors

deploy a software upgrade). The agent saves the state that results from a getState command for later use in a putState. In this way the state of the component, that may be large, is not required to be transferred across the network from the node to the executor and from the executor back to the node. Finally, after collecting the results of all commands in the batch, the agent returns them the executor.

It is important to note that, while the reconfiguration agent is prepared to transfer state during replacements in the node it is located, it is not prepared to transfer state from an existing component in a node and new component of the same type in another node. In this case, the state transfer must be done by the executor, who will retrieve the state and then send it to initialize the new component. In the particular case of distributed components, such as Infinispan, whose new instances must be initialized according to a number of configuration options, it is the component itself that controls the initialization of new instances in this case.

6.2 Specifying Strategies

Reconfiguration strategies define how an adaptation is applied through nodes and how is achieved locally in each node. From this point forward we will use *reconfiguration* to denote the changes made to the system to achieve an adaptation. The strategy describes a sequence of reconfigurations and pre and post-adaptation needs over the adaptation target. This target may be an instance of a component, (all the instances of) a distributed component, a node instance,

or the system, which is inherited from the selected adaptation.

The strategy itself is defined in terms of an *orchestration* protocol and a *local reconfiguration* technique. An orchestration defines the coordination among nodes, while the local reconfiguration technique describes how the reconfiguration is performed locally at each node. The combination of different orchestrations and local reconfigurations yields a large set of viable strategies, that can be applied in several distinct contexts.

Different adaptations typically have different coordination requirements. For instance, the reserved memory space for a caching service can be changed in multiple nodes without requiring inter-node coordination. However, the reconfiguration of the communication services used by the nodes (by replacing TCP by UDP, for instance), needs to be performed in a coordinated fashion, as communication would be impossible if each node is using a different communication service. Similarly, different adaptations may have different needs in terms of local reconfiguration, with varying costs and applicability constraints. For instance, changing the timeout value of a failure detector service in a given application can be performed on-the-fly with little cost; whereas replacing the implementation of a complex service may require to place the affected service(s) in a quiescent state, and even to capture and transfer part of the service's state to the new implementation. It is important to note that the local reconfiguration technique is, to some extent, independent of the orchestration adopted. Thus, we consider that is possible to reason about a reconfiguration strategy in terms of different combinations of distinct orchestrations and local reconfiguration techniques, as described in detail in the next paragraphs.

6.2.1 Orchestrations

An orchestration defines how nodes coordinate to perform the adaptation. Each orchestration depends on the exchange of messages between the adaptation executor and the reconfiguration agents or system effector. The exchange of messages allows the manager to communicate which reconfiguration needs to be performed and how, including if coordination is required. The coordination between the nodes involved in the reconfiguration works very much like a synchronization barrier, where none of the agents proceeds with further reconfiguration steps until all agents have completed the step and are ready for the next. An orchestration can have zero, one,

or multiple synchronization points. A *k-orchestration* will enclose k synchronization points and, therefore, $k + 1$ steps.

From our experience, there can be any number of orchestration steps in strategies, thus, we use a generic formulation where a strategy can have k steps: *k-orchestration*, with $k = 0, 1, ..., n$. In the literature, the most frequent orchestrations are those with $k = 0, 1, 2$, as depicted in Table 6.2.

k	Orchestration Name
0	Uncoordinated
1	Single Sync
2	Double Sync

Table 6.2: Orchestrations

6.2.2 Local Reconfiguration Techniques

A local reconfiguration technique is concerned with the manner the adaptation is performed locally, namely the reconfigurations that have to be performed to achieve it. One adaptation can be several reconfigurations together. Independently of the orchestration used, we have identified three different techniques to perform an adaptation in a given node:

- **Unprepared:** In this case, it is possible to execute the adaptation without any previous preparation of the adaptation target. This happens when the reconfiguration can be performed immediately. For instance, if the target is a component instance, then the reconfiguration can be performed without disturbing its operation or of the other component instances that interact with the target. Changing the fidelity level of a catalog component instance is an operation that can be performed immediately; the reconfiguration basically consists in changing a flag, that is consulted when a request arrives. Thus, it can be changed even in the middle of processing requests without causing failure to the request processing.

- **Prepared:** In this case, the adaptation cannot be executed without some preparation, otherwise the safety of the system may be affected. Consider for instance an adaptation that removes a total order protocol from a group communication system such as

Appia [MPR01] or Ensemble [vRBH+98]. In those systems, if the protocol is removed form the stack while the system is processing messages, it risks that messages are lost. Therefore, before the adaptation takes place, the group communication system needs to be placed in a quiescent state. The *prepared* local reconfiguration technique typically includes three phases: i) a pre-adaptation preparation phase; ii) an adaptation execution phase and; iii) a post-adaptation resume phase. Relevant actions that may be performed in the pre-adaptation preparation phase include: to place an instance in a quiescent state, capture the instance state, and to pause or stop the component instance. As noted above, relevant actions in the post-adaptation phase include to load the state on newly instantiated components, and to resume or start the execution of previously paused/stopped instances.

- **Assisted:** In this case, adaptation is performed with the assistance of a helper component that is instantiated locally so that the reconfiguration can be performed with minimal impact to the running system. The idea of using helper components has been used mainly in the context of protocol stacks, to avoid the interruption in the message processing caused by the *prepared* strategy discussed above. In this context, the helper components are called protocol switchers and multiplexers, as introduced in Section 2.2.4. Typically, an assisted technique also requires three phases: i) a pre-adaptation preparation phase, where the helper components are instantiated; ii) an adaptation execution phase, that is delegated to the helper components, and; iii) a post-adaptation phase, where the helper components are de-activated.

One final note regarding quiescence and achieving a quiescence state. Quiescence is often required in situations that demand a strong coordination between instances of the of the distributed component, but can also be required in a situation without coordination. For instance, when a service implementation is replaced by another compatible implementation (e.g. to install a bug-fix on-the-fly), it may not be necessary to coordinate the reconfiguration in all nodes. However, in each node, it may be necessary to put the service in a quiescent state to replace it.

6.2.3 Combining Orchestrations with Local Reconfiguration Techniques

A strategy is a combination of an orchestration and a local reconfiguration technique. We describe a number of relevant combinations employed in the examples and case-studies mentioned in this thesis. The description relies on the effector operators that were introduced in Section 4.5. It is not the goal of this section to explore or discuss the complete space of strategies that may be defined.

6.2.3.1 Flash

This strategy is adequate to adaptations that do not require coordination (i.e., that can be executed in a single step) and have no pre or post-adaptation concerns. It consists in a *0-orchestration* and a *unprepared* local reconfiguration technique. The strategy simply lists the commands corresponding to the changes made. For instance, an adaptation *toLowFidelity* that sets to *low* the *fidelity* parameter of a *Catalog* component instance can be performed using this strategy, as presented in Listing 6.1.

Listing 6.1: Example of a flash strategy

```
default toLowFidelity Strategy flash Parallel
 Step:
  effectorInstance(target).setParameter(fidelity, low)
```

The flash strategy can also be used to execute more complex adaptations, that consist in several reconfigurations, even pre and post adaptation concerns, as long as there are no needs for synchronization. For instance, the adaptation *deployComponents*, described in Listing 4.14, adds several component instances to an empty node and starts the instances. Since the node is not operational, all the reconfigurations can be performed in a single step without any synchronization.

Listing 6.2: Example of a flash strategy with multiple actions

```
default deployComponents Strategy flash Parallel
 Step:
   effectorsInstance(target).addComponent(WebServer)
   effectorsInstance(target).addComponent(Infinispan)
   effectorsInstance(target).addComponent(HomeCatalog)
   effectorsInstance(target).addComponent(BusinessCatalog)
```

```
...
forall e: effectorsDeployedInNode(target)
  e.start()
```

6.2.3.2 State-aware Flash

This strategy is suitable for adaptations that do not require coordination, but need to perform state transfer between components. It consists in a *0-orchestration* and a *prepared* local reconfiguration technique. The strategy is particular useful to replace a non-distributed component instance, retrieving the state of the old instance and initializing the new instance with it. This strategy can be used to upgrade the *Catalog*, *Account* and *User* components in the running example. Listing 6.3 presents one such adaptation for the *HomeCatalog* instances and the state-aware flash strategy that executes the upgrade.

Listing 6.3: Example of a state-aware flash strategy

```
Adaptation upgradeHomeCatalog:
 Target:
  ServerNode
 Impacts:
  target.componentRemoved(HomeCatalog)
  target.componentAdded(HomeCatalog2.1)
 Stabilization:
  60

default upgradeHomeCatalog Strategy state-aware flash
 Step:
   e_old = effectorSelect(HomeCatalog,target)
   e_old.getState()
   e_old.stop()
   effectorInstance(target).removeComponent(HomeCatalog)
   new = effectorInstance(target).addComponent(HomeCatalog2.1)
   e_new = effectorsInstance(new)
   e_new.putState()
   e_new.start()
```

6.2.3.3 Stop-and-go

This strategy is suitable for adaptations that require synchronization of all the adaptation targets before performing the reconfiguration, and after it. The strategy is composed by a

2-orchestration and a *prepared* local reconfiguration technique; it relies on a pre-adaptation preparation step, that halts the operation of all the targets. Only when all targets are halted, the strategy continues to the next step and the actual reconfiguration. After the successful reconfiguration of all the targets, their operation is resumed in a post-adaptation step.

The *activateTotalOrder* adaptation, presented in Listing 4.20 of Section 4.6, could be executed using this strategy, as it caters to all its needs. Listing 6.4 presents the strategy for this adaptation. In the first step, all the instances of Infinispan are paused. After that, in the second step, the value of the global parameter *multicast_properties* is set to *total* in all of the Infinispan instances. Finally, in the last step the operation of all Infinispan instances are resumed. It is worth of notice that in the particular case of Infinispan the parameter *multicast_properties* is set in all instances. This depends on the implementation of the components. In some components, it may be possible to set the value of the global parameter in one instance, and the component itself has mechanisms to propagate the new value to the remaining instances. In the case of Infinispan, while this parameter has to be equal in all instances, the component does not offer mechanisms to propagate the change.

Listing 6.4: Example of a stopAndGo strategy

```
default activateTotalOrder Strategy stopAndGo
 Step:
   forall e: effectorsComponentType(Infinispan)
     e.pause()

 Step:
   forall e: effectorsComponentType(Infinispan)
     e.setParameter(multicast_properties, total)

 Step:
   forall e: effectorsComponentType(Infinispan)
     e.resume()
```

6.2.3.4 Switcher-based

This strategy is customized for the exchange of distributed components with minimal interruption of the service. The strategy relies on having both the old and new component operating simultaneous with the support of helper components. The helper components route the requests

to one of components during the switching phase, until all requests are routed to the new component. There are several approaches to this type of strategy, as discussed in Section 2.2.4, some rely on a single helper component while others rely on two. The description of this strategy is inspired on the generic switching mechanism described in Section 2.2.4.3, but with a single helper component that handles all the incoming requests to the components. The strategy relies on a *4-orchestration* and in an *assisted* local reconfiguration technique.

Listing 6.5 presents the switcher-based strategy for the adaptation *Seq2TokenRAppia* that exchanges the implementation of the communication protocols in the *RAppia* framework [RRL07]. The adaptation exchanges two different implementations of total order protocols: sequencer-based is replaced by token-based. In the first step, an instance of the helper component and of the token protocol is added to all node instances. In the second step, the new instances are started. In the third step, the sequencer protocol is forced to achieve a quiescent state. In the last step, the sequencer protocol instances are stopped and removed, as well as the instances of the helper component. While this adaptation would be possible in the running example, the implementation of JGroups and Infinispan does not provide the support necessary for a switcher-based strategy. Thus, in the running example, the stop-and-go strategy would have to be used instead.

Listing 6.5: Example of a switcher-based strategy

```
Adaptation Seq2TokenRAppia:
 Target:
  ServerNode
 Impacts:
  forall sn: getNodeInstances(ServerNode, SequencerProtocol)
   sn.componentRemoved(SequencerProtocol)
   sn.componentAdded(TokenProtocol)
  ...
 Stabilization:
  100

default Seq2TokenRAppia Strategy Switcher-based
 Step:
   forall en: effectorsSelect(ServerNode, Sequencer)
     en.addComponent(Token)
     en.addComponent(Switcher)

 Step:
```

```
    forall ec: effectorsComponentType(Token)
      ec.start
    forall ec: effectorsComponentType(Switcher)
      ec.start

  Step:
    forall ec: effectorsComponentType(Sequencer)
      ec.makeQuiescent

  Step:
    forall ec: effectorsComponentType(Sequencer)
      ec.stop()
    forall en: effectorsSelect(ServerNode,Sequencer)
      en.removeComponent(Sequencer)
      en.removeComponent(Switcher)
```

6.3 Executing Adaptations

When the planning activity decides on a single adaptation, the adaptation executor will execute the corresponding strategy. If the rule-oriented planning is used, the strategy is given in the rule description; if the goal-oriented planning is used, the executor finds the default strategy for that adaptation and executes it. The commands are exchanged with the reconfiguration agents that oversee the affected effectors. However, the planning phase may select multiple adaptations to be applied. In this case, the adaptations are executed in the order as in the set of the adaptations, unless there are dependencies. When there are dependent adaptations, the dependency order (as specified in the adaptation model) is respected. For instance, if the selected adaptation is *addNode*, then, the depending adaptation will be performed after by order. Which means that the adaptation *deployComponents* will be performed next, and the *activateNode* adaptation is the final one to be executed (these adaptations were presented in Section 4.4).

However, not all adaptations need to be executed in some serial order. For instance, if the selected set of adaptations is a collection of *toLowFidelity* adaptations, each one changing the *fidelity* parameter to low in different component instances, then there is no conflict among them. Thus, they could be performed in parallel. However, the default execution mode described above would execute them one by one. Clearly, this may result in very inefficient execution plans. To address these scenarios, two strategies can be marked as *parallel*, which causes the execution of

Listing 6.6: Example of parallel strategies

```
Parallel
 Seq2TokenRAppia:Switcher-based AND toLowFidelity:Flash
 activateTotalOrder:Stop-and-go AND toLowFidelity:Flash
```

the affected adaptations concurrently, as presented in Listing 6.6. Two strategies are declared to be parallel with regard to each other, if they do not need to be executed serially on different or even in the same target. Thus, the description of Parallel states that any subset of the set described as parallel can be executed in parallel.

The adaptation executor explores the strategy parallelism to improve the execution of multiple adaptations performing two procedures: first *parallel strategy grouping* and then *strategy step merging*. The purpose of parallel *strategy grouping* is to combine the strategies in the minimum number of disjoint groups, such that all strategies in the same group can be executed in parallel. The purpose of parallel *step merging* is to generate a *plan* that allows to execute strategies that have been placed in the same group, and, thus, can be safely executed together. A plan has as many steps as the strategy with the largest number of steps. Each step is the collection of all the commands in that step of all the strategies in the group. For instance, the first step is the collection of all the commands of all the first steps in all the strategies. The order between commands in the same step of one strategy is maintained when the plan is generated. The execution of the plan is similar to a strategy, where the commands in the same step are executed in serial order, and a new step is only started after all the commands of the previous step are concluded.

Consider again the strategies associated with the adaptations *toLowFidelity* (in Listing 6.1) and *activateTotalOrder* (in Listing 4.20). Assume that as a result of the planning phase one needs to execute the adaptation *toLowFidelity* using the *flash* strategy on instances of *HomeCatalog* $hc1$ and $hc2$, and the adaptation *activateTotalOrder* using the *stopAndGo* strategy on component Infinispan with instances $ifn1$ and $ifn2$. Since these two strategies can be performed in parallel (see Listing 6.6), the plan that would execute these strategies in parallel would be the following.

Listing 6.7: Combined Execution Plan

```
Step:
```

```
  effector(hc1).setParameter(fidelity,low)
  effector(hc2).setParameter(fidelity,low)
  effector(ifn1).pause()
  effector(ifn2).pause()
Step:
  effector(ifn1).setParameter(multicast_properties,total)
  effector(ifn2).setParameter(multicast_properties,total)
Step:
  effector(ifn1).resume()
  effector(ifn2).resume()
```

One noteworthy detail is that the stabilization period after the execution of the adaptations will be the larger of the adaptations involved.

6.4 Discussion

The execution of adaptations depends on a large number of low-level aspects and properties of the system. The description of some strategies is so tied to the component that only a component developer, aware of all the details, is equipped to describe them. The *stop-and-go* strategy for the *activateTotalOrder* adaptation is one such example. While the system designer could describe such a strategy, she would need to have detailed knowledge about the component characteristics and its operation. The number of low-level aspects that are part of a strategy also difficult the task of improving the simultaneous execution of multiple adaptations. This approach follows a conservative approach to guarantee that the execution is always safe, with base on developers' and designers' knowledge.

The proposed approach allows that an adaptation may be executed by different strategies. In these cases, there are trade-offs between the strategies, very often in terms of resource cost and interruption time. The best strategy for a given adaptation depends of many factors, such as the adaptation target, the system implementation, the load, and any SLAs of the application, among many other factors. In the approach, the best strategy is either defined in the rules of the closed action policy or is marked as *default*. In the latter case, the selection of a default strategy is a compromise that that strategy is probably the best in the majority or in the common scenarios.

There are other solutions that attempt to make a better selection of the strategies, relying

on the study of the trade-offs involved in using one or the other strategy. In this topic we have made some research in the following topics:

- We have investigated the tradeoffs involved in using generic switchers (that can be applied to different communication protocols) and specialized switchers [FRR09]. That research has shown that it is advantageous to support multiple switchers in the same architecture, as specialized switchers can outperform a generic switcher.
- We have studied how to augment the Appia group communication framework to support switcher-based adaptation. This research culminated in the proposal of a number of new mechanisms for the Appia system [RRL07], that have been later implemented in the context of a MSc dissertation by another member of the group [Tav10].

While the study of the trade-offs of different strategies and selection of the best strategy are topics of interest in adaptive systems, they are somehow orthogonal to the main topic of this thesis. Therefore, we have opted to omit from the text further details on these ramifications of our work.

Summary

This chapter addresses the execution of adaptations selected by the policy interpreter. We discuss the main concerns of executing adaptations and the runtime support necessary to address those concerns. We address in detail the specification of strategies, which consist in an orchestration and a local reconfiguration technique. We provide examples of the most common strategies and how they are described in the approach. We also analyze how different strategies can be combined to improve the execution time of multiple adaptations. The chapter concludes with a discussion of how the low-level details have impact on the description of strategies and other work done in this topic.

Ever tried? Ever failed? No matter. Try again.
Fail again. Fail better.

Samuel Beckett in *Worstward Ho*

Chapter 7

Evaluation

In this chapter we cover the evaluation of the proposed approach. The evaluation is based on the case study introduced in Section 3.1 and is divided in two parts. While the system is always distributed, the first part addresses only non-distributed components. The second part focus solely on how the distribution of components is handled. The case study also illustrates how to apply in practice the concepts and abstractions proposed in this thesis.

7.1 Non-Distributed Components

In this section, we describe how the approach can be applied to the non-distributed components of the case study introduced in Section 3.1. To analyze the scalability and performance of the approach, we opted by synthetically generated adaptations to increase the set of adaptations for evaluating scalability and performance. The results obtained are presented in Section 7.1.6.

7.1.1 Revisiting the Case Study

We start by considering only the non-distributed components of the case study, as depicted in Figure 7.1. This involves the following components: *Catalog*, *User*, and *Account*. The *Catalog* component handles the product webpages, which consist in static content only. The *User* component handles dynamically generated content, customized to the user, and relies on the *Recommendation* and *Search* engines, at a different machine. The *Account* component

handles sensitive content. The *Catalog*, *User* and *Account* components serve two types of users: *business* and *home*. The server executes all these components as well as the Apache HTTP web server, which is not adapted.

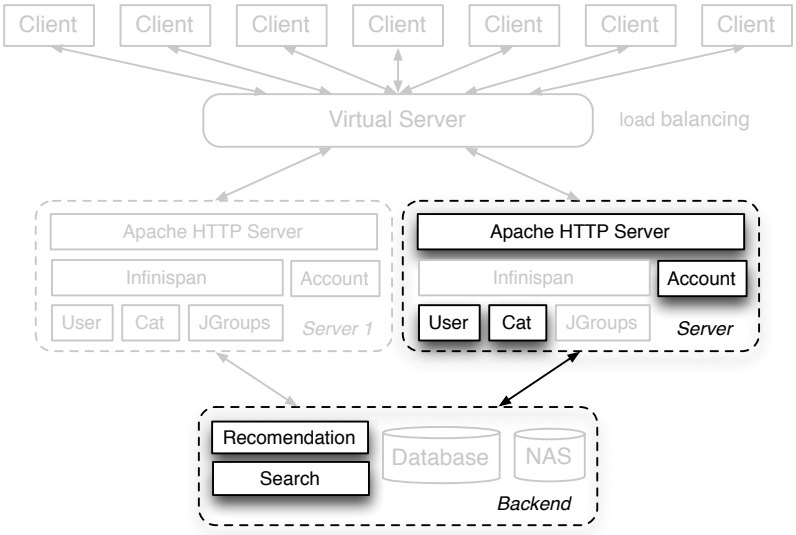


Figure 7.1: Case study: focus on non-distributed components

The website relies on a classical three-tier architecture: presentation, application, and data tiers. The application logic is executed in the middle-tier, typically in an *application server*; and the data tier consists of a database and its management services executed in a *data server*. In the case study, we use different machines to run the application server and the data server, as depicted in Figure 7.2. The first machine runs a web server and the set of components that implement the application logic. The data server runs the search and recommendation engines, as well as the database that stores the catalog and user data. Despite this separation in different nodes, in this section, this is transparent for for the self-management, thus, no node description is required.

In the next sub-sections, we describe the knowledge model and goal policies employed for this case study.

7.1.2 Knowledge Model

In this section, we describe the relevant portions of the knowledge model for the case study. While the case study also includes the recommendation and search engines in a backend server,

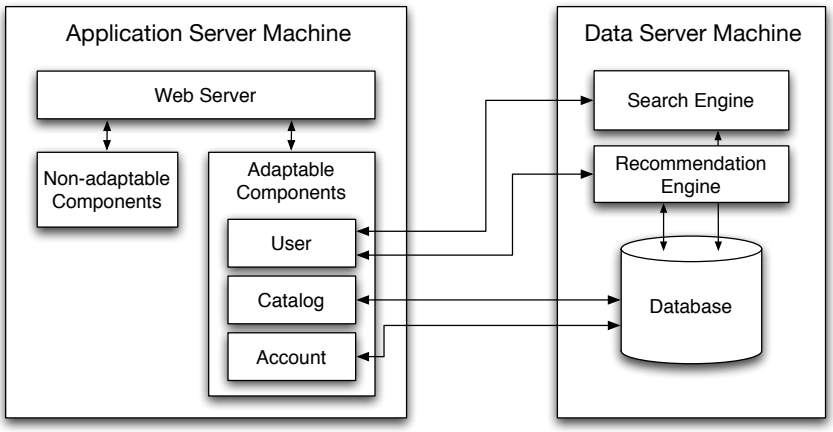


Figure 7.2: Interaction between components

they are not adapted directly. Therefore, we will focus on the adaptable components: the *Catalog*, *User* and *Account* components. We will start with the description of the architecture model, followed by the sensor and context models, the adaptation model, and finally the effector and executor models.

7.1.2.1 Architecture Model

As previously mentioned, the components in the case study are of types *WebServer*, *Catalog*, *User*, *Account*, and *Engine*, running in two types of nodes: *ServerNode* and *Backend*. Next, we describe all the component and node types, as well as the system. Their description is presented in Listing 7.1.

The components of type *Catalog* provide the product descriptions pages, which are static content webpages sent in a non-secure manner to the client. This component type has two sub-types: *HomeCatalog* and *BusinessCatalog*, for home and business users, respectively. The difference between components of both sub-types relies on the content provided to clients, namely, different prices and information displayed. Both component types have two *fidelity* modes: *regular* and *low*; in *low* fidelity the components offers lower image quality.

The components of type *User* generate user customized webpages, which are sent in a non-secure manner. The components generate webpages with personalized product recommendations and searches, when the user is logged in. This component type has two different sub-types: *HomeUser* and *BusinessUser*, which rely on *search* and *recommendation engines* [LS98, MCS00].

Both engines can provide *fresh* and *cached* results. The former demands more resources and takes longer to be generated, while the latter consumes fewer resources and can be provided with faster processing times. In the search engine, the fresh content is a list of products that fit the search keywords, sorted by current popularity indexes [LSY03]. The cached content is a previously generated product list, whose popularity indexes may no longer be up to date. The recommendation engine provides a list of recommended products, a content that is used to customize the web experience to a particular user. In the recommendation engine, the fresh recommendations depend on information regarding the user session and previous orders while the cached recommendations are produced from time to time or are the result of previously generated recommendations.

The components of type *Account* handle webpages that deal with sensitive user information. This information refers to account login, credit card information, billing and shipping addresses, and account settings, among others. There are two different sub-types of *Account*: *HomeAccount* and *BusinessAccount*. The latter includes invoice and budget management features. Both component types have two fidelity modes: *regular* and *low*.

The node of type *ServerNode* handles the applications server, while the node of type *Backend* handles data server. According to the description, the System can include the two types of nodes and can have at most two nodes.

Listing 7.1: Case study (ND): architecture model

```
Abstract Component StoreService
Abstract Component Business
Abstract Component Home
Abstract Component Engine

Abstract Component Catalog
    subtype StoreService
    Parameters
        fidelity:{regular,low}

Component BusinessCatalog
    subtype Catalog, Business

Component HomeCatalog
    subtype Catalog, Home

Abstract Component User
```

```
    subtype StoreService
    Parameters
        search:{fresh,cache}
        recommendation:{fresh,cache}

Component BusinessUser
    subtype User, Business

Component HomeUser
    subtype User, Home

Abstract Component Account
    subtype StoreService
    Parameters
        fidelity:{regular,low}

Component BusinessAccount
    subtype Account, Business

Component HomeAccount
    subtype Account, Home

Component SearchEngine
    subtype Engine

Component RecommendationEngine
    subtype Engine

Component WebServer

Node ServerNode
    Parameters
        is_active: boolean
    Components
        WebServer
        Catalog
        User
        Account

Node Backend
    Components
        Engine

System
    Parameters
        max_nodes = 2
    Nodes
        ServerNode
        Backend
```

The system initial configuration is presented in Listing 7.2.

Listing 7.2: Initial system configuration

```
System
  ServerNode = 1
  Backend = 1
Nodes
  ServerNode:{is_active=true}:{WebServer, HomeCatalog, BusinessCatalog, HomeUser,
      BusinessUser, HomeAccount, BusinessAccount}
Components
  Catalog:{fidelity=regular}
  User:{recommendation=fresh, search=fresh}
  Account:{fidelity=regular}
```

7.1.2.2 Sensor and Context Models

Online retail web sites face dynamic workloads, with predictable periods of overload, such as holiday season, but also with unexpected peaks, such as flash crowds [AHM+03]. The adaptive behavior in this case study aims at avoiding overload and at maintaining an appropriate balance between resource consumption and service quality. Thus, the description of the sensor and context models must address all these aspects.

In terms of load and resources, we consider that the load of the application server is captured by the consumed CPU time (for a discussion of other metrics that could have been used, see for instance [GMMR06, DHPB03]). The closer this resource is to the limit, the closer the system is to overload. By maintaining the CPU consumption below a certain value, it is possible to avoid the system overload. For the data server, the number of queries made to the engines is a good indication of the load. The higher the number of queries, the higher is the load. Similarly to the application server, maintaining the number of queries below a certain threshold helps to avoid the overload of the data server.

Regarding the service quality, it can be expressed as a combination of different factors: *Catalog* and *Account* components may provide content to the client with different fidelity levels; *User* components also have impact on the service quality due to the freshness of recommendations and searches provided to the client.

Listing 7.3: Case study (ND): sensor model

```
Sensor
 Target: StoreService
 Observable double CPUuse(): RMargin 0.01 periodically: 30
 Observable integer ProcessingTime(): RMargin 0.2 periodically: 30

Sensor
 Target: Engine
 Observable integer queries(): RMargin 10 periodically: 30

Sensor
 Target: system
 Observable integer hm_services(): RMargin 0 periodically: 30
 Observable integer bsn_services(): RMargin 0 periodically: 30
```

Listing 7.3 presents the sensor model for the case study. The model has descriptions of sensors for components of type *StoreService* and *Engine*, and for the system. The instances of type *StoreService* will have a sensor that provides information on the CPU consumption every 30 seconds. The *WebServer* component will provide metrics on the mean processing time for requests made to the business and home services separately. The components of type *Engine* give information regarding the number of queries received. The system will provide information regarding the number of active business and home services.

Listing 7.4 presents the context model for the case study. The model describes the composite observables that are used in the description of the system-wide observables that will be used as KPIs. Table 7.1 summarizes all the KPIs and CKPIs described in the context model.

Listing 7.4: Case study (ND): context model

```
Composite Sensor
 Target: Catalog
 Observable
   integer fidelity(): RMargin 0
   periodically:60
   target.fidelity

Composite Sensor
 Target: Account
 Observable
   integer fidelity(): RMargin 0
   periodically:60
   target.fidelity
```

```
Composite Sensor
 Target: BusinessUser
 Observable
   integer recmmdBsn(): RMargin 0
   periodically:60
   target.recommendation
 Observable
   integer searchBsn(): RMargin 0
   periodically:60
   target.search

Composite Sensor
 Target: HomeUser
 Observable
   integer recmmdHm(): RMargin 0
   periodically:60
   target.recommendation
 Observable
   integer searchHm(): RMargin 0
   periodically:60
   target.search

Composite Sensor
 Target: system
 KPI System-wide Observable
   integer resolution_bsn(): RMargin 0
   periodically:60
   from fidelity for Business with AF: Sum CF: Sum
 KPI System-wide Observable
   integer resolution_hm(): RMargin 0
   periodically:60
   from fidelity for Home with AF: Sum CF: Sum
 KPI System-wide Observable
   double recommend_bsn(): RMargin 0
   periodically:60
   from recmmdBsn with AF: Sum CF: Sum
 KPI System-wide Observable
   double recommend_hm(): RMargin 0
   periodically:60
   from recmmdHm with AF: Sum CF: Sum
 KPI System-wide Observable
   double search_bsn(): RMargin 0
   periodically:60
   from searchBsn with AF: Sum CF: Sum
 KPI System-wide Observable
   double search_hm(): RMargin 0
   periodically:60
   from searchHm with AF: Sum CF: Sum
```

```
KPI System-wide Observable
  double ptr_bsn(): RMargin 0
  periodically:60
  from ProcessingTime for Business with AF: Sum CF: Sum
KPI System-wide Observable
  double search_hm(): RMargin 0
  periodically:60
  from ProcessingTime for Home with AF: Sum CF: Sum
KPI System-wide Observable
  integer query_load(): RMargin 10
  periodically:60
  from queries with AF: Sum CF: Sum
KPI System-wide Observable
  double cpu_use(): RMargin 0.01
  periodically:60
  from CPUuse with AF: Sum CF: Sum

Composite Sensor
 Target: system
KPI Composed-System-wide Observable
  double mrt_bsn(): RMargin 0.2
  with JF: (2*ptr_bsn)/bsn_services
KPI Composed-System-wide Observable
  double mrt_hm(): RMargin 0.2
  with JF: (2*ptr_hm)/hm_services
KPI Composed-System-wide Observable
  integer qlt_bsn(): RMargin 0
  with JF: resolution_bsn+recommend_bsn+search_bsn
KPI Composed-System-wide Observable
  integer qlt_hm(): RMargin 0
  with JF: resolution_hm+recommend_hm+search_hm
```

The last four descriptions in the context model presented in Listing 7.4 are CKPIs. They aggregate different KPIs to provide a more general view of certain system aspects. One aspect is the overall quality of service. *Catalog* and *Account* components can provide images with different quality, while *User* components can provide different levels of freshness for search and recommendation results. The CKPIs *qlt_bsn* and *qlt_hm* gather the metrics referring to all component to give the overall quality by business and home services. The other aspect is the system processing time by business and home services, which is given by CKPIs *mrt_bsn* and *mrt_hm*, which give feedback on the processing time for requests, independently of the request distribution (IRD).

The context model also includes composite events, but they are generated automatically by

Name	Description
cpu_use	CPU consumption
resolution_bsn	Image quality of business services
resolution_hm	Image quality of home services
recommend_bsn	Freshness of business recommendations
recommend_hm	Freshness of home recommendations
search_bsn	Freshness of business searches
search_hm	Freshness of home searches
ptr_bsn	Mean processing time of business requests
ptr_hm	Mean processing time of home requests
query_load	Rate of search and recommendation queries
mrt_bsn	Mean processing time IRD
mrt_hm	Mean processing time of IRD
qlt_bsn	Overall quality of business services
qlt_hm	Overall quality of home services

Table 7.1: KPIs and CKPIs used in the case study.

the rule generator. Thus, they are addressed in Section 7.1.4.

7.1.2.3 Adaptation Model

As discussed before, all *Catalog*, *User* and *Account* components have configuration parameters that can be changed at runtime. By taking advantage of their adaptability capabilities, we have defined 32 adaptations. The impact values used in the case study are either obtained from the component developers, such as impacts regarding changes to image quality, or from benchmarks, for instance to compare cache retrieved recommendations and recommendations generated on request. To ensure some variety in terms of adaptation impacts, even similar adaptations (such as the ones described in Listing 7.5) have different impacts. Below, we detail the description of two adaptations.

The adaptations change the fidelity of *Catalog* components from regular to low. As stated in the *Impacts* statements, they reduce CPU consumption at the cost of degrading the service quality. In fact, the decrease in image quality reduces the CPU consumption and also the request processing time. Note also that the degradation of service quality in the business catalog allows a larger cut in CPU consumption than in the home catalog; this is due to the larger number and size of images that are involved in the business service.

The case study includes a total of 32 adaptations in total. The complete listing of all the adaptations is presented below:

Adaptation ToLowCatalogBusiness

Listing 7.5: Case study (ND): excerpt of the adaptation model

```
Adaptation ToLowCatalogBusiness
 Component:
  BusinessCatalog
 Requires:
  target.fidelity == regular
 Impacts:
  target.setParameter(fidelity,low)
  target.CPUuse ÷= 2.01
  target.fidelity −= 1
  target.ProcessingTime ÷= 1.99
 Stabilization:
  period = 60

Adaptation ToLowCatalogHome
 Component:
  HomeCatalog
 Requires:
  target.fidelity == regular
 Impacts:
  target.setParameter(fidelity,low)
  target.CPUuse ÷= 1.92
  target.fidelity −= 1
  target.ProcessingTime ÷= 1.4
 Stabilization:
  period = 60
```

Adaptation ToRegularCatalogBusiness
Adaptation ToLowCatalogHome
Adaptation ToRegularCatalogHome
Adaptation ToFreshRecommBusinessKeepSF
Adaptation ToFreshRecommBusinessKeepSC
Adaptation ToCacheRecommBusinessKeepSF
Adaptation ToCacheRecommBusinessKeepSC
Adaptation ToFreshRecommFreshSearchBusiness
Adaptation ToFreshRecommCacheSearchBusiness
Adaptation ToCacheRecommFreshSearchBusiness
Adaptation ToCacheRecommCacheSearchBusiness
Adaptation ToFreshSearchBusinessKeepHF
Adaptation ToFreshSearchBusinessKeepHC
Adaptation ToCacheSearchBusinessKeepHF
Adaptation ToCacheSearchBusinessKeepHC
Adaptation ToFreshRecommHomeKeepSF
Adaptation ToFreshRecommHomeKeepSC
Adaptation ToCacheRecommHomeKeepSF
Adaptation ToCacheRecommHomeKeepSC
Adaptation ToFreshRecommFreshSearchHome
Adaptation ToFreshRecommCacheSearchHome
Adaptation ToCacheRecommFreshSearchHome
Adaptation ToCacheRecommCacheSearchHome
Adaptation ToFreshSearchHomeKeepHF
Adaptation ToFreshSearchHomeKeepHC
Adaptation ToCacheSearchHomeKeepHF
Adaptation ToCacheSearchHomeKeepHC
Adaptation ToLowAccountBusiness
Adaptation ToRegularAccountBusiness
Adaptation ToLowAccountHome
Adaptation ToRegularAccountHome

7.1.2.4 Effector and Executor Models

The *Catalog*, *User*, and *Account* components are targets of adaptations, therefore, each one has an effector associated. All the effectors are prepared to handle the component state, achieve quiescence, and control the component operation by starting, resuming, stopping and pausing the execution. The description of *Catalog* and *Account* effectors is similar, as they have a same name adaptable parameter. The description of the *User* effectors focus on the two parameters that can be adapted. The description of the effectors is presented in Listing 7.6.

The adaptations available in the adaptation model do not have any execution requirements. Without pre, during, and post-adaptation needs, the adaptations can be executed using the

Listing 7.6: Case study (ND): executor model

```
Effector
 Target:
  Catalog
 Commands
  makeQuiescent()
  getState()
  putState(state)
  setParameter(fidelity,{low,regular})
  start()
  stop()
  pause()
  resume()

Effector
 Target:
  User
 Commands
  makeQuiescent()
  getState()
  putState(state)
  setParameter(search,{fresh,cache})
  setParameter(recommendation,{fresh,cache})
  start()
  stop()
  pause()
  resume()

Effector
 Target:
  Account
 Commands
  makeQuiescent()
  getState()
  putState(state)
  setParameter(fidelity,{low,regular})
  start()
  stop()
  pause()
  resume()
```

Listing 7.7: Case study (ND): excerpt of the executor model

```
default ToLowCatalogBusiness Strategy flash
 Step:
  effectorInstance(target).setParameter(fidelity,low)

default ToFreshRecommFreshSearchBusiness Strategy flash
 Step:
  effectorInstance(target).setParameter(recommendation,fresh)
  effectorInstance(target).setParameter(search,fresh)
```

most simple strategy: *flash*. Listing 7.7 provides examples of strategies for two adaptations, representative for the remaining adaptations and their strategies. All the strategies can be executed in parallel. Thus, the description of the parallel adaptations includes all the strategies. It is important to note, that despite some adaptations being conflicting because they have the same target, the Parallel description does not override the conflicts, it simply states that any subset of the adaptations and strategies can be executed in parallel.

7.1.3 Goal Policies

We have considered two different goal policies to guide the self-management. The reasons to present different policies are twofold. For one, to illustrate that it is possible to have different business strategies with the same knowledge model. The other reason is that it shows how the rank of goals affects the selection of adaptations, and that it is possible to have different behaviors just with a change in the rank order.

The goal policies have been designed to avoid system overload, while offering the best possible service quality. To avoid system overload, both the application server and the data server must be observed. We have considered relevant two different metrics: the CPU usage and the number of queries made to the search and recommendation engines. A limit to the CPU usage in the application server makes it possible to avoid its overload. Since several processes compete for the CPU, the limit only refers to the usage by adaptable components. For the same purpose, a limit to the number of queries is imposed on the data server.

Listing 7.8 and 7.9 present the policies used. They reflect the insights provided by related research, including policies to achieve optimal resource use for web servers [DHPB03, AB99],

Listing 7.8: Goal policy A

```
Goal limit_cpu: cpu_use Below 0.35
Goal limit_query_load: query_load Below 1200
Goal limit_mrt_bsn: mrt_bsn Below 1.6
Goal max_qlt_bsn: Maximize qlt_bsn MinGain 1 Every 900
Goal limit_mrt_hm: mrt_hm Below 1.9
Goal max_qlt_hm: Maximize qlt_hm MinGain 1 Every 1300
```

Listing 7.9: Goal policy B

```
Goal limit_cpu: cpu_use Below 0.35
Goal limit_query_load: query_load Below 1200
Goal limit_mrt_hm: mrt_hm Below 1.9
Goal max_qlt_hm: Maximize qlt_hm MinGain 1 Every 1300
Goal limit_mrt_bsn: mrt_bsn Below 1.6
Goal max_qlt_bsn: Maximize qlt_bsn MinGain 1 Every 900
```

intermediary adaptation systems [Maz06, IMS06, GMMR06], and web server and user experience improvement [Sou08]. The thresholds used in the goals were obtained from experience, by benchmarking the system.

Policy A starts with the *limit_cpu* and *limit_query_load* goals. The purpose of these goals is to avoid overloads by limiting the *cpu_use* of adaptable components in the application server and limiting the engines' *query_load* in the data server. In this manner, it is possible to avoid resource exhaustion, without causing underuse of resources. The *limit_mrt_bsn* and *limit_mrt_hm* goals refer to processing time, limiting the processing time for a request. Finally, the *max_qlt_bsn* and *max_qlt_hm* goals refer to content quality, urging the maximization of the content quality. The goals referring to the business clients come first in this policy to reflect their priority.

In contrast, policy B gives priority to home clients. The same goals are employed, however, using a different order: the goals referring to home clients are given higher priority then the equivalent goals for business clients. By switching between policies A and B one can favor a given type of clients according to some business target, for instance, by favoring the type that provides a better revenue at a given point in time.

7.1.4 Open Action Policy

The open action policy is built using the events extracted from the goal policy and the adaptations selected from the adaptation model. There are as many rules in the policy, as there are triggers. The triggers and events extracted from the goals are presented in Table 7.2. Thus, the open action policy has six rules.

Type	**Goal**	**Event**	**Trigger**
Exact	*limit_cpu*	Above_limit_cpu	*cpu_use*$> (0.35 + 0.01)$
Exact	*limit_query_load*	Above_limit_query_load	*query_load*$> (1200 + 10)$
Exact	*limit_mrt_bsn*	Above_limit_mrt_bsn	*mrt_bsn*$> (1.6 + 0.2)$
Optimiz	*max_qlt_bsn*	Increase_max_qlt_bsn	Every 900 s
Exact	*limit_mrt_hm*	Above_limit_mrt_hm	*mrt_hm*$> (1.9 + 0.2)$
Optimiz	*max_qlt_hm*	Increase_max_qlt_hm	Every 1300 s

Table 7.2: Events extracted from the goals used in the case study.

Besides the event, a rule also includes the viable combinations of all the adaptations screened for that event. For instance, the rule activated by the event *Above_limit_cpu* requires adaptations that decrease the CPU use, as those presented in Section 7.1.2.3. Similar adaptations for the *Account* component also decrease the CPU use. Thus, as presented in Listing 7.10, the rule will have several combinations of all these adaptations.

Listing 7.10: *Above_cpu_use* rule

```
When above_cpu_use
Do
  Select { [ToLowCatalogBusiness], [ToLowCatalogHome], [ToLowAccountBusiness], [
      ToLowAccountHome], [ToLowCatalogBusiness, ToLowCatalogHome], ...,
      [ToLowCatalogBusiness, ToLowCatalogHome, ToLowAccountBusiness,
      ToLowAccountHome] }
```

The open action policies generated for both goal policies have the same rules, since the order is not used in this process. As mentioned before, the open action policy has 6 rules, one for each goal of the policy. Each rule in the policy may have between 4 to 10 useful adaptations, used to generate the viable combinations. The rule with the largest number of combinations has 35 and the largest combination has 4 adaptations

7.1.5 Implementation and Experimental Setup

We conducted a study to evaluate the proposed approach, namely to analyze how successfully the rules generated offline drive the runtime adaptation, given changes that carry the system outside the desirable or acceptable behavior. We also analyzed how the approach handles the two different adaptive behaviors described in Section 7.1.3. To do so, we implemented a prototype of the framework in Java™, and developed experiments for the autonomic management of web-based applications.

The prototype implementation consists of the overall framework and the website. The Apache web server (http://httpd.apache.org) running on Linux is used to execute requests. To monitor the execution context a monitoring tool was implemented in Python and integrated with the framework prototype. The monitoring tool can be configured in terms of the time interval between readings, among other options. These configuration options are defined in the framework configuration file. The tool monitors the CPU usage, the number of requests and queries, and the mean processing time for each request. This information is collected per request and then interpreted to give information per service.

To analyze how the policy drives changes in the service quality when the resource consumption varies, we generated several workloads to force different adaptations. In periods when the load is high, the system will adapt one or more components to provide a lower quality. In periods when the load is light and the service quality is not at its best, the system will adapt to provide a higher service quality. After adapting, the KPIs readings are temporarily ignored until the end of the stabilization period.

The experimental testbed consists of 4 machines. The application server machine runs the Apache Web Server and all the components that implement the business logic. The data server machine runs the recommendation and search engines. The remaining two machines run workload generators, functioning as clients. All machines are connected by a 100 Mbps Ethernet. The application machine is a 8 x 3.22 GHz Xeon processor with 8 GB RAM running Linux (kernel v2.6.24-21). We used Apache HTTP Server v2.2.8 configured with 150 *MaxClients* and a *KeepAliveTimeout* of 15 seconds, with CGI, SSL, and rewriting modules enabled. The application server machine is a 2.8 GHz Pentium IV processor with 2 GB RAM running Linux

(Kernel v2.6.20-17). The client machines are similar to the application server and they run Pylot (http://www.pylot.org), an open source tool for testing performance and scalability of web services based on an XML file that describes the workload. The tool also allows to control the number of clients and the interval between requests. We modified the original Pylot tool to run several workloads in sequence, each for a period of time.

The backend adaptable components are implemented as follows. The *Catalog* components are implemented using several HMTL pages containing text and images with different sizes (from 5 to 500 KB), each one with a lightweight and a regular version. The *User* components are implemented as several CGIs that generate the HTML pages on the fly, and perform queries to the application machine's engines to retrieve the necessary information. The replies consist of a number of recommended products or search result products. The generated pages include images and text. Finally, the *Account* components consist of dynamically generated pages requested over HTTPS (with text and media), where a lightweight and a regular version are available.

To execute the adaptations, the change of component fidelity is achieved using the rewrite module of Apache web server. This module allows the requested URL to be rewritten on the fly. It allows us to add a *fidelity*, *search* or *recommendation* argument to the url, to control the component execution. For instance, if the *BusinessCatalog* component is using the lightweight version, the module will add *low* as an argument to the url.

7.1.6 Results

To demonstrate the advantages and evaluate particular aspects of the proposed approach, the system was subject to two distinct overload scenarios. The first scenario employs a workload that causes CPU overload, while the workload in the second scenario causes an overload in terms of queries performed to the engines. The experiments allow us to illustrate the rule evaluation process and the flexibility and ease of using different policies, by comparing and quantifying the gains of adaptation. In the analysis of these gains, it is important to note that the needed impact varies from workload to workload. In some workloads, an adaptation with a less dramatic effect is sufficient, while in others a larger impact is needed. Similarly, the impact of an adaptation

may be far greater than necessary, as it may be the only available adaptation or the only one with large enough impact to satisfy a goal.

These experiments are followed by another two, that address performance and scalability. The first experiment evaluates how the approach scales in term of large numbers of adaptations and how that affects the time necessary to complete the planning activity. The second experiment evaluates the performance gains of separating the goal-oriented planning in an offline and online phases.

7.1.6.1 CPU Use Overload

This scenario allows us to illustrate how the system behavior changes in face of a significant increase in the number of requests made by clients, causing overload in terms of CPU use. It is also of interest to see if the system is able to return to the best service quality when the overload ceases. Components were initially deployed with a configuration that yields the best service quality: *Catalog* and *Account* web pages are served with regular quality, while *User* web pages have fresh recommendations and search results.

In this experiment, the system was subject to a workload that consists in four consecutive load steps: *light* (LW), *medium* (MW), *heavy* (HW), and *light* (LW). The *light* step allows all services to be offered with maximum service quality. The *medium* step requires the service quality to be lowered in order to respect the *cpu_use* threshold. The *heavy* step requires the system to operate with an even lower service quality. Finally, the light step brings the load back to the start, allowing the system to offer the best service quality again.

The three load steps include requests to all components. Table 7.3 presents the number of clients and the interval between requests to each component. The difference between load steps is the decrease in the interval time, thus, increasing the request frequency. Each load step consists of a collection of URLs that are requested by each client. These requests are submitted in a random order. Each client waits for a reply before sending another request. Our experiment used 90 clients running concurrently. The client ramp up takes 5 seconds. The clients start sending requests as soon as they start.

We measured the system performance and resource consumption *without* and *with* adap-

Workload	Clients	Catalog Int. (ms)	User Int. (ms)	Account Int. (ms)
Light	90	300	3500	3500
Medium	90	150	3500	3500
Heavy	90	150	3500	900

Table 7.3: Load steps used in the first experiment's workload

tation. The first case corresponds to an empty goal policy. The system never adapts and the load continuously increases, resulting in overload. In the second case, two analysis were made, one for each of the policies described in Section 7.1.3. Under the workload described above, the behavior of the system under policies A or B is relatively similar and, hence, we opt for presenting only the results obtained with policy A. This happens because the two policies only differ on the type of clients they favor (business/home) and since the change in the load imposed on the system in this experiment is very significant, in order to keep the CPU use below the threshold, the system is required to decrease the quality of service for *both* types of clients.

The results that are depicted in Figures 7.3, 7.4, and 7.5 compare the behavior of the non-adaptive and of the adaptive system in terms of CPU use and processing time in face of the same workload. The change of load step is marked by vertical dashed lines, thus, we can observe the load and the resource consumption increasing until stabilizing, or decreasing when returning to LW in the end. The adaptations (when they occur) are marked by vertical full lines. Straight horizontal lines mark the goal limit for a particular KPI.

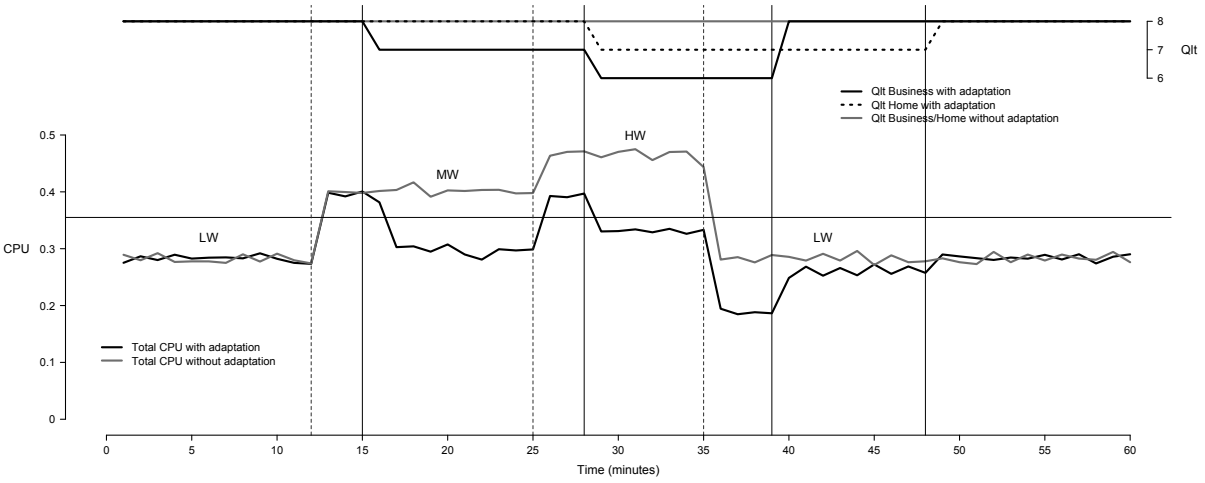


Figure 7.3: Comparison of CPU consumption with and without adaptation

Concerning CPU consumption, Figure 7.3 shows that the system can sustain significant load increases at the expense of degrading the service quality, when its behavior is adaptive. The figure illustrates that the increasing load results in higher CPU consumption. If the system is not adaptive, the load will push the CPU use to the limit. This happens during the HW, when the system is already consuming all the available CPU. On the other hand, if the system adapts, the CPU consumption can be maintained at a reasonable level, able to sustain any load peaks by degrading the service quality or offering the best service quality if the CPU consumption allows. The top of the plot shows the service quality evolution. The baseline is the service quality without adaptation, at its maximum value. The remaining lines are the service quality layered by business and home. The adaptations are triggered by violations of the *limit_cpu* goal.

The decision on how to adapt is made according to the current system state and strongly depends on the ranking of goals in the policy. The first adaptation (around minute 15) degrades the quality of business services *qlt_bsn*, and we can observe the CPU use slowly decreasing until it stabilizes below the limit. The second adaptation (around minute 28) degrades the fidelity of both business and home services, affecting the *qlt_bsn* and *qlt_hm*. These adaptations avoid system overload. The last two adaptations take place at the final workload, returning the system to its best quality, which is visible for *qlt_bsn* around minute 39, and for *qlt_hm* around minute 48. Note that policy B (not depicted in the figure) returns the system to the best quality using a different sequence of adaptations: since policy B favors home users, the effects on *qlt_hm* appear before the effects on *qlt_bsn*. For the reasons explained before, this is the only difference between the effects of the two policies for this concrete workload.

To analyze in more detail the impact of adaptation in terms of CPU savings, Figure 7.4 shows the CPU use layered by *Catalog* and *Account* components. The *User* components are not depicted because the CPU consumption of these components is the same during the entire experiment. The overall CPU use is also included, so that it is visible when the *cpu_use* goal is violated. We can observe that the CPU consumption of each component evolves through the sequence of load steps. The first adaptation changes the *BusinessCatalog* component to low fidelity, during the MW step. During the HW step, both *BusinessAccount* and *HomeCatalog* components are adapted to cut down on CPU consumption. When the load decreases, the *Busi-*

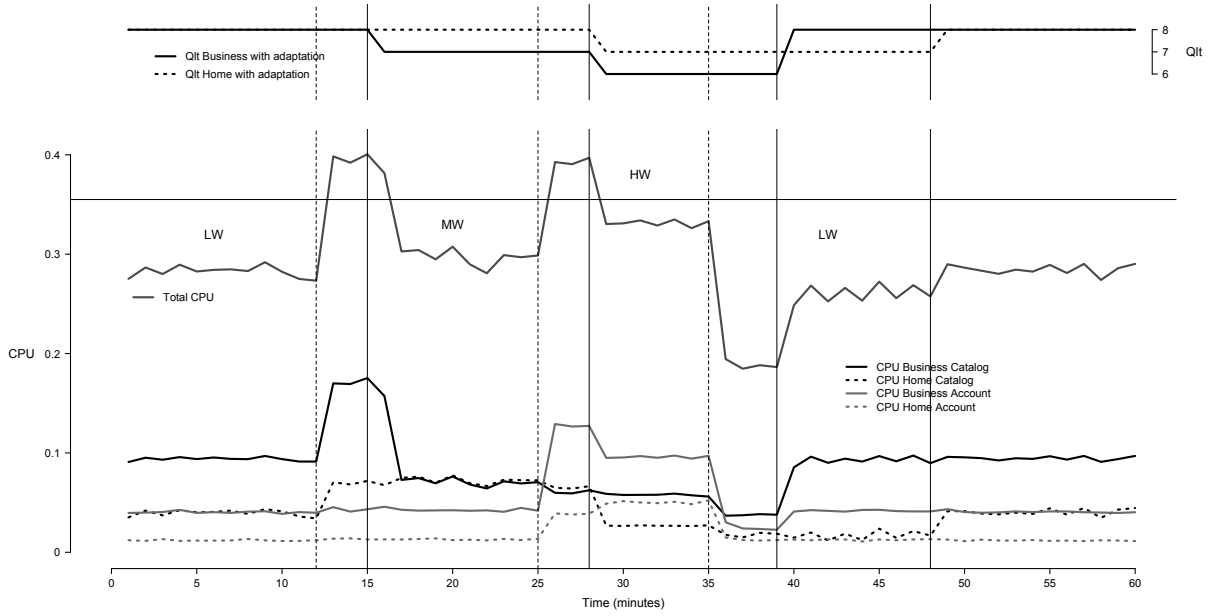


Figure 7.4: CPU consumption by component and overall

ness components are first returned to their best quality, followed by *HomeCatalog* component. Note that only *Catalog* and *Account* components can be adapted to cut down on CPU consumption: the contribution of *User* components is disregarded, since the majority of computations are performed at the application machine.

Figure 7.5 compares the processing times in both scenarios. The results show that by avoiding the overload with adaptation it is possible to maintain the baseline processing times. The processing times are layered by *business* and *home* services and the same applies to the service quality. If the system is not adaptive, there is a clear increase in the mean processing time when the load step changes from LW to MW, and from MW to HW. However, if the system is adaptive, when the system reacts to the violations of *limit_cpu* goal (addressed in Figure 7.3), it avoids resources exhaustion, maintaining the mean processing time. This is particularly visible around minute 28, where the *mrt_bsn* suffers an accentuated decrease. There are less perceptive decreases also in *mrt_bsn* around minute 15 and in *mrt_hm* around minute 28. Therefore, there is an improvement in terms of processing time at the expense of service quality. Avoiding the overload is not the only factor that favors better processing times: the decrease in the amount of content to be sent to clients also contributes to this goal. Sending less content not only allows faster sending times, but also frees more resources. When the load decreases, in the last LW

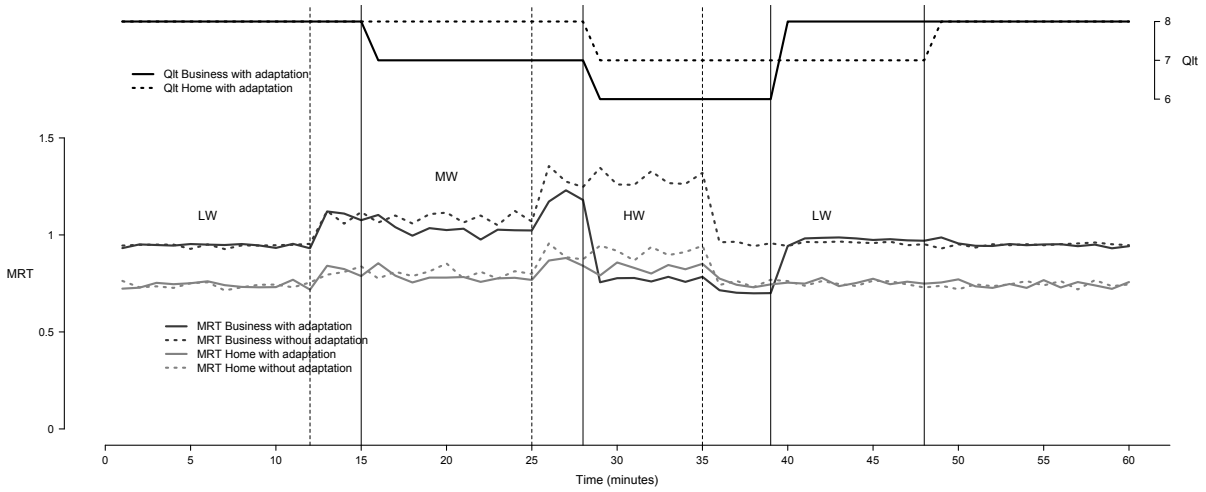


Figure 7.5: *Catalog* and *Account* mean processing time with and without adaptation

step, the value of the mean processing time per request is similar to the initial one, due to the adaptations that improve service quality again.

7.1.6.2 Query Rate Overload

We also provide results for a different workload scenario to illustrate how the system behavior changes in face of an increase in the number of requests handled by the *User* components, causing overload in terms of query load. Furthermore, the experiment in this scenario also illustrates how, under certain workloads, policies A and B cause the system to adapt in different manners. Finally, the experiment highlights the trade-offs involved when setting the evaluation period for optimization goals.

In this experiment, the system was subject to a workload that consists of three consecutive load steps: LW step, *Inter* step (IW) and MW step. The newly introduced *inter* step increases the load in terms of *User* requests, without violating the CPU use limit but making the engines *query_load* go beyond the established limit. As a result, service quality must be degraded to avoid overload. The IW step is characterized in Table 7.4. When compared to IW, the MW step increases the number of requests made to *Catalog* components, but decreases the number of requests made to the *User* components. This load step makes the system again degrade service quality to avoid overload, but, at the same time, also improves the service quality since the

number of queries to engines is no longer close to the limit.

Workload	Clients	Catalog Int. (ms)	User Int. (ms)	Account Int. (ms)
Inter	90	300	2600	3500

Table 7.4: Inter load step used in the second experiment's workload

Figures 7.6 and 7.7 present the individual processing times for all *home* and *business* services for both goal policies. They show the evolution of *query_load*, and *mrt_hm* and *mrt_bsn* KPIs, on a component basis, through out the workload. Again, the vertical dashed lines mark a change in the workload, while the full vertical lines mark when adaptation takes place. The horizontal line marks the *query_load* limit.

Figure 7.6 shows how the system performs under policy A. When the IW step starts there is an accentuated increase in the *query_load* that leads to the violation of the *limit_query_load* goal and to the system adaptation. The selected adaptation degrades the *qlt_hm*, showing the preference for business users. When the workload goes from IW to MW, the *limit_cpu* goal is violated and the system adapts as in the first experiment.

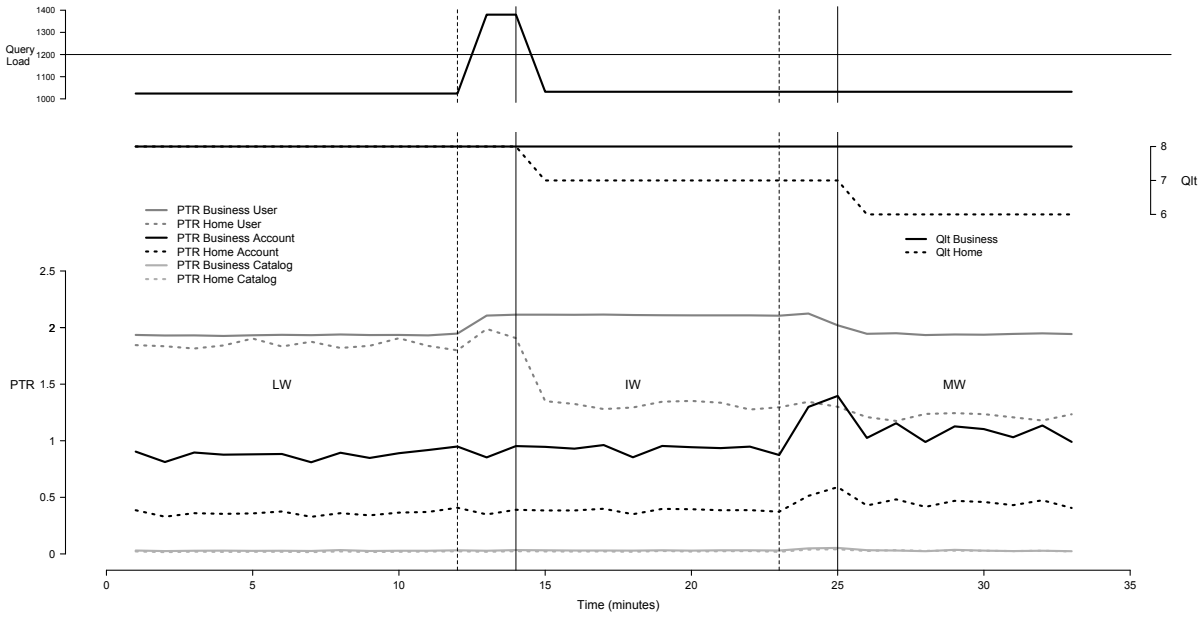


Figure 7.6: Processing time for requests under the first goal policy

Figure 7.7 shows the performance under policy B. When the *limit_query_load* goal is violated, the selected adaptation degrades the *qlt_bsn* instead of the *qlt_hm*, reflecting the preference for

home clients expressed in the policy. In the next workload, the *limit_cpu* goal is violated and the system adapts as with policy A.

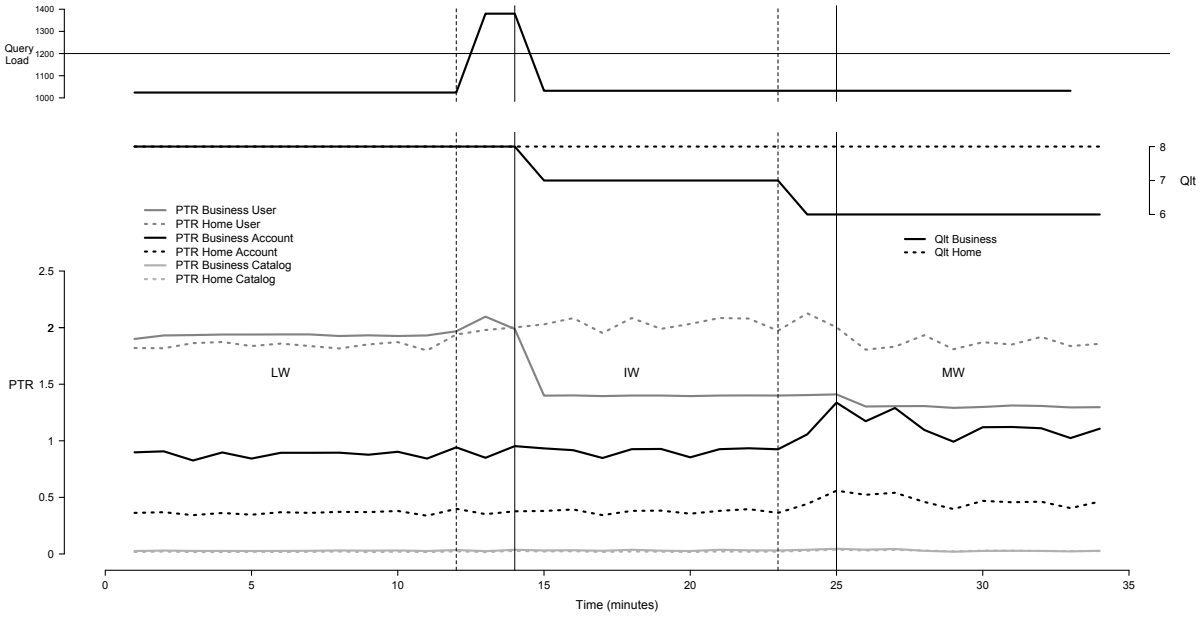


Figure 7.7: Processing time for requests under the new goal policy

Recall that when an exact goal is violated, to correct the system behavior faster, only the adaptations that affect the violated KPI are considered. Any adaptations tied to optimization goals will wait for the next evaluation period. We can observe this phenomenon in this experiment: when the workload changes from IW to MW, the system does not converge immediately to the optimal configuration; only the adaptations that help in reacting to the *limit_cpu* violation are immediately applied, other adaptations reacting to the lower query rate are only applied when the optimization goals are next evaluated (not depicted in the figure).

The evaluation of optimization goals is performed according to the defined time period. The situation just described also illustrates the trade-off involved in the specification of such time periods. The time period should be large enough to prevent the system from consuming an excessive amount of resources, but small enough to avoid delaying too long the necessary adaptations to achieve the optimal behavior. The definition of the appropriate value for this parameter is an important part of the policy specification. The approach favors the correction of exact goals. Thus, when the system enters a state that violates an exact goal, the framework attempts to correct it as fast as possible. The framework is less eager with respect to optimization

goals if the system is in a correct (but potentially sub-optimal) configuration. The larger the time period defined for the evaluation of optimization goals, the longer the system will take to reach the optimal behavior.

7.1.6.3 Scalability and Reaction Time

In this section, we report on the study conducted to evaluate the approach's performance and scalability using a larger number of adaptations. We extended the existing case study to allow another level in the fidelity of components *Catalog* and *Account*, now also accepting *high*. The number of adaptations increased to a total of 48, from the previous 32. The system was subject to the same workload, and despite some differences in the adaptations selected, the reaction times are depicted in Table 7.5 for the CPU overload scenario. As expected, the results show that there is an overall increase in the reaction times, a consequence of the larger number of adaptations and, consequently, adaptation sets to evaluate during the online phase. While, in the context of the case study this increase is not significative, these results alone do not allow to extract any conclusion about the scalability of the approach.

Event	**32 Adapt. Time (ms)**	**48 Adapt. Time (ms)**
Above_cpu_use (1st)	13.3	27.1
Above_cpu_use (2nd)	9.6	20.3
Increase_qlt_bsn	2.7	10.6
Increase_qlt_hm	2.9	11.4

Table 7.5: Reaction time during the *cpu overload* experiment for 32 and 48 adaptations

Therefore, to better analyze the scalability of the proposed approach in larger case studies, we opted to use several variations of the artificial case study (with adaptations and impacts randomly generated), instead of further extending the original case study. The variations are achieved by changing the number of available adaptations in the adaptation model. The use of an artificial case study offers two main advantages: to increase the adaptations specification size to virtually any size, while avoiding the effort and time necessary to devise the adaptations and their impacts. The policy for the artificial case study has 6 goals, 6 components, and 20 KPIs. The size of the adaptations specification is configurable, as well as the maximum number of impacts per adaptation. The adaptations and their impacts are generated randomly, according to the configuration. The number of adaptations ranges from 100 to 700. The number of impacts

Number of Adaptations:	**100**	**200**	**300**	**400**	**500**	**600**	**700**
Number of Sets	95	539	2699	7055	15679	25199	125999
Reaction Time (s)	0.141	0.327	0.961	1.483	2.885	4.545	23.58
RT/Sets (ms)	0.67	1.65	2.81	4.76	5.43	5.34	5.34

Table 7.6: Reaction times for different sizes of the adaptations specification

Number of Adaptations:	**100**	**200**	**300**	**400**	**500**	**600**	**700**
RT First (s)	0.029	0.130	0.229	1.433	2.186	4.347	7.916

Table 7.7: Reaction times using the *First* heuristic

goes up to 5 impacts, on different KPIs. To evaluate the reaction time, we artificially trigger an event with a corresponding state (where the KPI value of the violated goal is out of the threshold). The results obtained are depicted in Table 7.6 and are organized by the size of the adaptations specification, showing the number of evaluated sets and the corresponding *Reaction Time* (RT) for the algorithm.

As expected, the results show that as the number of adaptations increases, so does the reaction time to find the optimal solution. In fact, as depicted in Table 7.6, this time grows exponentially due to the increase in the number of sets of adaptations. However, in time constrained applications, one can often trade optimality for lower reaction times. In particular, when some exact goal is violated, what is important it is to find quickly a set of adaptations that corrects the problem. This can be achieved using a simple heuristic, as shown in Table 7.7. The *First* heuristic finds the first solution that allows to satisfy the goal corresponding to the triggered event. This solution may not be the optimal solution, as the goals ranked below the goal being addressed are not considered, or a better solution may be further down the list of combinations. This heuristic allows to substantially reduce the reaction times in the experiment (of course, in the worst case scenario, it will take as long as the regular algorithm), at the expense of optimality. Although the research for more sophisticated heuristics is out of the scope of this work, the results show that is possible to quickly find a set of adaptations that is able to address the state deviation identified, and search later for the optimal solution. For the latter purpose, our approach has the advantage of being automated and to benefit from the offline pre-processing phase. Furthermore, the reaction times obtained for the different cases are substantially lower than any human reaction time.

7.1.6.4 Policy Evaluation and Reaction Time

In the goal-oriented planning, we advocate the use of a two-phase approach to evaluate the goal policies. The offline phase identifies which adaptations correct each kind of violation of the policy goals. This selection relies on the static analysis of the impact functions and can become quite complex if we do not limit the classes of impact functions that can be used (e.g., to linear functions). The calculation of the viable combinations of adaptations, which can also be statically determined, is also performed in this phase. During the online phase, it is only necessary to get the actual system state and calculate the impact of each of the adaptations previously selected.

Alternatively, the policy evaluation could be carried out exclusively at runtime by calculating the impact of every adaptation that affects the KPI of the goal which is tied to the triggered event. The calculation of the viable combinations of adaptations that contribute to solve the violation, would be calculated afterwards. Given that this process is simpler, it is important to evaluate the performance gains achieved with the two-phase approach. In this section, we evaluate these gains in the context of the case study (the original with 32 adaptations) and of the artificial case study introduced in the previous section.

We considered the events triggered during the workload sequences described in previous sections and measured the time necessary to decide how the system should adapt in each case, using the two-phase and the single-phase approaches. The results are presented in Table 7.8 for the *cpu overload* and *query rate overload* experiments. As it can be seen, for all events, gains can be obtained when using the two-phase approach. A large fraction of the work is performed offline, improving the processing time of the system. The results also show that the gains obtained with the two-phase approach increase as the number of adaptations that have impact on the same KPI also increases. In the example, the fact that the number of adaptations that can be used to decrease *cpu_use* is larger than the number of adaptations available for the other events is reflected in the larger difference between reaction times observed for the first instance of *Above_cpu_use* event. This difference is much smaller in the second instance of this event. This happens because, when the second instance of *Above_cpu_use* event is triggered, the system has already been subject to adaptation and, as a result, some adaptations available for

dealing with *Above_cpu_use* event are not applicable anymore.

CPU Overload Events	**2-phase RT (ms)**	**Single-phase RT (ms)**
Above_cpu_use (1st)	13.3	71.5
Above_cpu_use (2nd)	9.6	18.1
Increase_qlt_bsn	2.7	10.7
Increase_qlt_hm	2.9	10.9
Query Rate Overload Events	**2-phase RT (ms)**	**Single-phase RT (ms)**
Above_query_load	12.5	29.2
Above_cpu_use	13.1	32.7

Table 7.8: Reaction times per event, during both experiments

It is also interesting to compare both approaches in terms of scalability. Table 7.9 depicts the reaction time of the single-phase approach for different variations of the artificial case study. When comparing the results obtained from both approaches (see Table 7.6 for the two-phase approach), they show that the two-phase approach takes less time to find the best set, despite analyzing a larger number of sets. This outcome is due to the offline generation of viable combinations of adaptations, which allows to reduce the reaction time.

Number of Adaptations:	**100**	**200**	**300**	**400**	**500**	**600**	**700**
Number of Sets	47	399	2591	3455	6911	20647	38879
Reaction Time (s)	0.113	0.394	1.911	2.826	21.981	719.541	1485.753

Table 7.9: Single-phase reaction times for different sizes of the adaptation specification

7.2 Distributed Components

In this section, we discuss the evaluation of the approach using the case study introduced in Section 3.1 with focus on the distributed components. These components are an in-memory distributed data grid and the underlying communication support. In-memory distributed data grids (IMDDGs) supply applications with a scalable storage repository where data can be accessed without bottlenecks and shared across a pool of virtual servers. The IMDDG used is *Infinispan* and the communication service is *JGroups*. Both have a large potential for adaptation. We focus on adaptations that are related with the distribution, such as scaling up and down the number of instances and adapting global parameters in distributed components.

7.2.1 Revisiting the Case Study

As previously mentioned, the case study used in this experiment focus on distribution, addressing the adaptation of only a subset of the components, as depicted in Figure 7.8. We summarize the adaptable components next. The system relies on *Infinispan* to speed up web servers' access to data, with a local instance serving each of system web servers. The local instance has its own cache that serves as a proxy to query and update remote caches of the platform. The website content is generated from data that is available in the cache of *Infinispan*. If the data is not available in cache (local or remote), it will be requested to the non-distributed components (*Catalog* and *User* components) and added to cache. *Infinispan* does not cache sensitive data, thus any requests that involve such data are made directly to the *Account* components. *JGroups* provides communication and coordination support between *Infinispan* instances. The management of the database is not considered in this case study.

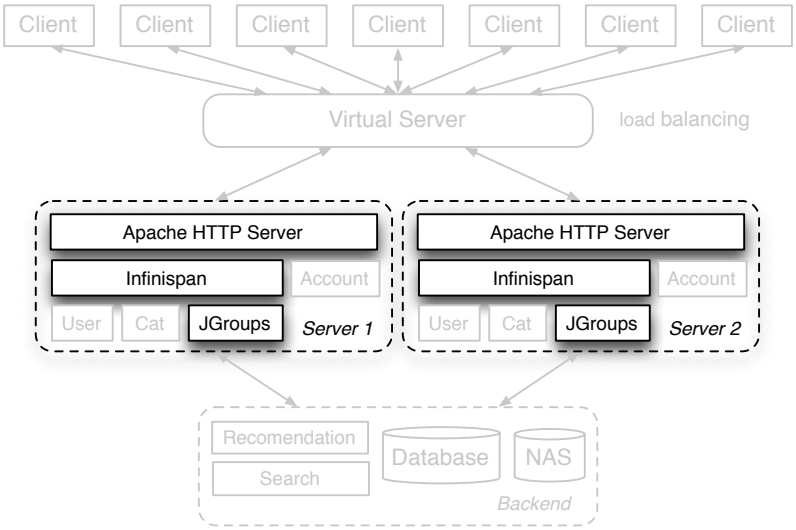


Figure 7.8: Case study: focus on distribution

The case study's self-management aims at taking advantage of the reconfigurability of the *Infinispan* and *JGroups*. The goal is to optimize performance in order to maintain user satisfaction, but at the same time to minimize resource consumption to cut on costs. Furthermore, and as mentioned before, we are also concerned with fault-tolerance and healing properties. The development and evaluation of the case study will focus on these goals.

7.2.2 Knowledge Model

In this section, we address the models that composed the knowledge model. The knowledge model introduced in Section 7.1.1 addresses the non-distributed components only. The knowledge model presented in this section complements that model, but only describes the distributed components *Infinispan* and *JGroups*. We also consider the *WebServer* component for comprehension purposes. We start with the architecture model, followed by the sensor and context models, the adaptation model, and we conclude with the effector and executor models.

7.2.2.1 Architecture Model

The case study is composed of *WebServer*, *Infinispan*, and *JGroups* components, running in nodes of type *ServerNode*. Next, we describe the component and nodes types employed in the case study, together with the system and the initial configuration. Their description is presented in Listing 7.11.

Infinispan is a distributed component that caches the *Catalog* and *User* data. *Infinispan* is prepared to support different clustering modes: *local*, *replicated*, *invalidation*, *distribution*, *L1 caching*. In this case study, the component is running in *replicated* mode all the time. This means that all the local caches have the same content, which allows speedy access to data in any instance, and sharing of cached content. The component has three different adaptable parameters. The *nb_threads* and the *multicast_properties* were already covered in Section 4.1. The global *communication* parameter determines which communication service is used. *Infinispan* uses *JGroups* as the default communication service, but there are other communication services such as *RAppia*. The *JGroups* component is also distributed and provides group communication services. It has an adaptable parameter *multicast_properties*, which control what type of order is used: *causal* or *total*.

Listing 7.11: Case study (D): architecture model

```
Distributed Component Infinispan
  Parameters
    Global communication:{jgroups,rappia}
    Global multicast_properties:{causal,total}
    nb_threads: integer
```

```
Distributed Component JGroups
    Parameters
        Global multicast_properties:{causal,total}

Node ServerNode
    Parameters
        is_active: boolean
    Components
        WebServer
        Infinispan
        JGroups

System
    Parameters
        max_nodes = 10
    Nodes
        ServerNode
```

The system initial configuration is presented in Listing 7.12. The initial set up consists of three instances of *ServerNode*, where *Infinispan* and *JGroups* instances are deployed and prepared to used causal order.

Listing 7.12: Initial system configuration

```
Configuration
    System
        ServerNode = 3
    Nodes
        ServerNode:{is_active=true}:{WebServer,Infinispan,JGroups}
    Components
        Infinispan:{communication=jgroups,multicast_properties=causal,nb_threads=2}
        JGroups:{multicast_properties=causal}
```

7.2.2.2 Sensor and Context Model

The user satisfaction is tied to the waiting time, which directly depends on the performance of *Infinispan*. Therefore, managing the operation of *Infinispan* is key to the system. In *Infinispan* there are two main aspects of interest to assess its performance: the service ratio and the latency of processing requests. The service ratio *(SR)* gives information on the requests processing rate. It it obtained from the rate of incoming (RI) (read and write) requests performed to *Infinispan*, and the rate at which these requests are served (RS). Thus, the service ratio is defined as $SR = RS/RI$. Ideally, the service ratio should be 1, otherwise, requests start to be queued and,

eventually, the application needs to be blocked or requests need to be dropped. The service latency (AL) is the average time needed to process the *read* and *write* requests to the *Infinispan* cache. Typically, writes are slower than reads given that multiple copies of the data may need to be updated and/or invalidated in response to a write request. AL is either low (1) or high (2), as it abstracts the high or low contribution of *Infinispan* for the system average latency.

The other concerns of this case study are the availability and healing properties. The availability depends on the abort ratio, while the healing properties through the number of active nodes. The abort ratio (AR) is the aborted requests per second and the global value is the average of the abort ratios observed in all instances of *Infinispan*. *Infinispan* is heavily optimized for read requests. A read does not require to acquire an explicit lock for the cache entries, instead directly read the entry in question. A write request, on the other hand, needs to acquire a write lock. This ensures only one concurrent write per entry, causing concurrent writes to queue up to change an entry. The write requests that cannot obtain the lock on all instances abort after a timeout. For instance, let us assume a key K, that is hashed to nodes A, B, and transactions $TX1$ and $TX2$ that have to acquire the lock for K. If $TX1$ starts in A and $TX2$ starts in B simultaneously, $TX1$ will acquire the lock for K in A first, while $TX2$ will acquire it in B. When they try to acquire the lock on the remote node that also has K, both transactions will be waiting for one another. They will fail after the timeout period elapses. The number of active servers (*active_nodes*) describes how many instances of *ServerNode* are up and running.

While not a mandatory requirement in this case study, saving resources when they are not necessary allows to cut down on costs. The power consumed (PC) by the system is a metric that captures how many resources are used to maintain the cache operational. It is important that *Infinispan* is able to scale elastically with demand to save resources.

The metrics introduced in the previous paragraphs are described in the context model, using information from the sensor model. The sensor model is described in Listing 7.13 and the context model in Listing 7.14. These metrics are used to describe seven KPIs, summarized in Table 7.10.

Listing 7.13: Case study (D): sensor model

```
Sensor
   Target: Infinispan
   Observable integer load(): RMargin 0 periodically: 60
   Observable integer throughput(): RMargin 0 periodically: 60
   Observable integer latency(): RMargin 5 periodically: 60
   Observable double abort_ratio(): RMargin 0.00000001 periodically: 60
   Observable Global double writePercentage(): RMargin 0.01 periodically: 60

Sensor
   Target: ServerNode
   Observable integer power_consumption(): RMargin 1 periodically: 60

Sensor
   Target: system
   Observable integer nodes(): RMargin 0
   Event node_failed with failed_node: ServerNode
```

Name	Description
active_nodes	number of active *ServerNodes*
AL	Abstraction of average latency
AR	Ratio of aborted requests
RS	Requests throughput
RI	Requests load
PC	Power consumption
SR	Ratio of requests processed (throughput÷load)

Table 7.10: KPIs and CKPIs used in the case study

7.2.2.3 Adaptation Model

In the case study, we focus on two settings to configure the platform: the replication degree and the replica update protocol. Regarding the first, while *Infinispan* can handle changes in the number of local instances running, it does not offer any support to adapt it during runtime. For instance, it does not adapt the number of servers to address changes in the workload. As already mentioned, by adding a new server/replica we are able to process more incoming requests (improve the service ratio), but at the same time, write requests will take longer to be performed and cost increases. In terms of replica update protocol, *Infinispan* uses a fixed configuration defined at deployment time. The adaptation of the configuration may yield improvements in performance, namely, through the activation of the total order guarantee for the communication. This allows to reduce the abort ratio at the expense of increasing the request latency. There

Listing 7.14: Case study (D): context model

```
Composite Sensor
   Target: system
   Observable
   integer writeTime(): RMargin 0
   periodically:60
   The writeTime() is given as described in the work by Didona et al [DRPQ12]

Composite Sensor
   Target: system
   KPI System-wide Observable
      integer active_nodes(): RMargin 0
      periodically:60
      from nodes for system with AF: Sum CF: Sum
   KPI System-wide Observable
      integer RI(): RMargin 10
      periodically:60
      from load for Infinispan with AF: Sum CF: Sum
   KPI System-wide Observable
      integer RS(): RMargin 10
      periodically:60
      from throughput for Infinispan with AF: Sum CF: Sum
   KPI System-wide Observable
      double AL(): RMargin 0.05
      periodically:60
      from latency for Infinispan with AF: Sum CF: Avg
   KPI System-wide Observable
      double AR(): RMargin 0
      periodically:60
      from abort_ratio for Infinispan with AF: Avg CF: Avg
   KPI System-wide Observable
      double PC(): RMargin 10
      periodically:60
      from power_consumption for ServerNode with AF: Sum CF: Sum

Composite Sensor
   Target: system
   KPI Composed-System-wide Observable
      double SR(): RMargin 0.01
      with JF: RS÷RI
```

Listing 7.15: Case study (D): adaptations

```
Adaptation activateTotalOrderInf
Adaptation activateCausalOrderInf
Adaptation activateTotalOrderJG
Adaptation activateCausalOrderJG
Adaptation removeNode
Adaptation addNode
Adaptation deployComponents
Adaptation activateNode
```

are eight possible adaptations, as presented in Listing 7.15. From the adaptations possible, we discuss in detail the adaptations shown in Listing 7.16.

The first adaptation activates the total order guarantee in all instances of the *Infinispan* component, increasing its contribution to the service latency, decreasing the abort ratio, and with impact on throughput. The second adaptation allows the same, but in the *JGroups* component. The third adaptation adds a new node, and the fourth adds the components. This is achieved by adding all the necessary components to the node: the *WebServer*, *Infinispan*, and *JGroups*. As a result, the number of *active_nodes* increases and has impact on *RS*. The impact on *RS* is specified using the average *write time*, computed by the context analyzer as described in [DRPQ12]. While this adaptation may affect the latency, the impact is negligible (compared with the first adaptation), thus, we do not consider it. The fifth adaptation activates the new node.

The impact functions of the available adaptations in *RS* are based on results obtained from distinct benchmarks made to the system, where different combinations of the write percentage, the key pool size, and the number of servers were explored. The impact functions used are not exact, but they do provide enough accuracy for the purpose in hand.

The dependencies and conflicts described in the adaptation model are presented in Listing 7.17. The explicit conflicts are two pairs which describe adding a new node and deactivating a old one, and vice-versa. The dependencies refer to the addition of nodes and to activating the total order in both distributed component, and vice-versa.

Listing 7.16: Case study (D): excerpt of adaptation model

```
Adaptation activateTotalOrderInf:
   Target:
      Infinispan
   Requires:
      target.multicast_properties == causal
   Impacts:
      target.parameterChanged(multicast_properties, total)
      AL += 1
      AR ÷= 3.33
      RS *= (11 − writePercentage ∗ log(1.2 ∗ active_nodes))
   Stabilization:
      period = 60 secs

Adaptation activateTotalOrderJG:
   Target:
      JGroups
   Requires:
      target.multicast_properties == causal
   Impacts:
      target.parameterChanged (multicast_properties, total)
   Stabilization:
      60

Adaptation addNode:
   Target:
      system
   Requires:
      active_nodes() < target.max_nodes
   Impacts:
      target.nodeAdded(ServerNode)
   Stabilization:
      60

Adaptation deployComponents:
   Target:
      ServerNode
   Impacts:
      target.componentAdded(WebServer)
      target.componentAdded(Infinispan)
      target.componentAdded(JGroups)
   Stabilization:
      60

Adaptation activateNode:
   Target:
      ServerNode
   Requires:
      target.is_active == false
   Impacts:
      target.parameterChanged(is_active, true)
      sensorSystem().active_nodes() += 1
      RS = (writePercentage ∗ (1 − AR) ∗ writeTime)^−1
   Stabilization:
      60
```

Listing 7.17: Case study (D): conflicts and dependencies

```
Conflicts
   Adaptations (deactivateNode,addNode)
   Adaptations (activateNode,removeNode)
   Adaptations (activateTotalOrderJG,activateCausalOrderInf)
   Adaptations (activateCausalOrderJG,activateInfinispanOrderInf)

Dependencies
   If addNode Apply
      deployComponents with Target target.NodeAdded(ServerNode)
      activateNode with Target target.NodeAdded(ServerNode)
   If activateTotalOrderInf Apply
      activateTotalOrderJG
   If activateCausalOrderInf Apply
      activateCausalOrderJG
```

7.2.2.4 Effector and Executor Models

The *Infinispan* and *JGroups* components are targets of adaptations, as well as the system, with the addition and removal of *ServerNodes*. The effector model describes the effectors for the system, *ServerNode* nodes, *JGroups* and *Infinispan*, which are presented in Listing 7.18. The component effectors are similar, catering to pre-adaptation and post-adaptation needs, and allowing to set the parameters. The *ServerNode* effector accepts commands to place the entire node in quiescence and add/remove all the components. The system effector is prepared to add and remove nodes of type *ServerNode*.

The adaptations described in the adaptation model are not executed in the same manner, therefore, there are different strategies. Listing 7.19 shows the strategy description for three different adaptations. The *deployComponents* adaptation is executed using the flash strategy. The adaptations *activateTotalOrderJG* and *activateCausalOrderInf* are executed using a different strategy, the stop-n-go. Listing 7.11 shows default the strategies for each adaptation and if they are parallel. In terms of parallel strategies, they are shown in Listing 7.20.

7.2.3 Goal Policy

The self-management is guided by the following objectives. One is to optimize performance, balancing the different trade-offs in terms of service ratio, service latency, and abort ratio. The

Listing 7.18: Case study (D): effector model

```
Effector
    Target:
        Infinispan
    Commands
        makeQuiescent()
        stateType getState()
        putState (state)
        setParameter (communication,{jgroups,rappia})
        setParameter (multicast_properties,{causal,total})
        setParameter (nb_threads,integer)
        start()
        stop()
        pause()
        resume()

Effector
    Target:
        JGroups
    Commands
        makeQuiescent()
        stateType getState()
        putState (state)
        setParameter (multicast_properties,{causal,total})
        start()
        stop()
        pause()
        resume()

Effector
    Target:
        ServerNode
    Commands
        makeQuiescent()
        addComponent(WebServer)
        removeComponent(WebServer)
        addComponent(Infinispan)
        removeComponent(Infinispan)
        addComponent(JGroups)
        removeComponent(JGroups)

Effector
    Target:
        System
    Commands
        addNode (ServerNode)
        removeNode (ServerNode)
```

Listing 7.19: Case study (D): excerpt of the executor model

```
default deployComponents Strategy flash Parallel
   Step:
      effectorsInstance(target).addComponent(WebServer)
      effectorsInstance(target).addComponent(Infinispan)
      effectorsInstance(target).addComponent(JGroups)
      forall e: effectorsDeployedInNode(target)
         e.start()

default activateTotalOrderJG Strategy stopAndGo
   Step:
      forall e: effectorsComponentType(JGroups)
         e.pause()

   Step:
      forall e: effectorsComponentType(JGroups)
         e.setParameter(multicast_properties,total)

   Step:
      forall e: effectorsComponentType(JGroups)
         e.resume()

default activateTotalOrderInf Strategy stopAndGo
   Step:
      forall e: effectorsComponentType(Infinispan)
         e.pause()

   Step:
      forall e: effectorsComponentType(Infinispan)
         e.setParameter(multicast_properties,total)

   Step:
      forall e: effectorsComponentType(Infinispan)
         e.resume()
```

Listing 7.20: Case study (D): parallel strategies

```
Parallel
 activateTotalOrderInf:Stop-n-go AND activateTotalOrderJG:Stop-n-go
 activateCausalOrderInf:Stop-n-go AND activateCausalOrderJG:Stop-n-go
```

Adaptation	Strategy	Parallel
activateTotalOrderInf	stop-n-go	no
activateCausalOrderInf	stop-n-go	no
activateTotalOrderJG	stop-n-go	no
activateCausalOrderJG	stop-n-go	no
removeNode	flash	yes
addNode	flash	yes
deployComponents	flash	yes
activateNode	flash	yes

Table 7.11: Adaptations and corresponding strategies

aim is to have the highest possible service ratio, and the lowest service latency and abort ratio, guaranteeing that user satisfaction is the best possible. Another objective is to be energy-efficient, namely, by reducing the power consumption required to maintain the system, thus, cutting on costs. Another objective is fault-tolerance, providing self-healing and self-protection features to the system, so that it can tolerate server failures without loss of data.

Considering these objectives, the goal policy chosen for the case study is one among several, as it is possible to derive different policies that give distinct priorities to different objectives. The policy consists in five goals. The first is to maintain the redundancy, i.e., a minimum number of servers/replicas. This self-healing property is the most important goal because it will allow the system to recover from fail overs and avoid downgrading the service to a critical level. The next three goals address performance and user satisfaction issues. The second goal limits the abort ratio because, when it is higher than the 0.008 threshold, the system becomes irresponsive, a consequence of being blocked most of the time (the threshold was determined experimentally, through benchmarks). In the third goal, the system attempts to process as many requests as possible, to maximize the service ratio. The fourth goal is to minimize the latency. Finally, the last goal minimizes the resource consumption, if other goals are not violated, to cut on costs.

```
Goal maintain_redundancy: active_nodes Above 3
Goal limit_abort_ratio: AR Below 0.008
Goal maximize_SR: Maximize SR MinGain 0.05 Every 300
Goal min_latency: Minimize AL MinGain 0.05 Every 400
Goal min_cost_resources: Minimize active_nodes Every 500
```

7.2.4 Open Action Policy

With the information described before, the planning is now able to generate the rules that will be used to manage the system. The extracted events are described in Table 7.12, one per goal. Before generating the adaptation rules, the specified adaptations need to be unfold. Since the maximum number of nodes is 10 and only one concrete node type is accepted by the system, the system may have nodes ranging from $\{ServerNode1, ..., ServerNode10\}$. Table 7.13 presents the adaptations that need unfolding and how many variants result from the process. The first four adaptations do not need unfolding because they target global parameters. The remaining adaptations need to be unfolded for every possible node, thus, each will have 10 variants.

Type	**Goal**	**Event**	**Trigger**
Exact	*maintain_redundancy*	Below_maintain_redundancy	active_nodes < 3
Exact	*limit_abort_ratio*	Above_limit_abort_ratio	$AR > (0.008 + 0.0001)$
Optimiz	*maximize_SR*	Increase_maximize_SR	Every 300 s
Optimiz	*min_latency*	Decrease_minimize_latency	Every 400 s
Optimiz	*min_cost_resources*	Decrease_min_cost_resources	Every 500 s

Table 7.12: Extracted events

Adaptation	**Unfolding**	**#Variants**
activateTotalOrderInf	no	no
activateCausalOrderInf	no	no
activateTotalOrderJG	no	no
activateCausalOrderJG	no	no
removeNode	flash	10
addNode	flash	10
deployComponents	flash	10
activateNode	flash	10

Table 7.13: Unfolding of adaptations and number of variants

The open action policy is presented in Listing 7.21. The list of viable combinations presented for each rule is not exhaustive.

7.2.5 Results

To illustrate the operation of our autonomic manager and the effect of the high-level goal policy, a deployment of the system was subject to a variable load. All experiments follow the

Listing 7.21: Case study (D): open action policy

```
When below_maintain_redundancy
   Do
      Select { [addNode(ServerNode1),deployComponents(ServerNode1),activateNode(
          ServerNode1)], ..., [addNode(ServerNode10),activateNode(ServerNode10),
          deployComponents(ServerNode10)], ... }

When above_limit_abort_ratio
   Do
      Select { [activateTotalOrderInf,activateTotalOrderJG] }

When increase_maximize_SR
   Do
      Select { [activateTotalOrderInf,activateTotalOrderJG],
        [addNode(ServerNode1),deployComponents(ServerNode1),activateNode(
            ServerNode1)], ... }

When decrease_minimize_latency
   Do
      Select { [activateCausalOrderInf,activateCausalOrderJG],
        [addNode(ServerNode1),deployComponents(ServerNode1),activateNode(
            ServerNode1)], ... }

When decrease_min_cost_resources
   Do
      Select { [removeNode(ServerNode1)], ... , [removeNode(ServerNode10)], ... }
```

same pattern: we first let the system stabilize in the best configuration for a given workload, then we change the workload characterization and observe how the system reacts.

7.2.5.1 Implementation and Experimental Setup

We used *Infinispan* version 5.0.0, extended with a number of sensors to export monitoring data and effectors to support the runtime change of the number of instances. *Infinispan* was deployed in full replication (i.e., each data item is replicated in every active instance) and, for the remaining parameters, with the default configuration included in the official distribution. Furthermore, we have used *JGroups* 2.11.0, which has been also augmented with an effector that is able to activate or deactivate the total order layer of the *JGroups* group communication stack. As before, we have deployed *JGroups* using the default configuration; for total order we use *JGroups* sequencer protocol. The load imposed by the web servers is emulated by Radargun benchmark [Rad], version 1.1.0. The benchmark simulates the clients, the virtual server load balancer, and the web servers at each node. The benchmark detects when a new server is added to the cluster, through a monitoring agent present in each node. This agent notifies the analyzer when an instance is locally created at the node, which in turn notifies the virtual server's load balancer. Radargun emulates the operation of the web server, by performing requests to *Infinispan*: it is possible to configure the load profile by setting parameters of the benchmark such as the read/write ratio, the size of the pool of objects to be requested, among other options. We have also performed a number of incremental improvements to the benchmark that allow us to have a finer control of the workload, including the definition of the *RI* and the length (in time) of each experiment. All software components of our deployment have been implemented in the Java™programming language. The external autonomic controller (analyzer, generator, interpreter, and executor components) is executed in a dedicated machine. The experimental testbed consists of eleven machines. One hosts the autonomic controller and the remaining machines can run an instance of the *Infinispan*/*JGroups* (up to 10 instances total). Each machine is a Dual Intel Xeon Quad-Core, 2.13 GHz clock speed, and 8 GB of RAM running Linux (kernel 2.6.32-21-server). All machines are connected by a 1 Gbps Ethernet.

Changes to the workload are made such that different adaptations are more appropriate

Workload	Object Pool	% of Writes	RI
HC-3	100	90	200
LC-3	5000	40	30
LC-5	5000	40	2400
LC-6	5000	40	2750

Table 7.14: Workloads

in each experiment. The workload transitions are based on the four workloads described in Table 7.14. The workloads are differentiated by two main characteristics: *high contention* (HC) or *low contention* (LC), and *load* (measured by the rate of incoming requests *RI*). The high contention workload captures a scenario where concurrent accesses to the same item occur often, which creates many opportunities for deadlocks and a potential increase in the abort ratio. This is achieved using a write-dominated sequence of requests accessing a small pool of objects. The low contention workloads capture scenarios where conflicts are infrequent and the potential number of aborts due to contention is small. This is achieved using a more balanced read-write ratio and a much larger object pool. The difference between the three LC scenarios is the total load, where 3, 5 and 6 state the number of servers that are necessary to keep the *SR* near to one.

Using this set of workloads, we have experimented 4 different transitions. Transitions LC-3 to HC-3; ii) HC-3 to LC-3; iii) LC-5 to LC-6; and iv) LC-6 to LC-5. Each of these transitions is discussed in detail next.

7.2.5.2 High and Low Contention

Transition LC-3 to HC-3 In the first scenario, the workload changes from low contention to high contention. When operating with low contention, *JGroups* is running without total order, as it allows to obtain a lower latency. When the workload changes, there is a significant increase in the abort ratio, which also worsens the service ratio, as fewer requests are served with success. The abort ratio and service ratio after the workload change and before the system reconfiguration are depicted in Figure 7.9a on the first part of the plot (until the vertical line that marks the adaptation). As the graphic shows, the abort ratio violates the *limit_abort_ratio* goal, because it is above the threshold specified in the policy.

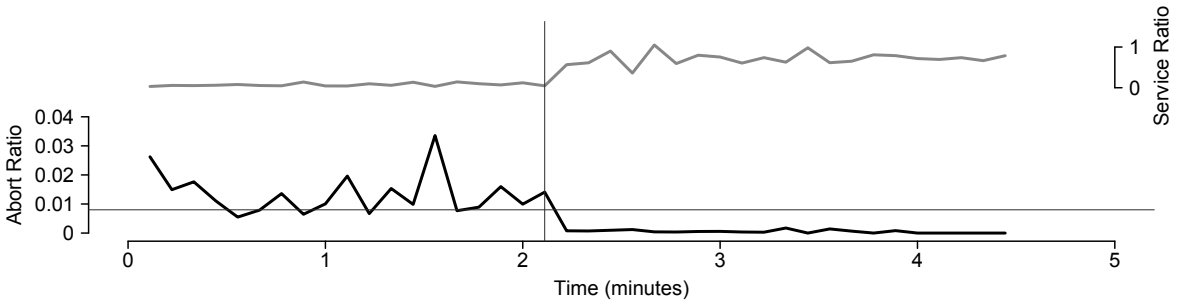
When the context analyzer triggers an *Above_limit_abort_ratio* event, the interpreter evalu-

ates the corresponding rule, using the current system state carried by the event. The selected adaptation is to *activateTotalOrder* guarantee. As a result, the abort ratio decreases and, consequently, the service ratio increases, which can also be observed in Figure 7.9a, after the adaptation, when the system stabilizes.

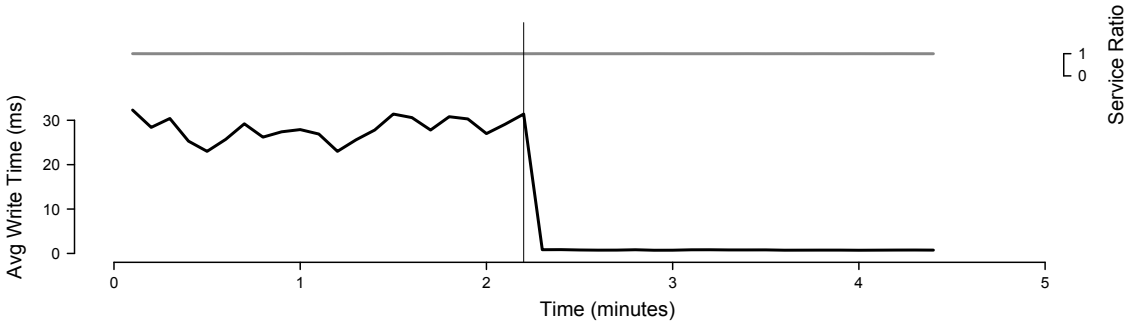
Transition HC-3 to LC-3 The second scenario illustrates the inverse adaptation. The system is initially under HC-3, which demands total order. Then the workload is changed to a low contention profile. This results in a decrease of the contention level and of the abort ratio. In this scenario, no exact goal is violated, but approximation goals still play a role in improving the system performance. This happens when the event *Decrease_min_latency* is triggered. The interpreter determines that it is possible to improve the latency without compromising any of the higher ranked goals, namely the service ratio or the abort ratio. The selected adaptations are *DeactivateTotalOrder* for *Infinispan* and *JGroups*. Figure 7.9b shows that not only the service ratio is not degraded by the adaptation, but the write latency is reduced. This effect is noticeable by observing the average write time (available as context information).

7.2.5.3 Variable Load

Transition LC-5 to LC-6 In the third scenario, the system is operating under a low contention scenario, such that total order is not required, and the load requires 5 instances to remain active to avoid the *SR* dropping below 1. Then, the workload is changed by increasing the rate of incoming requests, such that the 5 instances become overloaded. As a result, the service ratio decreases. When the event *Increase_maximize_SR* is triggered, the interpreter will select the *AddNode* adaptation (and the dependent adaptations) to increase the system capacity, thus, increasing the rate of served requests and, consequently, the service ratio. Figure 7.10a depicts the service ratio before and after the adaptation, which shows a clear improvement after the adaptation, returning the service ratio to 1. However, this demands more power consumption, since the newly active server is no longer idle, as Figure 7.10a also depicts. We opted to show the average power consumption because power consumption is not steady over time.



(a) First scenario: activating the total order guarantee



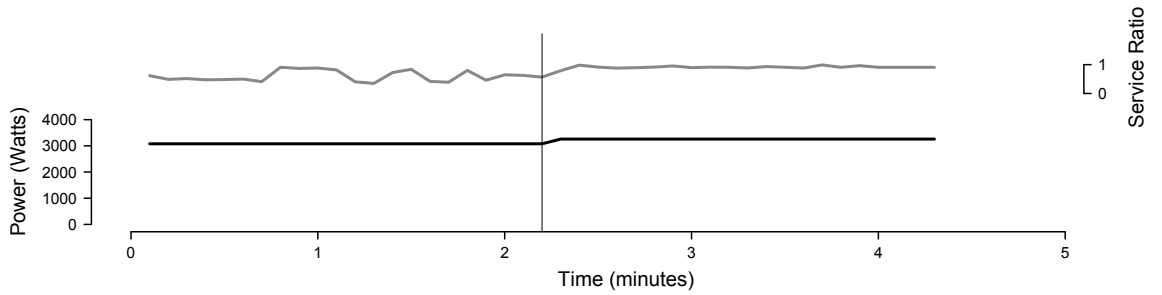
(b) Second scenario: deactivating the total order guarantee

Figure 7.9: Experimental results some minutes before and after the adaptation

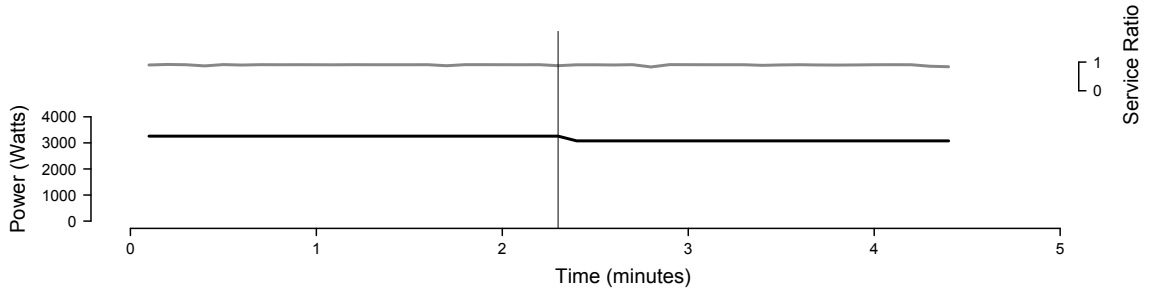
Transition LC-6 to LC-5 This final scenario is the inverse of the previous scenario. We simply decrease the rate of incoming requests, such that 6 instances are no longer required to maintain a *SR* of 1. Thus, after the change in the workload, the system is operating in a sub-optimal configuration, since resources are being unnecessarily consumed by the sixth *Infinispan* instance. While this is not a violation of an exact goal, it is tied to an approximation goal. When the event *Decrease_min_cost_resources* is triggered, the interpreter will select the *RemoveNode* adaptation to decrease the resource consumption. Figure 7.10b shows that after the adaptation, the service ratio is maintained, thus, still processing all incoming requests, and power is saved.

7.3 Discussion

The results show that the proposed approach is able to automatically manage the system during runtime; it successfully satisfies the management objectives in distinct scenarios, taking



(a) Third scenario: adding a new instance



(b) Fourth scenario: removing one instance

Figure 7.10: Experimental results some minutes before and after the adaptation

advantage of the available adaptations to improve the system performance and enforce designated properties. The approach provides enough flexibility to change the system's adaptive behavior, which is achieved by changing the goal policy. Therefore, the adaptation and self-management support does not require re-development. The proposed approach is also easy to extend. From our experience developing the case study, making the system evolve through the addition of new components or more adaptations of existing components can be done without need for a new goal policy. New KPIs and corresponding goals can be added without need to completely re-design the knowledge model and the goal policy.

While other approaches that employ high-level goal policies [BLMR04, SHMK08] lack the ability to balance conflicting goals, the proposed approach is well suited to express the trade-offs found when managing system. During the development of the case study, we encountered several trade-offs related with performance that were easily translated to goals. Nonetheless, it still is necessary to identify which aspects are more important, to rank the goals.

The description of adaptations is a more delicate issue. The proposed approach relies on the

assumption that is possible to establish a reasonable approximation for the impact of adaptations on KPIs in terms of their current value. To describe the impacts, other KPIs or context variables can be employed to allow a better estimation. For instance, the impact on the network utilization of an adaptation that would require notification to be sent to a set of listeners could be expressed as a function of the number of listeners, where *n_listeners* is a context variable whose value is known at runtime: $Component.net\_uti+ = unit\_cost * n\_listeners$. With the case study, we found that, while exact functions tend to be quite complex, rough approximations might be sufficient for the purpose at hand. The case study shows this aspect, using either rough approximations for cases where the impacts are more complex to determine, such as the latency, or the CPU consumption.

In some cases, the adaptation impact may be dependent on the system configuration prior to the system evolution. As a result, instead of a single adaptation, several descriptions may be necessary to cover all the different impacts. The estimation of an adaptation impact may also pose some challenges. For example, the exchange of components requires some experimental testing to quantify the impact of changing from one component to the other.

Still, we recognize that our assumptions may constrain the domain of applicability of the approach. However, it is important to recognize that other approaches also have their drawbacks. For instance, we have experimented with the use of machine learners to predict the behavior of adaptive systems and we were faced with the complexity of feature selection [CRR10]. Therefore, with the current state of the art, it is relevant to explore different directions in the design space. Most likely, future systems will embody a combination of different techniques. We believe that the current work contributes to the understanding of the benefits and limitations of an interesting path that has been under-explored in the literature.

Summary

This chapter presents the evaluation of the proposed approach, using the case study introduced in Chapter 3. It describes the knowledge model and goal policies used to manage the case study and presents the results obtained. The evaluation is separated in non-distributed and distributed components and concludes with a discussion of the obtained results and experiences

while developing the case study.

Chapter 8

Final Remarks

This thesis addresses the challenges of autonomously adapt systems built from multiple adaptable components, which may be distributed. It tackles two particular aspects. One is how the complexity of these systems affects the self-management support, and the other is to minimize the human effort of providing the knowledge and the adaptation logic for the adaptation. Thus, this work proposes a conceptual framework for the self-management of composed systems. The approach allows for action policies to describe the adaptation logic, but also novel goal policies to control the adaptive behavior and explore more efficiently the knowledge of system designers and component developers. The latter depends on an automatic process to generate an action policy that will be used to manage the system behavior. Furthermore, the approach also automatizes the selection of reconfiguration strategies to execute adaptations, using cost functions to select the best strategy.

The evaluation of the proposed approach was focused on the planning and execution activities of the self-management. Several prototypes were developed as proof of concepts of the approach, using different services, middleware, components, and applications. The results show that the prototypes provide the necessary self-management support to control and adapt the managed system. The goal-oriented planning clearly provides a more friendly approach to humans and takes advantage of the detailed knowledge of developers instead of burdening the system designer with all the knowledge gathering. The use of reconfiguration strategies, automatically selected, also relieves the system designer of developing a strategy for each adaptation.

The prototypes were also used in several experiments to explore and study the performance and scalability of the approach, as well as how it handles the distribution of components. In terms of performance and scalability, the approach can handle large numbers of adaptations with reaction times substantially lower than those by human operators. We also explore how the reaction times can be lowered at the expense of optimality, experimenting with different techniques.

During the design, development, and evaluation of the proposed approach, a number of realizations, observations and experiences became obvious. They are addressed next. Moreover, several aspects that could be improved were identified, as well as the open challenges that this approach still faces. They are discussed at the end of the chapter.

8.1 Lessons Learned from Experience

During the development of the approach and the prototypes, we have encountered several obstacles, issues and challenges. This section describes some of the more interesting issues we have found and the lessons learned while trying to overcome them.

In the literature overview, many adaptive solutions are for a particular system, protocol, service or component. These solutions take advantage of some adaptation and evaluate the gains obtained in different scenarios. The adaptation is identified by someone who is experienced with the system or elements, and that has a fairly good knowledge of its use and configurations. From our experience, this is not a trivial or simple matter. To identify the adaptations of components not developed by us, namely Apache HTTP Web Server and *Infinispan*, it was necessary to extensively benchmark and study the components to assess the adaptations, the gains they offer and the trade-offs involved. Furthermore, assessing this in distributed components (*Infinispan* and *JGroups*) increases the complexity of this task. As a result, despite several adaptations available, the effort of studying the adaptations is better spent if limited to those that have more promising impacts on the system behavior.

Benchmarking components or systems to study how the adaptations and different configurations affect their behavior just provides a starting point. In goal-oriented planning, these changes have to be quantified in actual impacts. This depends on the ability to identify the key factors

that affect the component behavior. While for some components this may be straightforward, for others it can be a complex task, as demonstrated by the impact functions of some adaptations of Infinispan. But this effort is compensated by greatly decreasing the system designer effort.

The system designer most important effort is to determine the desired behavior for the system. This may depends on several aspects, and it is usually not just one goal but several goals. In the goal-oriented planning, the description of the goal policy also means identifying the trade-offs and which goals are more important than others. The ranking of goals instead of using utility functions is one of the lessons learned. The description of a goal policy often demands more than one attempt, specially when there are trade-offs involved. Saying which goals are more important than others is far more simple than attributing a weight to a goal and tune it through several attempts. However, we did find some scenarios, as discussed in Section 5.2.6, where the ranking is not enough. In these cases the use of CKPIs allows to have apply weights to only the necessary goals, instead of all.

8.2 Open Challenges and Future Work

In an approach covering so many aspects of the self-management support, there is plenty of room for improvement, at all levels and activities. The planning activity, as one of the key contributions of this work, is the aspect that we are more acquainted with and, thus, can better discuss the challenges and future work directions.

The selection algorithm of the goal-oriented planning is an obvious aspect worth exploring. The algorithm described in this thesis identifies the optimal combination of adaptations based on the goal satisfaction and improvement. However, the algorithm also includes other criteria, such as selecting all adaptations when none satisfies the goal. It would be interesting to enrich the algorithm with different criteria and see how it performs. There are also other possible algorithms to select the optimal combinations; comparing the selections made by other possible algorithms would also be interesting. A comparison between solutions regarding performance, scalability, and reaction time would allow us to assess better the proposed algorithm in this work.

An aspect worth exploring are the variants of the algorithm that trade optimality by smaller reaction times. In this thesis, we addressed three different variants, two cut back on the number of evaluated sets and the other uses a different selection mechanism. It would be interesting to explore other variants and understand the advantages and drawbacks of each, and in which systems they would perform better. It would also be interesting to experiment them not only with larger numbers of adaptations, but also more components, KPIs, and larger goal policies. This would allow us to study how the different factors affect the reaction time.

The description of impacts is another aspect that could be improved. The main concern with impacts, is when the predicted impact does not match the real impact on the system. While the proposed approach can overcome these situations and continue to improve the system, it would be interesting to explore mechanisms that tune the impacts of adaptations. Reinforcement learning is one solution, while prediction of future states could also be used to tune the impacts, using for instance machine learning or Markov chains.

Another aspect that was not explored as in-depth as it should be is the expressiveness of the goal-oriented planning. It would interesting to select a number of policies that employ utility functions and try to describe them using goal policies. This would provide, not only a more complete study of the goal policies+CKPIs solution, but also identify any aspects lacking or further improvements needed.

Bibliography

[AB99] Tarek Abdelzaher and Nina Bhatti. Web content adaptation to improve server overload behavior. *Computer Networks*, 31(11–16):1563–1577, 1999.

[ADG98] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. *Fundamental Approaches to Software Engineering*, 1382 of LNCS:21–37, 1998.

[AHM+03] Ismail Ari, Bo Hong, Ethan Miller, Scott Brandt, and Darrell Long. Managing flash crowds on the internet. In *Eleventh IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems*, pages 246–249, oct. 2003.

[AHW04] Naveed Arshad, Dennis Heimbigner, and Alexander Wolf. A planning based approach to failure recovery in distributed systems. In *First ACM SIGSOFT Workshop on Self-managed systems*, pages 8–12, New York, NY, USA, 2004. ACM.

[Ant06] Richard Anthony. A policy-definition language and prototype implementation library for policy-based autonomic systems. In *Third IEEE International Conference on Autonomic Computing*, pages 265–276, Washington, DC, USA, 2006. IEEE Computer Society.

[Apa] Apache HTTP web server. See `httpd.apache.org`.

[AST09] Reza Asadollahi, Mazeiar Salehie, and Ladan Tahvildari. Starmx: A framework for developing self-managing java-based systems. In *ICSE Workshop on Software*

Engineering for Adaptive and Self-Managing Systems, pages 58–67, Washington, DC, USA, 2009. IEEE Computer Society.

[BB08] Raphael Bahati and Michael Bauer. Adapting to run-time changes in policies driving autonomic management. In *Fourth International Conference on Autonomic and Autonomous Systems*, pages 88–93, Washington, DC, USA, 2008. IEEE Computer Society.

[BCGZ06] Greg Brown, Betty Cheng, Heather Goldsby, and Ji Zhang. Goal-oriented specification of adaptation requirements engineering in adaptive systems. In *International Workshop on Self-adaptation and Self-managing Systems*, pages 23–29, New York, NY, USA, 2006. ACM.

[BLMR04] Arosha Bandara, Emil Lupu, Jonathan Moffett, and Alessandra Russo. A goal-based approach to policy refinement. In *Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 229–239, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[BLMT08] Roberto Bruni, Alberto Lluch, Ugo Montanari, and Emilio Tuosto. Architectural design rewriting as an architecture description language. In Karthik Bhargavan, Andy Gordon, Tim Harris, and Peter Toft, editors, *The Rise and Rise of the Declarative Datacentre*, pages 15–16. Microsoft Research Cambridge, 2008. Technical Report Microsoft Research MSR-TR-2008-61.

[CFSD90] J. Case, M. Fedor, M. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP). RFC 1157, RFC Editor, 1990.

[CGS05] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Making self-adaptation an engineering reality. In *Self-star Properties in Complex Information Systems*, volume 3460 of *Lecture Notes in Computer Science*, pages 158–173, Berlin, Heidelberg, 2005. Springer-Verlag.

[CGS06] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *International Workshop on*

Self-adaptation and self-managing systems, pages 2–8, New York, NY, USA, 2006. ACM Press.

[CHS01] Wen-Ke Chen, M Hiltunen, and R. Schlichting. Constructing adaptive software in distributed systems. In *Twenty-first International Conference on Distributed Computing Systems*, pages 635–643, Washington, DC, USA, 2001. IEEE Computer Society.

[CLG+09] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009.

[CM84] Jo-Mei Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, 1984.

[CRR10] Maria Couceiro, Paolo Romano, and Luís Rodrigues. A machine learning approach to performance prediction of total order broadcast protocols. In *Fourth IEEE International Conference on Self-adaptive and Self-organizing Systems*, pages 184–193, Washington, DC, USA, 2010. IEEE Computer Society.

[DDLS01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, London, UK, 2001. Springer-Verlag.

[Dey01] Anind Dey. Understanding and using context. *Personal Ubiquitous Computing*, 5(1):4–7, 2001.

[DHPB03] Yixin Diao, Joseph Hellerstein, Sujay Parekh, and Joseph Bigus. Managing web server performance with autotune agents. *IBM Systems Journal*, 42(1):136–149, 2003.

[DRPQ12] Diego Didona, Paolo Romano, Sebastiano Peluso, and Francesco Quaglia. Transactional auto scaler: Elastic scaling of in-memory transactional data grids. In *Ninth International Conference on Autonomic Computing*, New York, NY, USA, 2012. ACM Press.

[FHS+06] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, 2006.

[FLR10] João Ferreira, João Leitão, and Luís Rodrigues. A-OSGi: A framework to support the construction of autonomic OSGi-Based applications. In Athanasius Vasilakos, Roberto Beraldi, Roy Friedman, and Marco Mamei, editors, *Autonomic Computing and Communications Systems*, volume 23 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 1–16. Springer Berlin Heidelberg, 2010.

[FRR09] Cristina Fonseca, Liliana Rosa, and Luís Rodrigues. Custo da comutação dinâmica de protocolos de comunicação. In *Actas do primeiro Simpósio de Informática (Inforum)*, Lisboa, Portugal, 2009.

[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(10):46–54, 2004.

[GMMR06] Raffaella Grieco, Delfina Malandrino, Francesca Mazzoni, and Daniele Riboni. Context-aware provision of advanced internet services. In *IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 600–603, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[GSC09] David Garlan, Bradley Schmerl, and Shang-Wen Cheng. Software architecture-based self-adaptation. In Yan Zhang, Laurence Tianruo Yang, and Mieso Denko, editors, *Autonomic Computing and Networking*, pages 31–55. Springer US, 2009.

[GT09] John Georgas and Richard Taylor. Policy-based architectural adaptation management: Robotics domain case studies. *Software Engineering for Self-Adaptive Systems*, pages 89–108, 2009.

[GvdHT09] John Georgas, André van der Hoek, and Richard Taylor. Using architectural models to manage and visualize runtime adaptation. *Computer*, 42:52–60, 2009.

[HIM00] Dan Hirsch, Paola Inverardi, and Ugo Montanari. Reconfiguration of software architecture styles with name mobility. In *Fourth International Conference on Coordination Languages and Models*, pages 148–163, London, UK, 2000. Springer-Verlag.

[HL95] Walter Hürsch and Cristina Lopes. Separation of concerns. Technical report, College of Computer Science, Northeastern University, Boston, 1995.

[HM08] Markus Huebscher and Julie McCann. A survey of autonomic computing – degrees, models, and applications. *ACM Computer Survey*, 40(3):7:1–7:28, August 2008.

[HSMK09] William Heaven, Daniel Sykes, Jeff Magee, and Jeff Kramer. A case study in goal-driven architectural adaptation. In Betty Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 109–127. Springer-Verlag, Berlin, Heidelberg, 2009.

[HSUW00] Matti Hiltunen, Richard Schlichting, Carlos Ugarte, and Gary Wong. Survivability through customization and adaptability: The cactus approach. In *DARPA Information Survivability Conference and Exposition*, volume 1, pages 294–307, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

[IMS06] Gennaro Iaccarino, Delfina Malandrino, and Vittorio Scarano. Personalizable edge services for web accessibility. In *International Cross-disciplinary Workshop on Web accessibility*, pages 23–32, New York, NY, USA, 2006. ACM Press.

[Inf] Infinispan. See `www.jboss.org/infinispan`.

[JGr] JGroups. See `www.jgroups.org/`.

[KC03a] John Keeney and Vinny Cahill. Chisel: A policy-driven, context-aware, dynamic adaptation framework. In *Fourth IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 3–10, Washington, DC, USA, 2003. IEEE Computer Society.

[KC03b] Jeffrey Kephart and David Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[Kep05] Jeffrey Kephart. Research challenges of autonomic computing. In *Twenty-seventh International Conference on Software Engineering*, pages 15–22, New York, NY, USA, 2005. ACM Press.

[KKK96] Thomas Koch, Christoph Krell, and Bernd Kraemer. Policy definition language for automated management of distributed systems. In *Second IEEE International Workshop on Systems Management*, pages 55–64, Washington, DC, USA, 1996. IEEE Computer Society.

[KM98] Jeff Kramer and Jeff Magee. Analysing dynamic change in software architectures: a case study. In *Fourth International Conference on Configurable Distributed Systems*, pages 91–100, Washington, DC, USA, 1998. IEEE Computer Society.

[KM07] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[KT91] Frans Kaashoek and Andrew Tanenbaum. Group communication in the amoeba distributed operating system. In *Eleventh International Conference on Distributed Computing Systems*, pages 222–230, Washington, D.C., USA, 1991. IEEE Computer Society Press.

[KW04] Jeffrey Kephart and William Walsh. An artificial intelligence perspective on autonomic computing policies. In *Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 3–12, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[LCJS01] Yongzin Li, Ming Chen, Xuping Jiang, and Lihua Song. A logic-based policy definition language for network management. In *Twenty-sixth Annual IEEE Conference on Local Computer Networks*, pages 34–40, Washington, DC, USA, 2001. IEEE Computer Society.

[LM98] Daniel Le Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–533, 1998.

[LS98] Gerald Lohse and Peter Spiller. Electronic shopping. *Communications ACM*, 41(7):81–87, 1998.

[LSY03] G. Linden, B. Smith, and J. York. Amazon.com recommendations: item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, jan. 2003.

[LvRB+01] Xiaoming Liu, Robbert van Renesse, Mark Bickford, Christoph Kreitz, and Robert Constable. Protocol switching: Exploiting meta-properties. In *Twenty-first International Conference on Distributed Computing Systems Workshops*, pages 37–42, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[Maz06] Francesca Mazzoni. *Efficient provisioning and adaptation of Web-based services*. PhD thesis, Università di Modena e Reggio Emilia, 2006.

[MB11] Omid Mola and Mike Bauer. Towards cloud management by autonomic manager collaboration. *International Journal of Communications, Network and System Sciences*, 4(12):790–802, 2011.

[MCS00] Bamshad Mobasher, Robert Cooley, and Jaideep Srivastava. Automatic personalization based on web usage mining. *Communications ACM*, 43(8):142–151, 2000.

[MD89] Dennis McCarthy and Umeshwar Dayal. The architecture of an active database management system. In *ACM SIGMOD International Conference on Management of Data*, pages 215–224, New York, NY, USA, 1989. ACM Press.

[MGK96] Kaveh Moazami-Goudarzi and Jeff Kramer. Maintaining node consistency in the face of dynamic change. In *Third International Conference on Configurable Dis-*

tributed Systems, pages 62–69, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 3–14, New York, NY, USA, 1996. ACM Press.

[MLS04] Rebecca Montanari, Emil Lupu, and Cesare Stefanelli. Policy-based dynamic reconfiguration of mobile-code applications. *Computer*, 37(7):73–80, 2004.

[MPR01] Hugo Miranda, Alexandre Pinto, and Luís Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Twenty-first International Conference on Distributed Computing Systems*, pages 707–710, 2001.

[MR06] José Mocito and Luís Rodrigues. Run-time switching between total order algorithms. In *Twelfth International Conference on Parallel Processing*, pages 582–591, Berlin, Heidelberg, 2006. Springer-Verlag.

[MSKC04a] Philip McKinley, Seyed Sadjadi, Eric Kasten, and Betty Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[MSKC04b] Philip K. McKinley, S. Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. A taxonomy of compositional adaptation. Technical Report MSU-CSE-04-17, Department of Computer Science, Michigan State University, East Lansing, Michigan, 2004.

[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, 1999.

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Twentieth International Conference on Software Engineering*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[OMT08] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 899–910, New York, NY, USA, 2008. ACM Press.

[Oqu04] Flavio Oquendo. π-adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *SIGSOFT Software Engineering Notes*, 29(3):1–14, 2004.

[Rad] Radargun. See `sourceforge.net/apps/trac/radargun/`.

[RLR06] Liliana Rosa, Antonia Lopes, and Luís Rodrigues. Policy-driven adaptation of protocol stacks. In *International Conference on Autonomic and Autonomous Systems*, pages 5–11, Washington, DC, USA, 2006. IEEE Computer Society.

[RRL07] Liliana Rosa, Luís Rodrigues, and Antónia Lopes. Building adaptive systems with service composition frameworks. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, volume 4803 of *Lecture Notes in Computer Science*, pages 754–771. Springer Berlin / Heidelberg, 2007.

[SAW94] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *First Workshop on Mobile Computing Systems and Applications*, pages 85–90, Washington, DC, USA, 1994. IEEE Computer Society.

[SEM03] Dale Seborg, Thomas Edgar, and Duncan Mellichamp. *Process dynamics and control*. Wiley, New York, NY, 2003.

[SG96] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[SHMK08] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. From goals to components: a combined approach to self-management. In *International Workshop on Software Engineering for Adaptive and Self-managing Systems*, pages 1–8, New York, NY, USA, 2008. ACM Press.

[SK05] Nancy Samaan and Ahmed Karmouch. An automated policy-based management framework for differentiated communication systems. *IEEE Journal on Selected Areas in Communications*, 23:2236–2247, 2005.

[Slo94] Morris Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2(4):333–360, 1994.

[Sou08] Steve Souders. High-performance web sites. *Communications ACM*, 51(12):36–41, 2008.

[ST07] Mazeiar Salehie and Ladan Tahvildari. A weighted voting mechanism for action selection problem in self-adaptive software. In *First International Conference on Self-Adaptive and Self-Organizing Systems*, SASO '07, pages 328–331, Washington, DC, USA, 2007. IEEE Computer Society.

[ST12] Mazeiar Salehie and Ladan Tahvildari. Towards a goal-driven approach to action selection in self-adaptive software. *Software: Practice and Experience*, 42(2):211–233, 2012.

[Tav10] Tiago Taveira. Adaptive group communication. Master's thesis, Instituto Superior Técnico, Technical University of Lisbon, Portugal, 2010.

[TGM99] Gabriele Taentzer, Michael Goedicke, and Torsten Meyer. Dynamic accommodation of change: Automated architecture configuration of distributed systems. In *Fourteenth International Conference on Automated Software Engineering*, pages 287–290, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[TK04] Gerald Tesauro and Jeffrey Kephart. Utility functions in autonomic systems. In *First International Conference on Autonomic Computing*, pages 70–77, Washington, DC, USA, 2004. IEEE Computer Society.

[VEBD07] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *iEEE Transactions on Software Engineering*, 33(12):856–868, 2007.

[vRBH+98] Robbert van Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using ensemble. *Software: Practice and Experience*, 28(9):963–979, July 1998.

[WLF01] Michel Wermelinger, Antónia Lopes, and José Fiadeiro. A graph based architectural (re)configuration language. In *Eighth European Software Engineering Conference held jointly with Ninth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 21–32, New York, NY, USA, 2001. ACM.

[ZC06] Ji Zhang and Betty Cheng. Model-based development of dynamically adaptive software. In *Twenty-eighth International Conference on Software Engineering*, pages 371–380, New York, NY, USA, 2006. ACM Press.

[ZJY+09] Hui Zhang, Guofei Jiang, Kenji Yoshihira, Haifeng Chen, and Akhilesh Saxena. Resilient workload manager: taming bursty workload of scaling internet applications. In *Sixth international Conference industry session on Autonomic computing and communications industry session*, pages 19–28, New York, NY, USA, 2009. ACM Press.

Appendix A

Publications

The work and results presented in this thesis were partially supported by the Portuguese Science and Technology Foundation, through projects POSI/EIA/60692/2004, PTDC/EIA/71752/2006), and CMU-PT/ELE/0030/2009, and also by the INESC-ID and LASIGE multi annual funding through the PIDDAC Program fund grant.

List of Publications

International Journals

1. Self-management of Adaptable Component-based Applications, L. Rosa, L. Rodrigues, A. Lopes, M. Hiltunen, R. Schlichting. In IEEE Transactions on Software Engineering, 2012, IEEE Computer Society.

Book Papers

2. Dynamic Tuning of Communication Services Using High-level Goal Policies, L. Rosa, L. Rodrigues, A. Lopes. Book 2 on Software Engineering for Self-Adaptive Systems, R. Lemos, H. Giese, H. Muller and M. Shaw (Eds.), 2012, Springer.

International Conferences

3. Goal-oriented Self-management of In-memory Distributed Data Grid Platforms, L. Rosa, L. Rodrigues, A. Lopes. In Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science, 2011, IEEE Computer Society. (short paper)

4. From Local Impact Functions to Global Adaptation of Service Compositions. L. Rosa, L. Rodrigues, A. Lopes, M. Hiltunen, and R. Schlichting. In Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems, 2009, Springer.

5. Modeling Adaptive Services for Distributed Systems. L. Rosa, A. Lopes, L. Rodrigues. In Proceedings of the 23rd Annual ACM Symposium on Applied Computing, 2008, ACM Press.

6. A Framework to Support Multiple Reconfiguration Strategies. L. Rosa, L. Rodrigues, A. Lopes. In Proceedings of the 1st International Conference on Autonomic Computing and Communication Systems, 2007, ACM Press.

7. Building Adaptive Systems with Service Composition Frameworks. L. Rosa, L. Rodrigues, A. Lopes. In Proceedings of the 10th International Symposium on Distributed Objects, Middleware, and Applications, 2007, Springer.

Other Related Publications

8. Large-scale Peer-to-peer Overlay Networks for Autonomic Monitoring. J. Leitão, L. Rosa, L. Rodrigues. In Proceedings of the IEEE Globecom Workshops, USA, 2008, IEEE.

9. Custo da Comutação Dinâmica de Protocolos de Comunicação. C. Fonseca, L. Rosa and L. Rodrigues. Actas do Primeiro Simpósio de Informática (Inforum), Portugal, 2009.