StarMX: A Framework for Developing Self-Managing Software Systems

by

Reza Asadollahi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2009

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The scale of computing systems has extensively grown over the past few decades in order to satisfy emerging business requirements. As a result of this evolution, the complexity of these systems has increased significantly, which has led to many difficulties in managing and administering them. The solution to this problem is to build systems that are capable of managing themselves, given high-level objectives. This vision is also known as *Autonomic Computing*.

A self-managing system is governed by a closed control loop, which is responsible for dynamically monitoring the underlying system, analyzing the observed situation, planning the recovering actions, and executing the plan to maintain the system equilibrium. The realization of such systems poses several developmental and operational challenges, including: developing their architecture, constructing the control loop, and creating services that enable dynamic adaptation behavior. Software frameworks are effective in addressing these challenges: they can simplify the development of such systems by reducing design and implementation efforts, and they provide runtime services for supporting self-managing behavior.

This dissertation presents a novel software framework, called StarMX, for developing adaptive and self-managing Java-based systems. It is a generic configurable framework based on standards and well-established principles, and provides the required features and facilities for the development of such systems. It extensively supports Java Management Extensions (JMX) and is capable of integrating with different policy engines. This allows the developer to incorporate and use these techniques in the design of a control loop in a flexible manner. The control loop is created as a chain of entities, called *processes*, such that each process represents one or more functions of the loop (monitoring, analyzing, planning, and executing). A process is implemented by either a policy language or the Java language. At runtime, the framework invokes the chain of processes in the control loop, providing each one with the required set of objects for monitoring and effecting.

An open source Java-based Voice over IP system, called CC2, is selected as the case study used in a set of experiments that aim to capture a solid understanding of the framework suitability for developing adaptive systems and to improve its feature set. The experiments are also used to evaluate the performance overhead incurred by the framework at runtime. The performance analysis results show the execution time spent in different components, including the framework itself, the policy engine, and the sensors/effectors. The results also reveal that the time spent in the framework is negligible, and it has no considerable impact on the system's overall performance.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Ladan Tahvildari for all her guidance and support over these years. Her advice and encouragement helped me in all the time of research and writing of this thesis.

I would also like to thank my dissertation committee members: Dr. Kostas Kontogiannis and Dr. Rudolph E. Seviora, for having accepted to take the time out of their busy schedules to read my thesis and provide me invaluable comments and inspiring remarks.

I am very grateful to all members of the Software Technologies and Applied Research (STAR) group for their cooperation, and specially to my friend, Mazeiar Salehie, for all his tremendous support, guidance, and thoughtful feedbacks throughout all my research.

My grateful thanks go to my beloved wife, Negin, who has always been extremely understanding and supportive. My love and passion for her go far beyond the expressive power of words.

Last but not least, I would like to thank my lovely four-month-old son, Arvin, for staying calm and letting me work on this dissertation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the past few decades, the complexity of computer-based systems has increased significantly. The new large-scale distributed systems provide lots of facilities and features. They are also integrated into corporate-wide computing systems that provide more services. The ever-increasing complexity of computing systems results in difficulties in managing these systems and maintaining their Quality of Service (QoS) requirements [34, 51]. A solution is to create systems that are able to manage themselves in order to address these issues dynamically, based on a set of high-level objectives, and with minimum human intervention. This approach is also known as *Autonomic Computing* [51].

According to [41], autonomic computing refers to a computing environment that is capable of managing itself, and can dynamically adapt to changes in accordance with business policies and objectives. An autonomic system refers to a system with a *control loop*, which monitors itself and its environment, analyzes the situation, and takes actions to change either the environment or its behavior. Autonomic computing is a broad research area, which includes systems from different domains like hardware, robotics, networks, grid computing, and software. In the context of this research, we focus only on *self-adaptive software systems* .

Self-adaptive software aims to adjust its behavior based on the information it collects from itself and its execution context. Therefore, the system exhibits two basic characteristics: *self-awareness*, which means that the system is aware of its state and behavior, and *context-awareness*, which means that the system is aware of its operational environment [34, 37]. IBM also defines four characteristics of these systems, which reflect different aspects of self-management [41]:

- *Self-configuring*: The system capability of adapting to changing conditions and ad-

justing itself automatically. It also enables addition/removal of components to/from the system dynamically.

- *Self-optimizing*: The system capability of monitoring and measuring performance related parameters in varying conditions and optimizing its behavior in order to meet performance objectives.
- *Self-healing*: The ability to detect, diagnose, and repair problems. This property may also enable the system to proactively prevent failures from happening in the future. It improves software reliability and availability.
- *Self-protecting*: The ability of a system to detect malicious attacks and to defend itself against them.

All of these properties are often called self-* properties, and a hierarchical view of them is presented in [78] (See Figure 1.1). The picture shows that self-awareness and context-awareness are fundamental characteristics of other properties, and self-adaptiveness or self-managing is achieved by enabling major level properties.

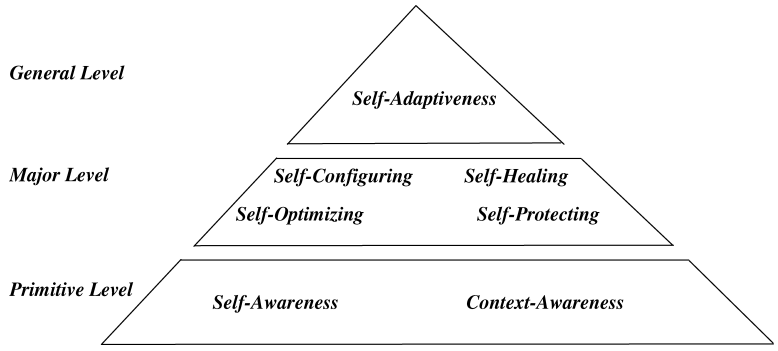

Figure 1.1: Self-* Properties Hierarchy

In the literature, there is no clear distinction between the terms *autonomic*, *adaptive*, *self-adaptive*, and *self-managing* system. In this dissertation, all of these terms are used to refer to the same concept and are used interchangeably.

1.1 Problem Description

Researchers aim to provide solutions to enable self-* properties and to incorporate self-adaptation behavior in software systems. Kephart [50] categorizes the research challenges involved in the realization of self-managing solutions. In particular, the design and architectural concerns of self-managing components and systems, as well as the related technologies that support fundamental issues such as monitoring, event correlation, and constructing a feedback loop are among these challenges. Software frameworks can effectively support this category of challenges. They can provide a design solution and a reusable code base for common and recurring problems in the construction of such systems, thereby simplifying the development of self-managing systems, and eliminating the need to reinvent the wheel.

A key characteristic of such a framework is reusability, which extends the usage domain of the framework. The proposed software frameworks for self-adaptation are mostly designed for a particular problem, have dependencies on other frameworks or execution environments, or lack some necessary features. These limitations affect their reusability and generality. This is one of the current research gaps in this area.

Moreover, self-adaptation solutions should rely on standards and well-established principles to enable interaction between the management layer components and the underlying system [41]. This aspect can also be supported by an appropriate framework. Two key enabling technologies in this area that have been improved in recent years are:

- Java Management Extension [82]: The *de facto* standard for application management in enterprise Java-based systems.
- Policy engine: An engine for execution policies. Policies are extensively used for describing adaptation logic, and there are many commercial and open source policy engines.

These two techniques are standard solutions in the field, and they complement each other. There is therefore an opportunity to utilize and integrate these solutions by a framework, in order to facilitate building self-managing systems.

This research effort presents a novel software framework as a step towards addressing the mentioned challenges and filling the gap in the research area. The proposed framework provides a set of fundamental features and facilities for creating self-managing solutions. These elements should be utilized by a developer to build a complete solution. For example, the control loop is a common pattern of such systems. Using our framework, the developer does not need to think about the architectural design of the loop; instead he/she can concentrate on the logic of the control loop.

1.2 Thesis Contributions

The major contribution of this thesis is to provide a solution to the common problems of self-managing systems. It presents a novel software framework that simplifies the development of such systems and improves productivity. It is able to integrate different available techniques and mechanisms to support self-adaptation (*e.g.* JMX or policies). It aims to separate the generic aspects of self-adaptation from its problem specific parts, and provides a reusable solution for recurring problems in the domain of Java-based systems. It is generic, in the sense that it has no dependency on any specific software tool and can be used to address different self-* properties. The developer utilizes the framework provided features, configures the framework, and deploys it to the operational environment to enable self-management behavior.

The following summarizes the contributions of this research:

- It presents a simple and flexible approach to construct control loop via a configurable sequence of processes.
- It supports different mechanisms for describing management logic (*e.g.* programmatic or descriptive), which specifies what should be done and when.
- It enables access to different sensor and effector components through a standard approach.
- It provides solutions to common monitoring techniques, which can be incorporated into the control loops.
- It effectively addresses the separation-of-concerns principle by keeping management logic separate from application logic.
- It supports several mechanisms to enable communication between self-managing elements.
- It provides a runtime infrastructure and environment that contains self-managing elements and enables dynamic adaptation behavior.
- It prepares a platform for more research efforts in this area, allowing different adaptation solutions to be implemented and tested.
- It is available as an open-source project, which enables its extension and usage by other researchers and industry practitioners.

1.3 Thesis Organization

The rest of this thesis is organized as follows:

Chapter 2 presents a literature review of research related to this work. It first outlines useful techniques for enabling self-adaptation behavior. It then summarizes a set of software frameworks that support dynamic adaptation. A discussion on solutions towards self-healing problems is presented next, followed by a section that describes the approaches for designing self-managing systems. Finally, some experimental efforts arereported.

Chapter 3 provides an overview of the proposed framework and its conceptual architecture. First, a brief discussion on the importance and role of software frameworks in building large software systems is presented. Then, the functional and non-functional requirements of our framework are outlined. The next section describes how different technologies are integrated into the framework to provide a management platform for different applications. The last section presents the architecture of our work from a high-level point of view and elaborates its key features and characteristics.

Chapter 4 explains the architecture of our framework in terms of its components and their interactions. In the first section, the core component of architecture, *i.e.* the *execution engine*, is described, followed by a discussion on how control loops are supported in the architecture. Next, several services provided by the framework to enable its runtime behavior are outlined. Finally, we demonstrate how the framework operates at runtime and how self-managing behavior is enabled by the system.

Chapter 5 describes a five-step process for developing a self-managing system using the proposed framework. It first defines the required steps of the process. It then gives some examples and sample scenarios to clarify the process. The next section presents the configuration properties of the framework, along with their meanings and attributes.

Chapter 6 reports the conducted experimental studies that evaluate the effectiveness of our framework. In the first section, we describe the case study, which is a voice-over-IP system. In the next section, we define the research objectives of the experiments and how they are analyzed. The last section provides a discussion on the results of the experiments and reports framework performance analysis results.

Finally, Chapter 7 finishes the thesis by drawing conclusions from the presented research. It discusses future directions for the research, and outlines some concluding remarks.

Chapter 2

Related Work

Researchers and practitioners have proposed a variety of approaches and techniques to support dynamic adaptation and management. These solutions address different aspects of the self-adaptation problem, and they can be classified, for example, based on how, when, and where adaptation takes place in the system [62]. A comprehensive taxonomy is also presented in [78] to categorize self-adaption solutions. This is a hierarchical taxonomy, and its first level includes: a) Object to Adapt, which deals with what and where questions; b) Realization Issues, which correspond to how questions; c) Temporal Characteristics, which concern the when aspect; and d) Interaction Concerns, which deal with where, when, what, and how questions. These taxonomies cover a broad range of research in the field, and are suitable for analyzing and comparing the quality of adaptation achieved by different solutions. Considering the scope of this research and its focus of attention, we categorize the related research efforts as follows.

Section 2.1 outlines common techniques used for sensing and effecting purposes. These techniques are often utilized by different frameworks to create a control loop. Section 2.2 describes more closely related topics to our work, including several approaches that present an infrastructure or a framework to support self-management at runtime. It also discuses several design solutions that facilitate building autonomic elements or systems. Solutions that are concerned only with a specific self-* property, like self-healing, are covered in Section 2.3. Section 2.4 explains several conceptual models and languages, which aim to capture the adaptation behavior and to correlate adaptation actions to system events. Finally, various empirical studies and experimental research, which are not supported by a framework but related to the topic of this research, are discussed in Section 2.5.

2.1 Enabling Techniques

Separation-of-Concerns is considered to be one of the enabling technologies for building self-managing systems [62]. Adaptation logic is a cross-cutting concern and must be maintained separate from business logic. Separating such concerns from the functional logic facilitates development and maintenance of the system. *Aspect Oriented Programming* (AOP) is a widely used technique that enables this property. AOP provides a facility to develop cross-cutting concerns in modules called *aspects*. Aspects are merged with the application code by a separate compiler or dynamically. This process is called *weaving*, and the compiler is the *aspect weaver*. The locations in the code that are woven by aspects are *point-cuts*.

Sensors and effectors are parts of the adaptation logic that directly interact with the system components or resources. AOP is a useful technique to build these components [62, 78]. Several research efforts utilize this approach; for example, CASA [65], TRAP/J [73], and TRAP/BPEL [26]. [75] also reports some experience in using dynamic aspects for sensing and effecting. Quality Objects (QuO) [58] is an aspect-based framework for developing distributed adaptive applications with QoS requirements. It proposes a set of aspect languages to define QoS states, mechanisms for monitoring resources, and adaptation behavior. The QuO adaptation model and aspect model are elaborated in [25] and are compared with other aspect-oriented languages like AspectJ [8].

Design patterns provide reusable solutions to recurring problems in software systems. Several design patterns including Wrapper, Proxy, and Strategy [28] are mentioned as effective patterns to implement adaptation solutions, particularly for effectors to change some artifacts and apply actions [62, 78]. Such patterns facilitate interactions between effector components and underlying resources. CASA [66] employs the Bridge design pattern [28] to replace a component dynamically. [58] and [73] also use the Wrapper design pattern. In another research effort, the Observer [28] pattern is used for monitoring and the Strategy [28] pattern is used for effecting [11]. Moreover, Oreizy *et al.* [71] discuss conductive architectural patterns, such as Pipe-and-Filter and Publish-Subscribe, which are utilized by [7, 79] for supporting dynamic adaptation behavior.

Using *policies* or rules in the *if-condition-then-action* format is cited as an effective technique in autonomic systems to declare the required adaptation behavior (*e.g.* [3, 15, 49, 59, 81, 89]). The main advantage of this approach is its simplicity. Policies can be used as a simple technique to create a closed control loop such that appropriate actions are executed when special conditions are met.

Computational reflection is another key technology for self-adaptive systems [62]. This refers to the system's ability to observe or to alter itself dynamically. Reflection can be

either structural or behavioral. It consists of two activities: introspection to observe the current state, and intercession to modify the system. The Apapta framework [79], described in the next section, and TRAP/J [73] have been designed based on this concept. TRAP/J is a software tool for making Java applications adaptable. It focuses on effecting techniques and applies behavioral reflection and AOP to achieve its goal. It augments the program code with extra code and aspects to transform Java classes into adaptable code.

Another well-known technique for sensing and effecting is *Java Management Extensions* (JMX) [82]. It provides a standard instrumentation mechanism to observe system state and to invoke management commands. Besides our framework, this approach is supported by other research efforts, for example [2] and [31].

Using Web Services as instrumentation interfaces of systems is an emerging standard. For example, Web Services Distributed Management (WSDM) [69] defines a standard for managing heterogenous resources. In this approach, web services are used as sensors and effectors of remote systems. Martin *et al.* report their experience in using the WSDM standard and autonomic Web Services [60].

2.2 Adaptation Frameworks

Several software frameworks and tools have been proposed by researchers and practitioners to assist in enabling self-managing capabilities and building autonomic systems. They aim to support building a complete control loop to address self-management concerns. Some of these solutions have been designed for particular techniques, while others offer more generic approaches applicable to different contexts. However, one of the common properties in all these solutions is their support for the separation-of-concerns principle, which enables maintaining the adaptation logic separate from the application code. In this section, we elaborate several frameworks from the literature and outline their key capabilities and features.

Rainbow is a framework for supporting self-management concerns using knowledge of the system architecture [29]. The adaptation infrastructure consists of components categorized into three layers: the *system-layer*, which includes sensors and effectors; the *architecture-layer*, which serves as the adaptation engine and includes different functions for the control loop; and the *translation-layer*, which maps low-level system properties to their corresponding architectural attributes. The adaptation logic is defined by a specific language supplied by the framework [19]. It captures the adaptation logic in the form of *tactics*, which represent a condition-action scenario for a specific problem; and *strategies*,

which represent a flow of actions with decision points to fix system problems. Each strategy is defined as a tree of tactics that tackles common quality issues, with conditions describing each branch. However, the difficulty of describing the systems architecture as expected by the framework, and its limitations in dealing with environmental properties are some of the concerns with using this solution.

In another work, the Accord framework is presented to support the development of autonomic elements and their composition via dynamic rule definition [57, 56]. Autonomic elements are defined by means of the *functional* port, which serves as the functional interface of the managed component; the *control* port, which serves as set of sensors and actuators and the constraint set that controls access to them, and the *operational* port, which serves as the interface to formulate, inject, and manage rules, and encapsulates a set of rules used to manage the runtime behavior of the element. To enable the dynamic management of components, it proposes a rule language as well as a rule execution engine. It is a part of a big project, and has been designed based on the assumption that several other frameworks are available in the execution environment. This dependency issue affects the reusability of the framework in different environments.

The J3 Process, presented in [87, 88], proposes a model-driven environment to support fine-grained self-management. It consists of a development tool and a framework that supports required runtime services. In this process, the management logic is visually defined at the component level in some models using *J2EEML*. The models are used by *Jadapt* to generate the artifacts required at runtime for self-management. *JFense* is a reusable framework, which uses the generated artifacts at runtime for monitoring, analyzing, planning, and executing. The J3 process is limited to deal with EJB components only and cannot be utilized for managing different types of resources or system-wide properties.

Adaptive Server Framework (ASF) was designed to support adaption behavior in server-based systems [31]. It helps the separation of management logic from business requirements. The ASF architecture consists of two layers: the *management-layer*, which includes mechanisms and services used to monitor the runtime behavior of the application, and the *adaptive-component-layer*, which consists of several components to collect data, analyze the state, and tune the system's behavior. To construct a control loop, sensor, monitor, analyzer, and effector components must be developed using pre-defined interfaces. Their composition is defined in a configuration file, and they are governed by a set of policies presented in XML format. More information about this framework and its evaluation results can also be found in [32]. ASF states that the management layer is based on JMX, but the role of this technology and its relationship with sensor and effector components have not been clearly specified. The XML format for policies looks too verbose.

Autonomic Management Toolkit (AMT) [2] is also an adaptation support framework, which employs rule engines for reasoning and decision making, and JMX for sensing and effecting purposes. It can be integrated with different rule engines through an interface, which eliminates the dependency on a particular decision-making engine and enables switching to other engines. This framework is limited to rules for describing management logic and supports no other mechanisms.

Mukhija *et al.* present CASA (Contract-based Adaptive Software Architecture) as a framework to enable dynamic adaptation via component recomposition at runtime [65, 66, 67]. CASA offers a runtime environment to monitor changes in the system and to apply adaptations when needed. Adaptation policies are defined in the form of XML-based contracts, thereby separating adaptation concerns from the application logic. This framework supports different adaptation techniques, including: dynamic change in lower-level services, dynamic weaving and unweaving of aspects, dynamic change in application attributes, and dynamic recomposition of components, in order to address adaptation requirements in a diverse set of systems. CASA monitors the system at runtime, and in the case of detecting a change, it evaluates the application contracts and applies the required adaptations as described in the contract. As mentioned before, it employs AOP and the Bridge design pattern as two enabling techniques for effecting. The dynamic recomposition of components imposes several constraints on the design of application components, and its performance is arguable.

Presented in [79], Adapta is a framework for developing self-adaptive component-based systems. It provides a runtime environment for monitoring different properties of system resources and generating events if changes are detected. The events notify the adaptation engine to execute reconfiguration actions. The adaptable elements and reconfigurable actions are defined using an XML-based language, called *AdaptaML*. Any modification to this XML is detected by the framework at runtime, allowing dynamic introduction or removal of adaptation actions. The adaptation actions are *updating application parameters* and *replacing algorithms* with a well-defined state transfer mechanism. Built above the CORBA middleware, this framework utilizes the Publish/Subscribe design pattern to handle events, and computation reflection to apply reconfiguration actions. Although it provides a distributed event processing mechanism, it offers a limited set of adaptation actions, which is attributed to the lack of a standard approach in sensing and effecting.

Another group of research in this area present a multi-agent view of autonomic systems. The Cognitive Agent Architecture (COUGAAR) [35] is an open source distributed agent infrastructure that provides foundations for several research efforts. Gracanin *et al.* [33] present a two-layer architecture based on COUGAAR for developing autonomic applications. The architectural layers provide a translation between domain specific concerns and

COUGAAR facilities to support dynamic adaptation. [44] also presents an infrastructure on top of the COUGAAR architecture, which maps different COUGAAR features to autonomic computing concepts, enabling self-managing system development. For example, it utilizes the COUGAAR plug-in model to construct the MAPE loop components, and the blackboard model as the communication medium. DOSE [13] is another agent-based platform capable of semantically searching web resources. It is an autonomic system that can manage and optimize its knowledge-base through a set of sensors and interaction with external services [12].

2.3 Problem-Specific Solutions

Several research efforts aim to address only particular aspects of autonomic systems. A majority of this group deal with fault recovery and reliability/dependability improvement, enabling systems with self-healing capabilities. [22] and [30] propose approaches that rely on the architectural model of the system to provide self-healing. In [80], Shin presents a two-layer model for designing self-healing components. One layer is the healing layer and the other is the service layer. Bellur *et al.* [10] propose a probabilistic model based on Bayesian Belief Networks (BBNs) to capture failure propagation into system components, and use the model to identify and isolate the root cause of failure and then to initiate a recovery plan.

Candea *et al.* present an autonomous recovery technique supported by a framework to reduce downtime in J2EE systems [16, 17]. The approach utilizes a technique called micro-reboot (component reboot) as the means of recovery and repair to address self-healing concerns in these systems. The framework detects the anomalies at runtime and reports them to the recovery manager module to decide which components must be restarted. While the framework is effective in particular cases, it is limited to the reboot technique and cannot be considered a general-purpose autonomic framework for self-healing.

2.4 Design Solutions for Dynamic Adaptation

A different category of research in this area provides architectural or design solutions for self-adaptive and self-managing systems. One of the most widely referenced efforts is IBM's architectural blue print for autonomic computing [41]. IBM describes the design of *autonomic manager*s, as the adaptation regulator entities, and their orchestration to construct a full autonomic architecture. An autonomic manager is a component that

implements a control loop to automate the management functions and to externalize the management logic from the application resource. The control loop is often called MAPE loop, which is comprised of four modules: *monitoring*, to collect needed data from the system; *analyzing*, to diagnose and reason about the observed situation; *planning*, to decide about the appropriate actions; and *executing*, to perform the actions. These modules are supported by a *knowledge-base* containing the guidelines and policies. Autonomic manager is also considered to be a manageable resource that can be controlled by other autonomic managers.

Kramer and Magee suggest a three-layer architecture model for handling self-management concerns [52]. The layers are hierarchical and deal with event at different levels of abstraction. The lowest layer is close to the real system and provides fast responses to the changes in the environment. If it is unable to reasonably react to that event, it reports the event to the next layer, which has more information about the system objectives and other components. This layer in turn tries to manage the situation and will report it to the top-most layer to be processed.

Goal-Attribute-Action-Model (GAAM) [77] is an approach for enabling adaptive behavior based on the system goals. This approach models the system's high-level objectives in the form of *goals*. It identifies the system attributes to be observed and the actions to be taken. Each goal provides a logical relationship between a set of attributes and a set of actions, which is defined by the conditions of the attributes and a prioritized list of appropriate actions. Goals are continuously evaluated to check whether they are satisfied. There is also a *voter* component, which collects the preferred action list of different goals and makes the final decision by incorporating the knowledge of conflicting actions and their priorities. The output of the voter will be the final set of actions that should be executed in response to the observed situation.

Autonomic System Specification Language (ASSL) is a formal language for describing autonomic systems [84]. It is used to formally define an autonomic system architecture and properties of its autonomic elements. The language models the system architecture in three tiers: a) *Autonomic System*, which includes high-level service level objectives and self-managing policies; b) *Autonomic System Interaction Protocol*, which includes messages, negotiation protocols and communication channels; and c) *Autonomic Element*, which consists of detailed service level objectives, policies, events, actions, etc. The architectures of the autonomic elements are further elaborated in [85], which shows their functional units and their centralized and distributed architectural styles.

An architectural solution for managing server-based systems is also presented in [90], which has the capabilities of both predictive and reactive autonomicity approaches. In

this model, a generic controller system is designed to provide predictive decisions by a *feed-forward* control strategy based on the inputs to the controlled system; and to support reactive behavior by a *feed-back* strategy based on the results of comparing the desired behavior and the observed data. Several other modules including the data collector, data processor, predictor, and comparator are also involved in supporting the controller module. In a different research effort, the key considerations for designing autonomic systems and autonomic elements are discussed [89]. The required supportive services (e.g. registry, broker, and negotiator) and design patterns for addressing different self-* properties are also described.

2.5 Other Experimental Researches

In this section, we outline a few other related studies that report experience in building self-managing systems.

In [1], Abdellatif describes how the management of middleware can be improved by a component-based design. The research concentrates on J2EE application servers and shows how a fractal component model [14] is suitable for designing an arbitrary number of controllers for both primitive and composite components. In this model, each component is a unit of management and can be reconfigured at runtime. This approach is evaluated for managing the application server to provide the best performance and availability for the deployed applications and services. Desertot *et al.* [23] also present a J2EE server model that is capable of addition/removal of server instances dynamically to/from a server cluster to deal with load variability at runtime.

In the context of service-oriented architectures (SOA), in which applications consist of several services, dynamic service adaptation has gained a lot of attention in recent years, due to the importance of service availability. One technique to address this challenge is to replace a service implementation dynamically when the service becomes faulty [42]. This study presents a framework and employs the proxy design pattern to replace the service smoothly.

2.6 Summary

This section outlined a group of research efforts related to the topic of this thesis. We learned that no one adaptation approach can be preferred over all others, and that there is

no agreed-upon solution to meet the concerns and challenges of software adaptation. We also demonstrated that there is a diverse set of studies in this area and that researchers deal with the problems from different perspectives and at different levels of abstractions ranging from conceptual to implementation-specific approaches. The key is to understand their advantages and limitations and either combine them or use each in their best performance environment. Another observation is that the presented frameworks for enabling dynamic adaptation are not generic enough to be applicable in different contexts, and that they have been designed to provide a particular solution to a specific problem. This is one of our motivations to develop a multi-purpose generic adaptation infrastructure, which allows the developer to incorporate his/her desired techniques, while providing solutions for recurring problems in the autonomic domain.

The next two chapters will describe the design and architecture of our proposed framework. Chapter 3 provides a high-level view of the framework, while Chapter 4 presents more detail by discussing its internal architecture and behavior.

Chapter 3

Proposed Framework: StarMX

This chapter presents an overview of the proposed framework, called StarMX, which aims to address the challenges of self-managing systems development. StarMX is a generic framework which incorporates dynamic adaptation behavior into software systems and facilitates enabling self-* properties in them. It also shapes the design of self-managing systems by providing an infrastructure and fundamental features for constructing closed control loops. The motivation for this research and for designing this framework originates from the lack of a reusable design approach that is applicable in different contexts and suitable for addressing different self-managing aspects. This framework has been released as an open source project [1], allowing researchers and industry practitioners to work with it and to provide feedback to improve it.

In this chapter, we first provide a brief overview of software frameworks and their role in software development. Next, we discuss the key requirements for enabling self-managing solutions. We also present the technologies supported by the proposed framework and its conceptual architecture.

3.1 Software Frameworks: Concepts and Definitions

In the object-oriented domain, a software framework is defined as a reusable, semi-complete software that provides an abstract design for a family of related problems, which can be instantiated in different applications [47]. In relation to architecture, it is defined as "a micro-architecture which provides an incomplete template for the system within a specific

[1] http://sourceforge.net/projects/starmx/

domain" [43]. According to the definitions, a key characteristic of a framework is that it is intrinsically *incomplete*, which means that there are certain classes or methods used by the framework that are missing, and they must be provided by the developer to instantiate the framework. These missing parts are often called *slots* or *hooks*, which enable integrating the framework with its usage environment [55].

Nowadays, software frameworks are widely used in industry for addressing different technical problems. Open source communities have considerably improved the quality of software and reduced development effort by providing reusable, and platform specific/independent frameworks based on recent technologies. For example, Hibernate [36] is a very popular object-relational mapping persistence framework used in Java-based applications, and Struts [6] is a framework which facilitates web tier component development in server side Java EE systems.

Fayad *et al.* [27] summarize the key benefits of object-oriented software frameworks as follows:

- **Reusability**: A framework provides design and code reuse and substantially improves productivity by eliminating the need to rebuild and revalidate the solution to a recurring problem.
- **Inversion of Control**: This is the runtime characteristic of a framework, which differentiates it from a class library. A framework maintains its own thread of control, manages the sequence of execution, and invokes application objects at appropriate moments through well-defined interfaces.
- **Modularity**: A framework encapsulate the design and implementation of a specific problem and enhances system modularity.
- **Extensibility**: A framework allows extending or overriding its behavior at certain places by providing stable interfaces and hook methods.

Frameworks also have close relationships with other reuse techniques like *patterns* and *class libraries*. The main similarities and differences of a framework with the mentioned techniques can be summarized as:

- **Framework vs. Pattern**: A pattern is only a design solution to a recurring problem. It is of logical nature and is language independent. A framework provides both design and implementation solutions for a problem in a specific programming language. A framework may also include the physical realization of several design patterns to address a problem [21].

- **Framework vs. Library**: A library is a collection of implementation classes that provide a set of services. A Framework is an extension of a library with inversion of control capability. It is active, has its own sequence of control, and invokes application objects, while a library is typically passive and is invoked by the application.

Software frameworks facilitate the design and implementation of software systems by enhancing reusability and improving productivity. Therefore, in designing a framework, it is important to take into account the point of view of a developer who wants to use the framework. The design challenges of a framework include *development effort*, *learning curve*, and *integrability* [27]. The quality of addressing these challenges affects the usability and reusability of the framework:

- **Development effort**: the amount of effort needed to instantiate and adapt a framework in a particular context.
- **Learning curve**: the period of time needed for a typical developer to learn when and how to use the framework properly. It is also defined as the complexity of the framework from the developer's point of view
- **Integrability**: ease with which the framework can be integrated with a usage environment. Standard-compliant frameworks have better integrability with their execution contexts.

3.2 Requirement Definition

The StarMX framework aims to support dynamic adaptation behavior in software systems and to facilitate developing self-managing systems. Therefore, the framework must provide an appropriate infrastructure and facilities to help the developer to build a self-managing environment. It must capture the commonalities in the design and development of such systems and should not enforce a particular approach to the problem. Some of the basic design concerns that a developer must deal with and needs to find a solution for are listed here:

- *How to build closed control loops and create autonomic elements?*
- *How to find and access the sensor and effector components within the control loop modules?*

- *How to enable the control loops to perform adaptation?*
- *How to design a knowledge repository for the control loops and enable data communication among different control loops?* The repository can be used to store the historical data related to the system state observations and the decisions made in the control loop in response to the discovered problems.

To achieve the framework objectives and to clearly address its desired goals, we have identified a set of key requirements. They are used as the basis for designing our framework's architecture. The requirements are categorized as being *functional* or *non-function*. Although self-managing requirements are considered to be non-functional requirements of software systems, in the context of our framework, the requirements that deal with self-management capabilities are the core functionalities of the framework. Non-functional requirements take quality attributes like performance and scalability into account.

The set of functional requirements include:

- **FR1: Control loop creation** - The framework must provide facilities for creating control loop, which is the key characteristic of a self-managing system that correlates sensory inputs to adaptation actions. This facility must be flexible in design, in the sense that it should provide the opportunity for the developer to apply different techniques or algorithms for self-management.
- **FR2: Compatibility with IBM's architecture** - The reference architecture for autonomic managers, presented in [41], is one of the standard approaches in this research area. The StraMX framework must enable the realization of that architecture and allow incorporating modules for monitoring, analyzing, planning, and executing. Section 4.1.2 discusses how we addressed this requirement.
- **FR3: Sensor/effector lookup** - Sensor and effector components are used in the control loop to observe system state and to apply adaptation actions. The framework must support mechanisms to enable access these components from the control loop.

  Regarding the design of sensor and effector components, we believe that these components are problem specific and their internal designs depend on the domain and the software application designated to be managed. These components usually interact with other components in the target system and there is a coupling between them. For example, a sensor component that provides some information about the system's current performance extracts this data from the internal system components or the application server resources, and it may not be a generic component. Therefore, the

internal design of these components is not within the scope of our framework. Instead, StarMX should support standard mechanisms, like JMX, to access the components.

- **FR4: Control loop activation** - The framework must provide a means for triggering control loop execution at runtime. These are often called monitoring techniques: two common methods are *timer-based* and *event-based*.

- **FR5: Information repository** - To store important data about the current status of the system at runtime, an information repository is required. The stored historical data are used for better decision making and management in future.

- **FR6: Communication** - A self-managing system typically contains several control loops to deal with different aspects of self-management at the level of different resources or components. Each control loop may require some information about other control loops to make a decision, or it may need to notify another control loop about the occurrence of an event. The framework must provide mechanisms which enable these kinds of communications.

- **FR7: Separation of Concerns** - Self-managing requirements are considered as non-functional requirements of a system and they should be maintained separately from those systems. The proposed framework must be designed to address this principle.

- **FR8: Standard-based** - The framework must take advantage of existing standards and well-established principles (such as JMX and policies) instead of proposing new concepts, languages, or models to address the above requirements. This property will enhance reusability, minimize the learning effort, and facilitate framework integration with different target contexts.

In the non-functional requirements, we are more concerned with the following properties. However, several other quality attributes like reusability, flexibility, and extensibility should have also been considered in the design of this framework in order to improve the quality of the final product.

- **NFR1: Performance** - Performance is one of the major concerns in self-managing systems. Adaptation logic will affect the overall system performance due to the cost of sensing, effecting, and decision making. This framework aims to minimizes the performance overhead by applying efficient techniques. Our performance objective is to reduce the amount of time spent in the framework to lower than *1 millisecond* per adaptation (This consideration is based on our previous experience in this field).

- **NFR2: Scalability** - This property shows how well a system can handle a growing amount of load. Because of adaptation, if the performance overhead imposed on the system is significant, the scalability of the framework becomes more important. The framework must be able to transfer the adaptation load to another machine or remotely manage the system.

- **NFR3: Simplicity** - The framework utilization and employment must be simple, in order to reduce the development effort and to decrease the learning curve as well.

- **NFR4: Manageability of the framework itself** - As a software component that exists in the system at runtime, the framework may face situations that need dynamic adaptation. The framework should expose a set of sensor and effector components as its manageability interfaces to be used by other management frameworks or consoles.

In the following sections, we describe the framework's conceptual architecture and show how these requirements are addressed.

3.3 Enabling Technologies

Different technologies support the realization of self-managing solutions by providing mechanisms for sensing and effecting or describing management logic. StarMX utilizes *JMX* and *policy engine* as two standard technologies, and combines them into an integrated management framework to provide an infrastructure for building self-managing applications. In short, StarMX enables using JMX sensors and effectors in policies for decision making. Supporting these two technologies, makes the framework *standard-compliant* and enhances its usability and reusability, thereby addressing *FR8*.

Figure 3.1 illustrates an external view of StarMX and its interactions with JMX and a policy engine. The presented combination provides an infrastructure to manage the underlying software application and to create a self-managing environment. Note that the arrows show the direction of interactions.

3.3.1 Java Management Extensions (JMX)

JMX provides a standard approach for application instrumentation for the purpose of state observation and command invocation in the domain of Java-based systems. It presents an

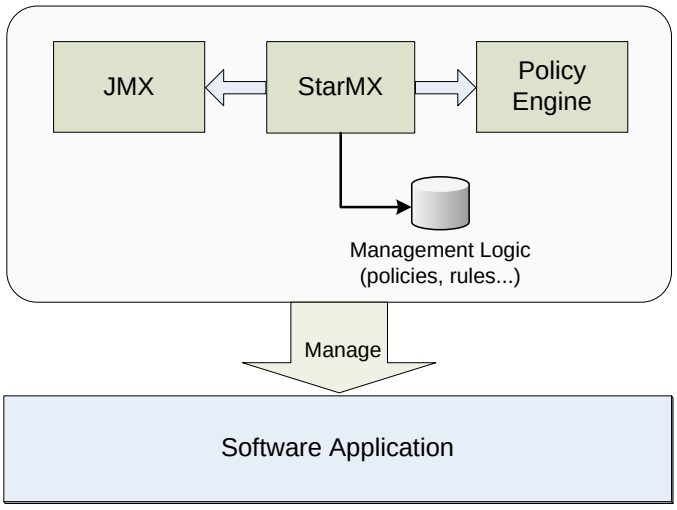

Figure 3.1: Self-Managing Application Enabled by StarMX, JMX, and Policy Engine

architecture, design patterns, and APIs for application management. In this section, we present a brief overview of this technology, followed by its relationship with StarMX.

The JMX architecture offers the following main benefits to applications:

- It provides a standard mechanism for managing heterogenous resources in Java applications.
- It presents a scalable management architecture so that application management can be distributed across different machines.
- It can be integrated with other management solutions using its well-defined APIs.

Figure 3.2 shows the JMX architecture and the relationship between its components [83]. This architecture is characterized by three levels: *Instrumentation level*, *Agent level*, and *Distributed Services level*.

The instrumentation level provides mechanism to make a resource manageable. A resource can be anything like an application, a component, a user, a database and so forth. Resource instrumentation is enabled by means of objects called Managed Beans or MBeans. These are simple Java objects that interact with the underlying resources and expose interfaces to make them manageable through the agent level. Several design patterns have been proposed, for example in [38, 53], for effectively using MBeans and

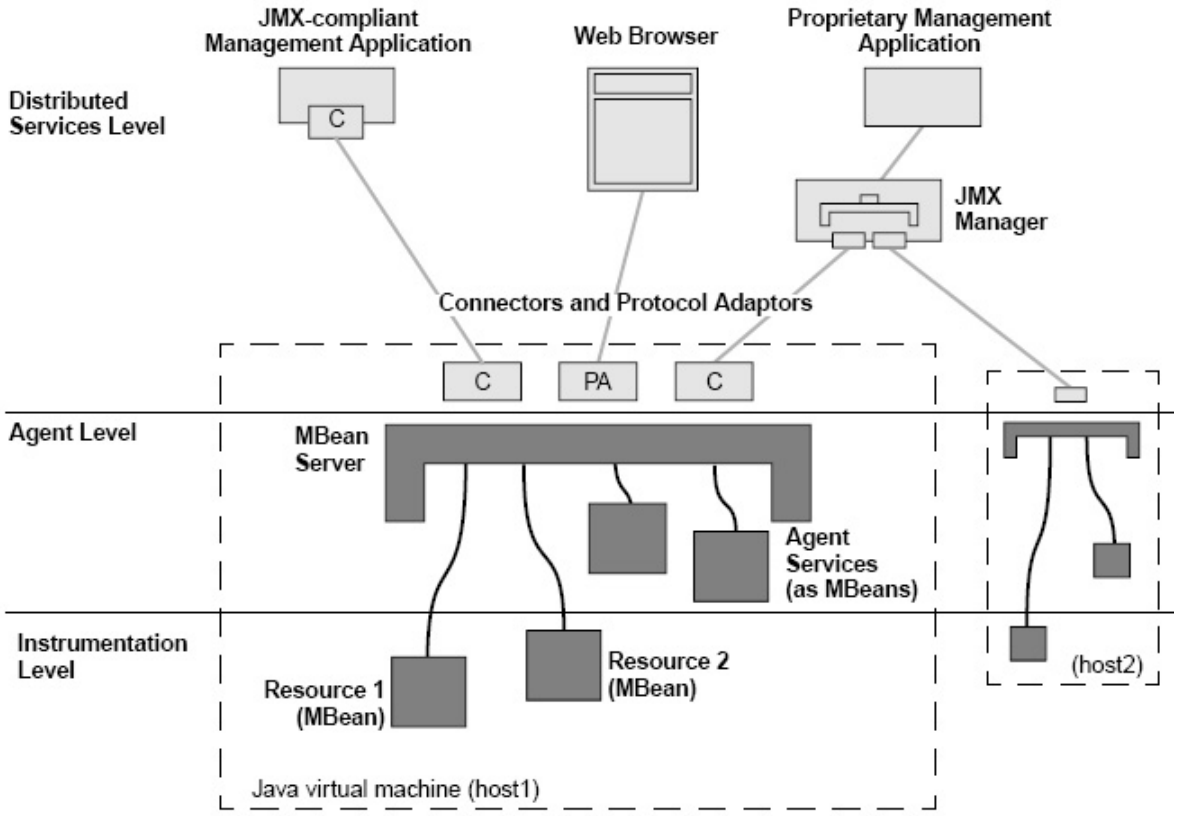

Figure 3.2: JMX Architecture

designing the interaction between MBeans and the real system resources. Moreover, this level offers a notification mechanism that allows MBeans to generate and propagate events to other components. With this mechanism, it is possible to construct a publish-subscribe model to listen for and handle events.

The agent level provides facilities to control and manage resources. It is build upon the instrumentation level and includes a server object called MBeanServer. All MBean objects defined in the instrumentation level must be registered to MBeanServer to enable resource management, and all access to them is provided through the MBeanServer. This mechanism allows different management applications to monitor and tune a system. It also provides a remote access API that allows the distributed and remote management of resources.

The distributed services level provides the interfaces for implementing JMX managers.

This level defines management interfaces and components that can operate on agents or hierarchies of agents.

JMX provides a portable and scalable mechanism to manage different resources, but it only provides means for sensing and effecting purposes. It is not concerned with the self-management logic that correlates sensors with effector. There are JMX Management Console applications available in the market that are used by administrators to observe the system status and to invoke operations on the system. In this model, the control loop is created manually by the administrator. The StarMX framework makes self-management possible by enabling different JMX features and integrating them with other techniques to build control loops. StarMX interacts with JMX through its *agent level* facilities and provides the following services via a simple configuration. These services are elaborated in more detail in the following chapters.

- It provides a variety of mechanisms for accessing *MBeanServers* to interact with different types of *MBean* objects as sensors/effectors, addressing *FR3*
- It enables using *MonitorMBeans* as the monitoring component of control loops or for dispatching notification, addressing *FR4*
- It utilizes JMX *Notifications* for activating control loops and event dispatching, addressing *FR4* and *FR6*
- It supports remote access to MBeanServers, which enables remote application management, addressing *NFR2*

3.3.2 Policy Engines

As discussed in Chapter 2, there are several research projects that encourage using policies for describing adaptation logic. Reusable policy/rule engines, which parse and execute policies, facilitate using this technique for defining self-managing requirements and addressing self-* properties. Currently, there are many commercial and open-source policy engines used in academic and industrial projects. Apache Imperius [4] is an open source policy engine, which provides an object-oriented implementation of the CIM-Simplified Policy Language (CIM-SPL) [24]. CIM-SPL is a policy language designed for managing computing resources using Common Information Model constructs. JBoss Drools [46] is also a well-known open source rule engine, which provides a unified and integrated platform for rules, workflow, and event processing.

Regarding the diversity of policy languages and engines, StarMX is equipped with the capability to collaborate with different policy engines in an abstract manner via an *adapter* interface. This mechanism eliminates the dependency on any particular policy engine. It also allows the developer to choose his/her favorite engine and integrate it with the framework. The details of this feature will be explained in the next chapter. It should be noted that the framework is not limited to policies for defining management or adaptation logic: using Java classes is supported for this purpose as well.

3.4 High-level Architecture

A global view of the framework and its interactions with external entities like JMX and policy engines was presented in the previous section. Next, we look at the internal architecture of the framework from a high-level viewpoint. As depicted in Figure 3.3, it consists of two main elements: *Execution Engine* and a set of *Services*.

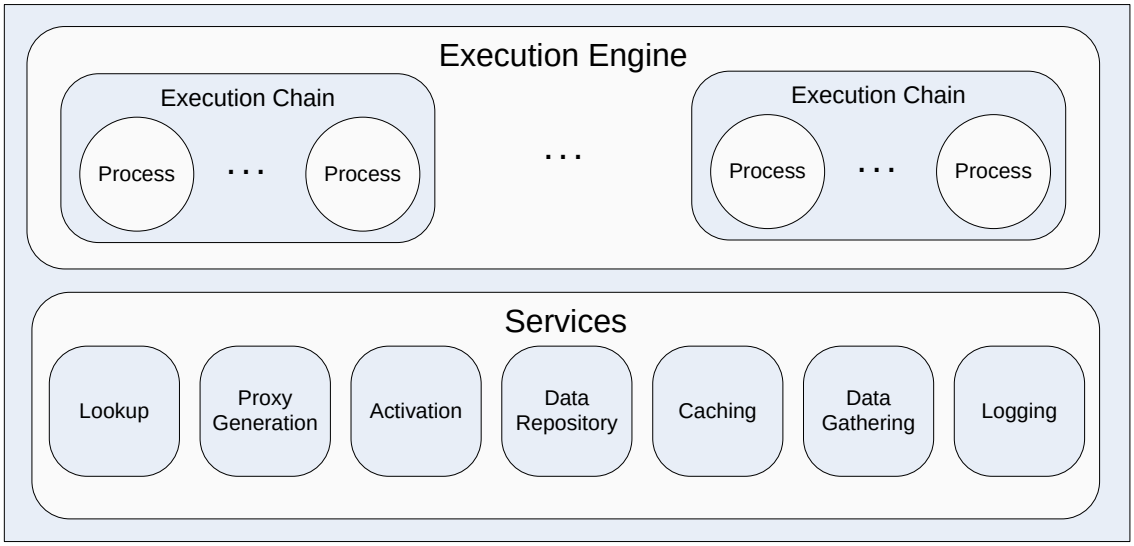

Figure 3.3: StarMX High-level Static Architecture

The Execution Engine module is the core of the system that automates self-managing operations. The two key components of the execution engine are *Execution Chain* and *Process*. Execution chains represent management control loops and can be considered as autonomic managers. They address all of our requirements regarding control loops (*FR1* and *FR2*). As shown in the figure, the system may be configured to have several execution

chains such that each one deals with a different self-managing concern. For example, it is reasonable to have one execution chain for each self-* property (e.g. self-optimizing) or one execution chain for each important manageable resource, or a combination of both approaches. Different functions of the control loop are defined by means of process components. They contain management logic and can be specified by policies. They use sensors and effectors to perform their jobs. This approach separates management logic from the business logic, thereby supporting the *separation-of-concerns* principle (*FR7*).

The role of execution engine is to regulate the control loops and to activate them based on their configured properties to deal with the runtime situations. Framework configuration is defined in an XML file containing the information on processes, policies, and control loops used by the execution engine at runtime. The behavior of execution engine demonstrates the *inversion-of-control* property of the framework. Processes that define management logic are framework *slots* provided by the developer.

The set of services provided in the service layer is designed to address our other requirements described in Section 3.2, and to provide some nice-to-have facilities. Of course this set can be enriched by providing more services in the future. However, these services are utilized by the execution engine and are available to the self-managing application developer through a set of APIs.

- *Lookup*: Enables access to sensor/effector components within control loops (*FR3*)
- *Proxy Generation*: Creates a proxy for an MBean type sensor/effector, so as to represent it as a simple object
- *Activation*: Provides mechanisms for activating control loops at appropriate times (*FR4*)
- *Data Repository*: Provides facilities for control loops to store data for keeping track of the history of observations and decisions, and to share data with other control loops (*FR5* and *FR6*)
- *Caching*: Improves the speed of access to sensor/effector components by a caching mechanism (*NFR1*)
- *Data Gathering*: Collects some statistical data about the runtime behavior of the control loops
- *Logging*: Keeps the records of different events happened in the framework at runtime, in a log file

3.5 Summary

This chapter presented an overview of the proposed software framework for supporting dynamic self-management. The framework is designed based on a set of requirements to facilitate developing control loops and autonomic managers. It is based on standards and supports the separation-of-concerns principle to enable separation of adaptation from the application logic. The framework permits the developer to use JMX and policies to create a management environment. The high-level architecture of the framework includes an execution engine, which runs and governs control loops, and a set of services to support the execution engine runtime functions. More details about the architectural components, provided services, and the framework runtime behavior are presented in the next chapter.

Chapter 4

Framework Architecture

This chapter explains the StarMX architecture, including its components and their relationships and runtime interactions. Several design patterns like proxy and chain-of-responsibility have been used to facilitate the interaction between different components. The first section discusses the execution engine modules and its internal design. The next section describes the services that support the framework behavior, and the last section shows how the framework behaves at runtime and enables dynamic adaptation.

4.1 Execution Engine

Execution Engine is the core module of the framework that automates self-managing operations. It executes the management logic defined by the application developer to adapt the system with its current situation using the *services* provided in the service layer. In other words, it enables the control loops to perform their jobs. In order to address different aspects of self-management, several instances of control loops might be needed. For example, one control loop is designed to address self-configuring, and another is designed to address self-healing. In another approach, one control loop is designed to address all self-* properties for each individual important resource. In this framework, each instance of the control loop is represented by one execution chain component, and execution engine is able to handle several instances of these components at runtime. In the following, we explore the architecture of execution chains.

4.1.1 Execution Chain Architecture

Figure 4.1 displays the architecture of an execution chain. This architecture model aims to provide a flexible approach for the developer to construct a control loop with access to the sensors and effectors. The wide arrows in the picture show the direction of interactions and the black thin arrows reflect the flow of control. We describe this architectural model in terms of its *components*, *connections*, and *semantic*.

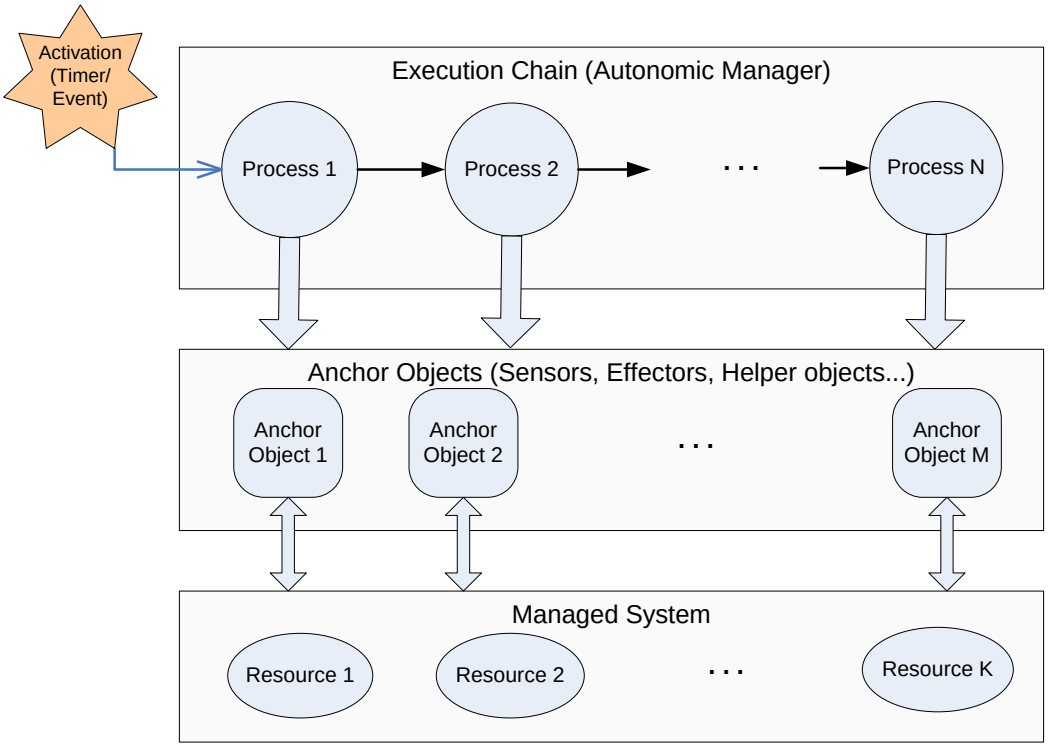

Figure 4.1: Execution Chain Architecture

Components

The components involved in this design include: one *execution chain*, a set of *processes*, a set of *anchor objects*, and an *activation service*.

- **Execution chain**: This is the core component of the model and represents a control loop.

- **Process**: This is an executable entity and the building block of execution chains. Each process may perform a single function or a group of functions of the control loop, such as monitoring, analyzing, planning, and executing. Each execution chain contains a sequence of one or more processes.

- **Anchor object**: Each process needs a collection of anchor objects to perform its task. These objects can either be *management endpoints* (sensors and effectors) of the underlying resources or *helper* objects that provide some services. The required set of anchor objects for each process and their lookup information are defined in the framework configuration file, described in the next chapter. Each anchor object can be used by different processes. Hence, these objects can be accessed concurrently by multiple processes. To improve performance, the framework does not provide any concurrency control mechanism for these objects, and it is left to the developer to handle this issue properly.

- **Activation service**: Each execution chain is associated with an activation service, which enables its execution at appropriate times. It can be a timer-based or an event-based service.

Note that managed system resources are not considered as components of this architecture, since they are visible only to the anchor objects, and not to the processes.

Connections

The connections among the components of this model are:

- **Execution chain - Process**: The execution chain maintains an ordered list of its processes through an *aggregation* relationship. The design of StarMX allows reusing one process in multiple execution chains. For example, if the process performs a reusable planning logic, it can be used by several execution chains.

- **Process - Process**: The processes in an execution chain are connected based on the *Chain-of-Responsibility* design pattern [28]. This pattern allows execution of a sequence of process which is useful for realizing the MAPE loop functions, since they should be performed in order.

- **Process - Anchor object**: Each process relies on its anchor objects' interfaces for interaction with them. There is no interaction between an anchor object and a process.

Other facilities have been designed to support data communication among processes. Processes can exchange data with each other through provided data repositories or shared memories. These mechanisms can also be used to exchange data based on a *Blackboard* architectural style. Moreover, there is no direct interaction or connection between two execution chains, but there are means that enable building a *publish-subscribe* model among execution chains to allow the broadcasting and handling of an event. A more detailed discussion on these techniques is available in Section 4.2.

Semantics

At runtime, an execution chain is activated by an event or at fixed time intervals. Upon activation, the involved processes are executed in order. The required set of anchor objects for each process is prepared and injected into the process right before its executing. During execution, a process uses its anchor objects to send or receive data to/from the managed system. The execution is terminated once the last process finishes its execution.

4.1.2 Realization Scenarios

This section presents how a developer can realize different components of execution engine, including: anchor objects, processes, execution chain, and activation service, to construct a control loop.

Anchor objects are used only by processes, and their interfaces are transparent to the framework itself. Therefore, StarMX does not enforce any constraint on the design or implementation of these objects. For example, they do not need to implement a particular interface. Instead, StarMX supports standard forms of access to these objects, and an anchor object can be presented by the following techniques:

- *MBean* or *MXBean*: These are application management interfaces in the JMX architecture. The application developer may implement application-specific sensors and effectors through a set of MBeans, or use the MBeans provided by other frameworks or the Java EE application server. MXBeans are special types of MBeans offered by JMX used to manage different JVM resources like CPU, threads, and memory.

- *JavaBean*: An anchor object can also be a simple Java object that provides some utility services to the process or performs management operations. It can be instantiated either directly by the execution engine or by a factory class, provided by

the developer, using the *Factory-Method* [28] design pattern. The factory method approach allows the developer to prepare a management interface around any kind of resource and present it to the framework as a simple object. This technique is useful to integrate StarMX with legacy systems, and it works as an alternative to JMX. For example, the developer can create a factory that returns an object that interacts with a resource using Java Native Interfaces (JNI).

The internal logic of a process is also defined by the developer, and it is totally transparent to the framework. There are two approaches for defining a process: *descriptively* using a *policy language*, or *programmatically* using the *Java language*. Using a policy language to defines the self-* requirements in form of condition-action policies is a common technique. The choice of policy languages is left to the developer, and StarMX is capable of working with external policy engines. It employs an *Adapter* design pattern [28] to abstract the interaction with different engines. Figure 4.2 illustrates the static relationships among execution chain, process, and external policy engine components. As shown in the picture, an implementation of *PolicyAdapter* must be provided by the developer to enable this interaction. Currently, StarMX is equipped with adapters for the Imperius framework [4] and the IBM ABLE rule engine [40]. The user can also provide an adapter class for other policy engines.

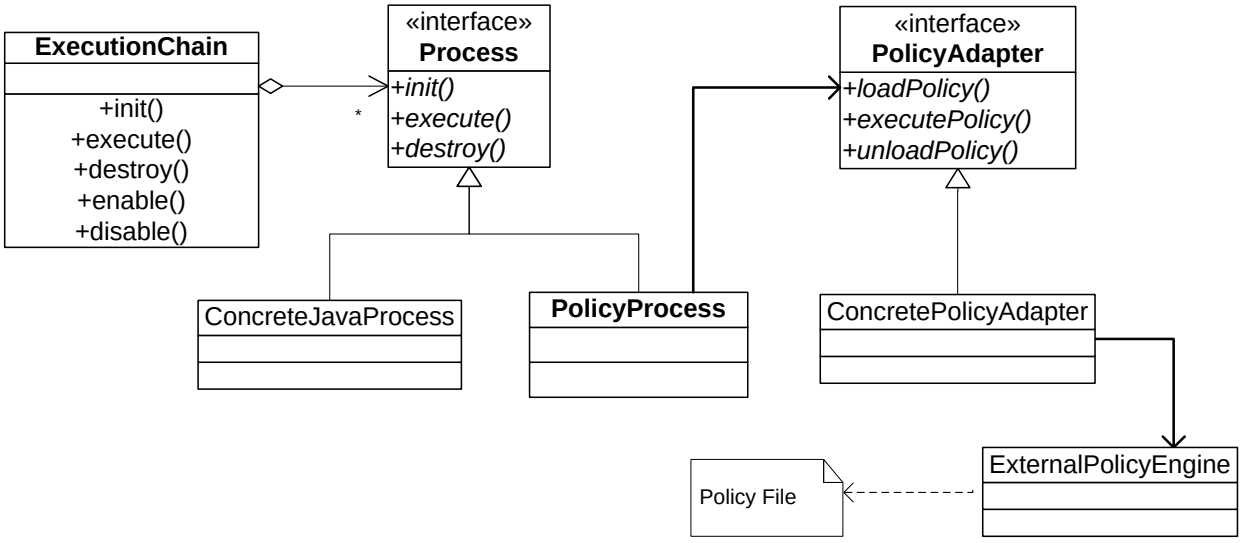

Figure 4.2: Process Classes and Integration with External Policy Engines

The implementation of an adapter class is simple: it basically needs to forward method

calls to the appropriate methods of the policy engine. The *loadPolicy* method is invoked only once by the framework for each policy at startup. It is responsible for performing initialization tasks like compiling and parsing the policy script. The *executePolicy* method is invoked at runtime whenever the corresponding execution chain is activated. This method should call the related policy engine method in turn. The *unloadPolicy* method is called at the shutdown phase to perform clean up tasks, if needed.

Researchers argue about the efficiency of policies for addressing the self-managing requirements. Huebscher *et al.* refer to the conflicts that may occur among policies at runtime [39]. The limitations of policy languages with respect to their constructs and syntax also impose constraints on the specification of the management logic. For example, it is hard to find a policy language that supports for-loop or nested if-else conditions, as needed for implementing the self-managing requirements. Moreover, although describing management logic as policies is fairly simple, it may become a tedious task in complex systems, and a high level of expertise is required to fine-tune the policies [15].

For the above reasons, StarMX also allows implementing a process using the Java language to benefit from all features of the programming language in the specification of the management requirements. In this case, the management logic is developed by implementing the *org.starmx.core.Process* interface, shown in Figure 4.2. The methods in this interface are self-descriptive, and they handle the process life cycle. The *init* and *destroy* methods are invoked to provide a chance for initialization and cleanup activities at the startup and shutdown phases respectively, and *execute* is called whenever the process must be executed as a result of activating the execution chain. The management logic, which contains monitoring, analyzing, planning, and executing functions, is implemented in this method.

Furthermore, the composition of processes in execution chains to build control loops can be either *static* or *dynamic*. In static mode, the chain of processes is defined in the configuration file, while in dynamic mode, several execution chains may form a bigger control loop on-the-fly. In this case, the execution of a process may result in the activation of another execution chain by sending an event. More details on this model will be provided in Section 4.2 in introducing the *activation* service. Activation service properties such as the time intervals or the event that activates the execution chain are also described in the configuration file. Configuring the framework is explained in Section 5.2.

Realizing IBM's architecture

The execution chain architecture (Figure 4.1) provides a flexible mechanism to design a control loop (autonomic manager), as presented by IBM (Figure 4.3).

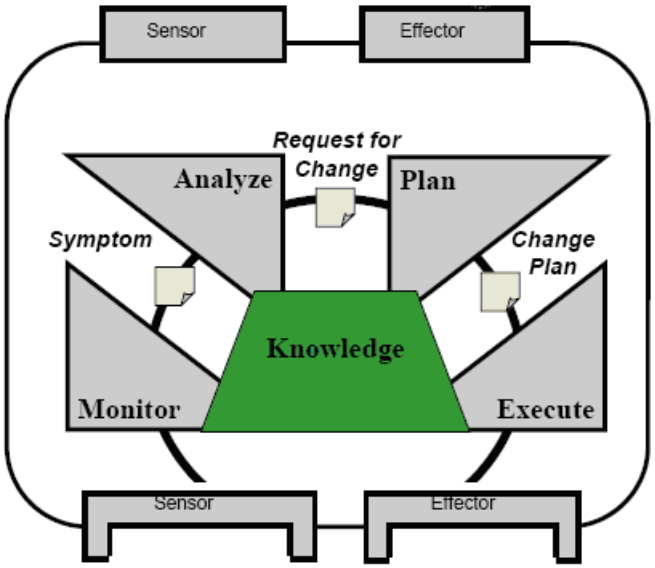

Figure 4.3: Autonomic Manager Architecture presented by IBM

- In StarMX, the control loop can be created by a single process or an arbitrary number of chained processes, and they map very well to the four MAPE loop entities. Our framework provides flexibility in the design of control loops based on the complexity of the problem. For example, in a simple scenario, all four functions can be merged into one process presented as a policy; or, in a complex case, analyzing and planning can be split into several processes.

- StarMX promotes the notion of anchor objects which are more generic than sensors and effectors. An anchor object may provide either or both sensing and effecting functions or some utility services.

- The relationship between processes and anchor objects is flexible and each process can use several different anchor objects, whereas in IBM's architecture, there is no interaction between the analyzing and planning modules and the sensors and effectors.

4.2 Runtime Services

A set of services are provided by internal framework components to support the runtime operation of the execution engine, and they are helpful in enabling self-managing behavior in target systems (Figure 3.3). Some of these services are used by the framework internally and some are available to the user via the framework external interfaces and APIs. This section describes the main purpose of each service and how it contributes to the framework behavior.

Lookup

This service provides mechanisms to access the anchor objects of the processes. It is responsible for finding and preparing an anchor object instance based on its type. In the case of MBean, it supports several approaches for accessing the MBeanServers and locating the specified MBean object, such as JNDI lookup or JMX Remote API. In the case of JavaBean, it either instantiates the object by itself or contacts a factory to perform this task. This service is invoked by the execution chain prior to the execution of each process to locate the required set of anchor objects for that process.

Proxy Generation

The objective of this service is to create a proxy object [28] dynamically when the located anchor object points to an MBean. The purpose of creating a proxy is to represent the MBean as a simple Java object with an interface. The proxy contacts the MBean through the MBeanServer and hides all internal details related to the MBean method calls. The proxy generation happens at lookup time.

Activation

This service supports different techniques for triggering execution chains for execution. The two enabled methods are:

- *Timer-based*: This is a polling model, which allows the system to continuously check its internal state and to take appropriate actions in response. In this approach, the execution chain is triggered at fixed time intervals (e.g. every 10 seconds).

- *Event-based*: This is a push model, which activates the adaptation process when a certain event happens. In this case, the execution chain subscribes to receive a particular event. The event can be published by a component in the managed system. By receiving the event, the execution chain will be executed. The JMX notification architecture is utilized for this purpose. Based on the JMX specification, MBeans can create a notification object as an event, customize its properties, and send it. Several listener objects can register to receive the event. In our framework, an execution chain registers itself to handle the event.

The framework provides detailed configuration properties for the mentioned activation mechanisms, as discussed in Section 5.2. Regarding the event-based technique, there are more facilities that can be utilized by the developer. The occurrence of an event can also lead to the activation of more than one execution chain. For example, several execution chains can register themselves to be activated by the same event. Moreover, the framework offers two approaches for receiving and handling an event by an execution chain:

- *Synchronous*: This approach blocks the event publisher until all processes in the activated execution chain are executed. This mechanism enables the dynamic construction of control loops through sending an event from a process and handling the event synchronously by another execution chain. As shown in Figure 4.4, during the execution of P12 in ExecutionChain1, an event is sent, which results in the activation of ExecutionChain2. At this moment, P12 is blocked until the execution of ExecutionChain2 is completed. This results in creating a control loop containing P11, P12, P21, P22, and P13 processes. Of course, depending on the moment that the event is raised, P21 and P22 may be executed right at the beginning or in the middle of P12's execution.

- *Asynchronous*: This technique allows executing the activated execution chain in a separate thread, and therefore not blocking the sender. In this case, the second execution chain is executed concurrently with the first execution chain.

Data Repository

The objective of this service is to enable processes, to store data for future usages or to exchange data with other processes. The data may include the history of observations or decisions. This service also enables data communication among execution chains and processes. This facility is comparable to the knowledge part of the autonomic managers (in

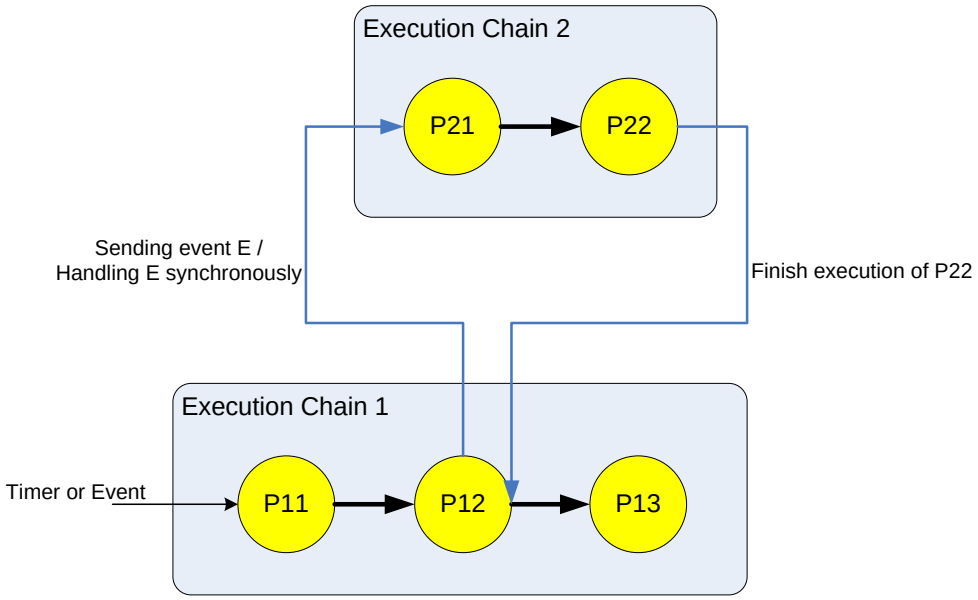

Figure 4.4: Dynamic Control Loop Construction: An Example

Figure 4.3). The data repositories are currently provided through the working memories with different scopes. The framework offers three types of repositories to address different communication needs among processes. These scopes are in alignment with the available memory scopes in the J2EE architecture in the web tier. Furthermore, the design of these repositories enables implementing them as a persistent data store like a file or database.

- *StarMXScope*: A global scope shared among all processes. Any information stored in this scope is always accessible to all processes. It is created when the framework is initialized and is destroyed when the framework shuts down. This scope is similar to the *ServletContext* memory in J2EE applications, which is a shared memory among all web tier components.

- *PolicyScope*: A private scope related to a policy-based process. It is created when the process is deployed and is destroyed when the process is undeployed. It allows creating stateful policies. Note that Java-based processes are stateful objects and they do not need a separate scope to store runtime data. This scope corresponds to the *HttpSession* memory in J2EE applications, which contains the user's session data.

- *ExecutionScope*: A scope associated with an active instance of the execution chain.

Any information in this scope is only accessible by the subsequent processes in the chain. It is created at the activation time of an execution chain and is destroyed at the end of the execution. A nested execution chain (an execution chain activated by another chain, see Figure 4.4) uses the existing ExecutionScope associated with its parent. This scope resembles the facility provided by the *ServletRequest* object in the J2EE web-based system, which allows a web component to send some data to the forwarded Servlet or JSP.

Another interesting feature of the designed repositories is the support for realizing a *Blackboard* architectural style through subscribing different listeners to the repository. Such listeners are notified of data manipulation operations like addition, update, or removal.

In addition, this facility can be used by the developer to resolve conflicts between different execution chains. The concurrent execution of multiple execution chains may result in some conflicts as they may contradict each other in the adaptation decisions. Processes can use data repositories to negotiate with each other by reading and writing data into them, and the can proceed based on the satisfaction of some specific conditions in the repository.

Caching

This service aims to improve the performance of the lookup service by holding references to previously accessed anchor objects. It saves proxy generation or object instantiation time in the next access. This service is able to detect the registration and deregistration of MBeans and to invalidate the cache at appropriate times.

Data Gathering

This facility collects statistical data about the execution of each process and execution chain. This data currently includes execution count, failure count, and average execution time, but the service can be extended, using its API, to gather more information. The data will be made available to the processes, helping them to adjust their behavior. The collected data can also be logged into a file for administrative use.

Logging

The purpose of this mechanism is to provide a configurable logging facility to record different framework events into a log file. The log file can be analyzed by the administrator later.

This service is designed based on the Log4J [5] framework. Log4j is a very popular logging framework used in many applications. It allows creating objects, called *logger*s, which log the messages to output streams. Each log message has a *log-level* attribute (*e.g.* DEBUG, INFO, WARNING, ERROR, etc.), and the logger can be configured to only log messages with a log-level equal to or higher than a particular log-level. Each logger is associated with objects, called *appender*s, which physically write messages to output streams. Each appender writes the log messages to a different output stream (*e.g.* console or log file). Hence, when a message is going to be logged by a logger, it is sent to all associated appenders to be written to all destinations. Moreover, the layout and format of log messages can be customized to include properties like timestamp, thread name, and logger name.

4.3 Runtime Behavior

The runtime behavior of the framework is divided into three phases: *startup*, *operation*, and *shut down*. At startup, StarMX prepares the environment for the optimized operation of the execution chains. All services are initialized, and processes and execution chains are deployed based on specified properties in the configuration file. In this phase, each execution chain is deployed such that it is ready to be activated at the appropriate times. This is done by scheduling a timer or subscribing for an event (or a class of events).

Once the framework has successfully started, it is ready to operate. In this phase, the execution chains are invoked by their own activators for execution. Upon activation of an execution chain, all processes in the execution chain are invoked in order. For each process, first, its required set of anchor objects are prepared through the *lookup*, *proxy generation*, and *caching* services. Next, the process is called with the anchor objects, provided as arguments. The process invokes the anchor objects' methods to obtain data from or send commands to the application. The sequence diagram, depicted in Figure 4.5, displays the execution steps at runtime, described above. In the diagram, the green boxes are framework classes or interfaces, and the yellow boxes are non-framework objects developed by the user.

If a policy language is used to define a process, the related policy engine is invoked through its adapter class to execute the policy with the provided anchor objects. A policy-based process is a specialized class of process, which is internally represented by a *PolicyProcess* object. As illustrated in Figure 4.2, it is a concrete implementation of the *Process* interface, which interacts with a policy engine through its adapter. Figure 4.6 shows the the execution sequence when a policy-based process is involved.

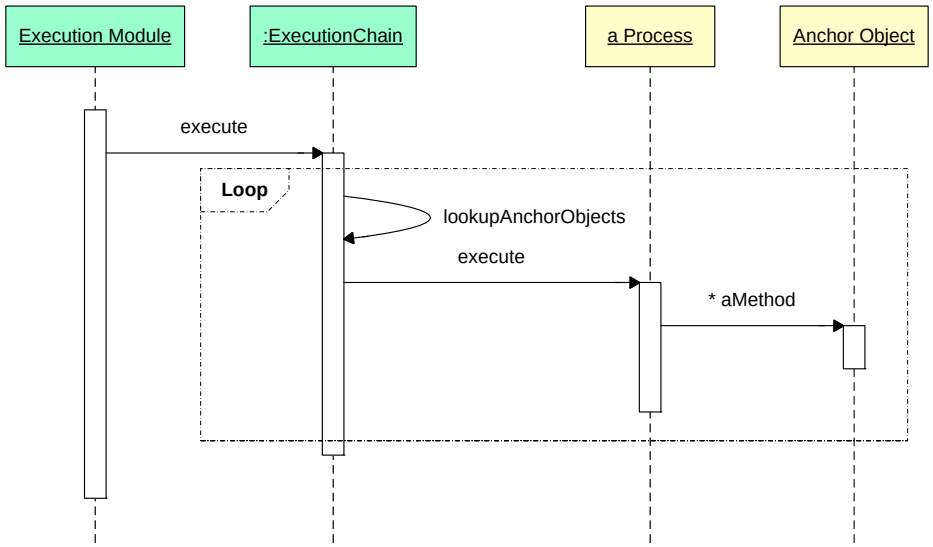

Figure 4.5: Process Execution Sequence Diagram

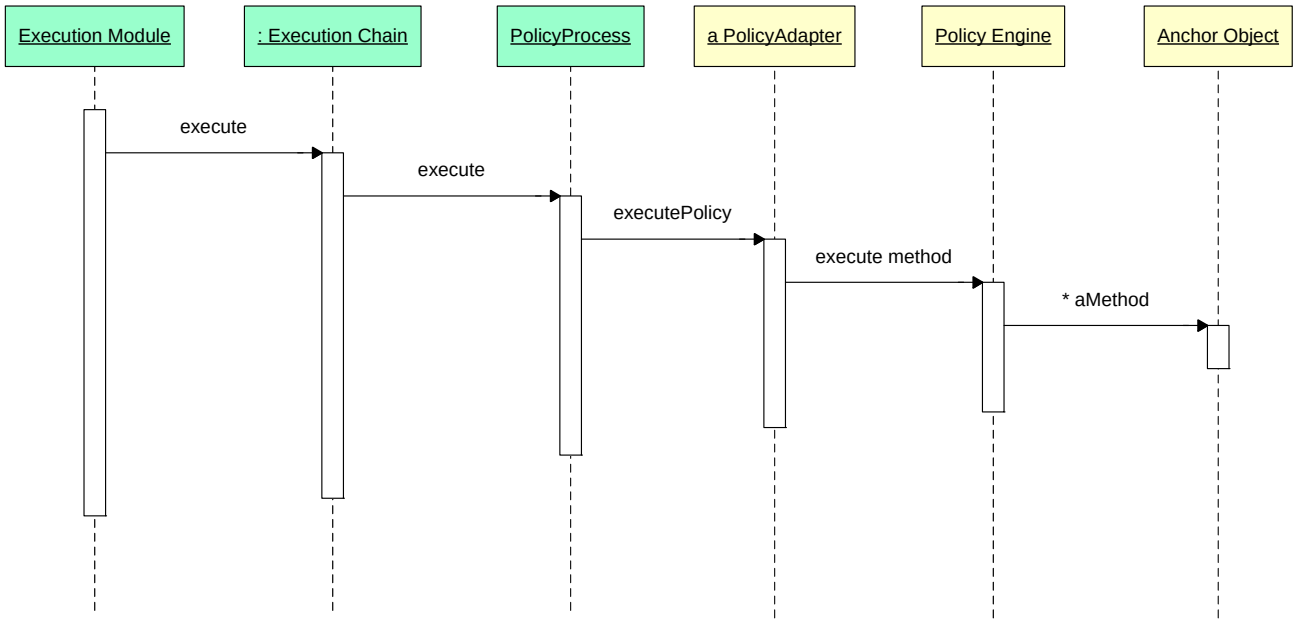

Figure 4.6: Policy-based Process Execution Steps

Furthermore, when the anchor object is an MBean, it is represented by a proxy to be used by the processes. Figure 4.7 shows the sequence of invocations that will happen as the result of calling an MBean proxy from a process. In the diagram, the MBean related classes are shown in yellow and the managed resource is displayed in blue.

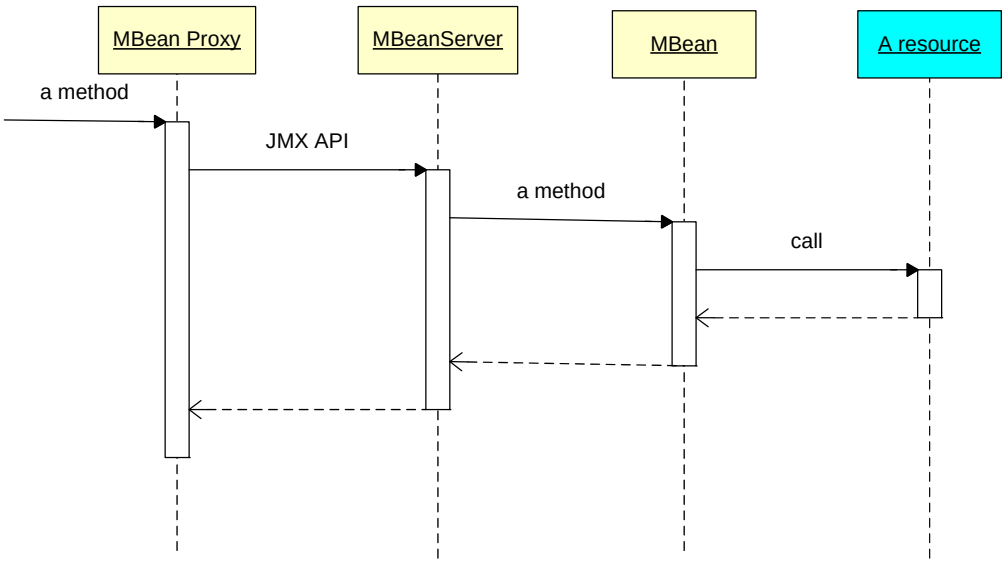

Figure 4.7: MBean Type Anchor Object Invocation From a Process

During the execution, a process can also access the provided memory scopes (data repositories) to read or store data, or to share it with other processes. Access to the memory scopes is provided via the *ExecutionContext* interface. This is a framework object that is created each time an execution chain is activated, and it is passed to all processes in the chain. As illustrated in Figure 4.8, it provides access to different memory scopes, and to the information of the event that has activated the execution chain. It also allows a process to stop the execution of the chain by calling the *noExecuteNext* method. As the result of invoking this method, the subsequent processes in the chain will not be executed. This feature is useful, if a process finds a situation in the managed system that the control loop cannot deal with, and it is better to stop the execution early. Note that while ExecutionContext is available to Java-based processes programmatically, a policy engine adapter should pass it to the policy engine as an anchor object to make it available to a policy. In this case, the policy script can use the ExecutionContext object like any other anchor object.

```
public interface ExecutionContext {

    public TimerEvent getTimerEvent();
    public Notification getNotification();

    public Scope getStarMXScope();
    public Scope getPolicyScope();
    public Scope getExecutionScope();

    public void noExecuteNext();
    ...
}
```

Figure 4.8: ExecutionContext Interface

The StarMX framework is designed to serve the best performance at runtime by reducing the amount of time spent in the framework during execution. However, the total execution time of the control loops can be affected by other external factors such as the time consumed in the anchor objects or in the external policy engine. Section 6.3.3 discusses this issue in more details.

Finally, at the shut down phase, the framework undeploys the execution chains and all processes and stops working.

4.4 Summary

This chapter describes the architectural details of the framework and its novel features. In brief, it allows to create closed control loops in the form of execution chains, which compose several processes. Each process deals with part of the self-managing issues and interacts with the managed system through a set of anchor objects. Several services are provided to support this behavior including: lookup, proxy generation, activation, data repository, caching, data gathering, and logging. At runtime, upon arrival of an event or based on a scheduled timer, the processes of an execution chain are executed in order and each one is provided with the required set of anchor objects. The user is responsible for developing processes, their correlation, and their required anchor objects to construct control loops and enable self-managing behavior. The next chapter will explain several models for application management and how to realize these models using StarMX.

Chapter 5

Developing Self-Managing Application

Developing an application with self-managing capabilities, or converting a legacy system to an autonomic system is challenging and should be carried out according to a well-defined process. Different approaches and models have been proposed by researchers to address self-* properties in computing systems (*e.g.* [41, 52, 77, 84]). This chapter describes how a self-managing application can be developed using the proposed framework. The first section outlines several usage scenarios for building adaptive and self-managing systems and shows how they can be realized by this framework. The next section presents the steps required to be followed for building such systems, and the last section explains the framework configuration properties.

5.1 Development Steps

Creating self-managing software systems, regardless of the techniques or models that are used for dynamic problem detection and resolution, requires a systematic approach that helps the developer to proceed step by step to achieve the final result. In an abstract comparison, it is similar to a software development methodology or process, which starts from the requirement specification and ends at deployment. Based on our experience in enabling systems with adaptation behavior [74, 75], we defined a five-step process to build self-managing systems using StarMX. Figure 5.1 demonstrates these steps and their input and output artifacts. Arrows show the flow of data and are annotated with the related artifacts.

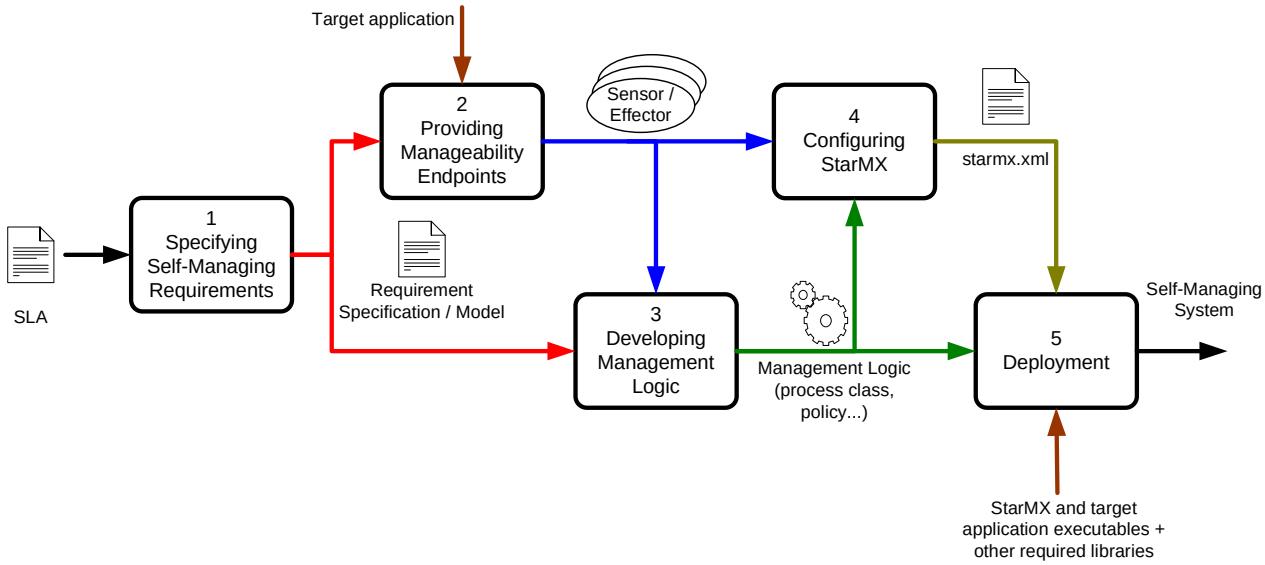

Figure 5.1: Required Steps to Develop a Self-Managing Software System

- **Step1: Specifying Self-Managing Requirements** - Self-managing solutions mostly focus on non-functional and QoS (Quality of Service) requirements. Therefore, these solutions deal with performance, security, reliability or other quality factors. These requirements are also referred to as self-* properties as discussed in [76]. The goal of this step is to clearly specify or model these requirements and the expected objectives for the target system.

  – *Inputs*: Service Level Agreement (SLA), which is decomposed into Service Level Objectives (SLO). Other non-functional requirements can also be considered as the input to this phase.

  – *Outputs*: Self-managing requirement specification

  The way that these requirements should be specified or modeled is an open research area and is not within the scope of this work. For example, goal-oriented modeling is one of the techniques for engineering adaptation requirements, cited by different sources [54, 64, 72]. Moreover, the StarMX framework does not impose any constraint on working with any aspect of self-managing properties.

- **Step1: Providing Manageability Endpoints** - Manageability endpoints (sensors and effectors) are the gateways for interfacing with a resource for management pur-

poses. Based on the self-managing requirements specified before and the resources designated to be managed, the required set of sensors and effectors are identified. These objects should be designed and instrumented into the target application in this phase. As discussed earlier, these can de designed as JMX MBeans or as simple Java objects.

  - *Inputs*: Self-managing requirement specification and the target system
  - *Outputs*: A set of sensors and effectors

  As discussed in Chapter 2, it is also possible to use Web-Services as management interfaces of a system. Section 5.1.1 shows how this can be achieved using the StarMX framework.

- **Step3: Developing Management Logic** - This is the core part of the process for building a self-managing system. To accomplish this part, an adaptation solution must be designed, for example based on the techniques discussed in Chapter 2. The management logic is developed as a set of Java-based processes, policies, and anchor objects that form control loops. We elaborate the design and implementation of three approaches as examples, namely: Policy-based adaptation, GAAM [77], and layered adaptation [52], in Section 5.1.1.

  - *Inputs*: Self-managing requirement specification and the set of sensors and effectors
  - *Outputs*: Developed management logic as a set of policies, Java-based process classes, and anchor objects

- **Step4: Configuring StarMX** - The next step is to define the construction of control loops and their properties for the framework. StarMX uses an XML configuration file containing information of the anchor objects, processes, and execution chains. For each anchor object, its lookup information is specified, and if it is an MBean, information for accessing the associated MBeanServer is also defined. For each process, its type information and the needed anchor objects are defined, and for an execution chain, the order of its processes along with the activation mechanism are declared. More information about this topic is presented in Section 5.2.

  - *Inputs*: The set of sensors and effectors, policies, Java-based processes, and anchor objects
  - *Outputs*: Framework configuration file (*starmx.xml*)

- **Step5: Deployment** - The last step is to integrate the framework with the target system, deploy it, and test whether the created self-managing system performs as what is expected.
  - *Inputs*: The StarMX and target system executable, other required libraries or frameworks (*e.g.* policy engine), and the outputs of step 3 and step 4
  - *Outputs*: Self-managing system

  There are two deployment options: *local* and *remote*. In local deployment, StarMX is deployed on the same server and the same JVM that the target system executes on. It must be started and shut down properly (by the target system). The best practise to start and shut down the framework is to do so at the application startup and shut down stages. In remote deployment, StarMX is deployed on a different server and started as a separate application that manages the target system. This deployment mode is enabled using JMX features for remote access to the MBeans.

  Selecting the appropriate deployment option in a real environment is a trade-off among the performance overhead of each approach, ease of deployment, and other domain-specific concerns. For example, in local mode, the framework consumes target system resources (e.g. CPU and memory), while in remote mode, only the sensing and effecting operations use the system resources. This is because in remote mode, the MBeans are collocated with the application objects in the same JVM. Moreover, remote access to the MBeans is not as fast as local access, and is therefore not a good approach for time-sensitive adaptations. On the other hand, in remote deployment, one running instance of StarMX can be used to manage several systems and distributed resources by proper configuration of the framework.

5.1.1 Sample Scenarios

This section describes several scenarios where StarMX can be utilized to address self-managing concerns. They are selected from the solutions discussed in the literature to reveal how different capabilities of the framework are used to enable autonomic properties. Other approaches discussed in Chapter 2 are also possible candidates for this section.

Policy-based Model

Using action policies is one of the simple models to enable self-management. Each policy takes an action if a condition is met. This mechanism creates a closed control loop

consisting of monitoring and executing functions. In this approach, policies are defined in separate files, and they are introduced to the framework (in the configuration file) as separate processes that contribute to the control loop. Since the policies are actually executed by a policy engine, an adapter class should also be created for that engine by implementing the *PolicyAdapter* interface to delegate the execution of a policy to the engine (See Section 4.1.2).

Moreover, policies can be used for different control loop activities like monitoring, analyzing, planning, and executing. For example, we can have monitoring policies whose action is to send an event when a particular situation is observed. In this case, a group of policies are chained together to form the control loop (implemented as the processes of an execution chain) such that each policy is responsible for one or more aspects of the management.

Goal-Attribute-Action Model

GAAM is presented in [77], and we provided a brief overview of this approach in Section 2.4. Goals are system objectives that should always be satisfied. More precisely, each goal is defined based on some conditions and some suggested actions. For example, a performance goal states that the response time must be lower than 1 second; and if it is greater than 1 second, one of the following actions should be performed: tuning parameter A, changing parameter B, switching from algorithm X to Y, disabling service C, etc. To avoid conflicts among the goals and to achieve the best result, a voter module exists in the system that analyzes the decisions made by the goals. The voter coordinates the decision making process and generates the action plan to be executed.

This model can be implemented in different ways using StarMX features. The following describes one possible approach which is used in the development of our case study (described in the next chapter). In this design, each goal is implemented as a Java objects with an active/inactive state, which generates a list of preferred actions upon request. Voter is a StarMX process which traverses a list of goal objects, collects the preferred actions of activated goals, and decides which actions must be executed and in which order. The internal algorithm of the voter could be anything and is not the matter of discussion here. Goal evaluation is performed by means of policies, checking system attributes, and activating/deactivating the goal.

Several execution chains (control loops) are defined to be executed at different time intervals. Each execution chain is comprised of a set of goal evaluation policies, which perform both monitoring and analyzing tasks, one voting process for deciding and plan-

ning, and one executing process to execute the voter's result. Each policy evaluates some conditions using a set of sensors, and activates or deactivates one specific goal. The goal object and sensors are the anchor objects of the policy. At runtime, the policies first determine the statuse of the goals, then the voting process is executed to produce the preferred action set, and finally the executing process translates the action plan to some method invocations on the effector objects.

Multi-layered Management Model

A multi-layered model for managing systems is presented in [52]. In this model, system management activities are categorized in a hierarchy of layers. Lower layers deal with management details and provide fast responses to events, and higher layers perform further analysis and provide solutions to more complicated situations. Each layer first tries to resolve the problem based on its own knowledge, and sends an event to the next higher layer if it fails to resolve the problem. For example, a lower layer may try to resolve a performance problem by tuning system parameters. However, parameter tuning may prove ineffective in addressing the problem as time goes on. To resolve this issue more knowledge of the system components is required. Thus, the lower layer sends an event to the higher layer to deal with the problem. This scenario will be repeated until either the problem is resolved or a human administrator is informed.

To build this model with StarMX, the management layers of this model can be considered as different execution chains that interact with each other through the *event publishing* facility and handle events *asynchronously*. The source of events can be a system resource or an execution chain. The asynchronous event handling allows lower layers to respond to other events without blocking.

Using Web-Services for Application Management

JMX MBeans are one of the well-known standards for application instrumentation. As discussed in Section 2.1, Web Services Distributed Management (WSDM) [69] defines a standard for managing heterogenous resources via web-services.

To enable this technique in StarMX, it is required to present a web-service as an anchor object. It can then be used by policies or Java-based processes as a sensor or effector. An anchor object can be a simple Java object that is either instantiated by the framework or via a *factory* class. In this case, the developer can provide a factory class that reads the web service definition from its WSDL (Web Service Definition Language) file, and creates

a corresponding proxy object to represent the web-service as a simple Java object with an appropriate interface. The final step is to define the information of the factory class in the framework configuration (see Section 5.2.3).

In the future, the framework can be extended to support web services as anchor objects through generating a proxy object for a web service based on its WSDL information automatically, thereby eliminating the need for developing a factory class.

5.2 Framework Configuration

StarMX configuration is defined in the *starmx.xml* file. This is an XML language for defining the information of anchor objects, processes, and execution chains via XML tags. This file is read during the framework startup phase. A summary of the key configuration items is presented in Table 5.1. A detailed discussion of these items and their attributes is provided in the following sections. A complete example of this file is also available in Appendix A.

Table 5.1: StarMX Configurable Items

Item	**XML Tag**	**Purpose**
MBeanServer	*mbeanserver*	Access point to MBeans
MBean / MXBean	*mbean*	MBean and MXBean as an anchor object
JavaBean	*bean*	Simple Java object as an anchor object
Monitor MBean	*monitor-mbean*	A monitor with an event propagation facility
Process	*process*	A policy-based or Java-based process
Execution Chain	*execute*	An execution chain
Timer	*timer-info*	Timer-based activation for an execution chain
Event	*notification-info*	Event-based activation for an execution chain

5.2.1 MBeanServers

MBeanServer's information is used to provide access to MBeans, MXBeans, or Monitor MBeans. The following shows the syntax for defining an MBeansServer.

```
<mbeanserver id=""
             lookup-type="platform|jndi|jmx|find|factory" />
```

-*id*: A unique identifier for the MBeanServer, which is referenced by the MBeans.

-*lookup-type*: The technique used to create a connection to the MBeanServer. Depending on the specified value, more information should be declared.

```
Example: Creating an MBeanServer connection to the MBeanServer in JBoss5
          application server.
<mbeanserver id="jboss_ms" lookup-type="jmx" >
    <jmx-param service-url="service:jmx:rmi://localhost
                    /jndi/rmi://localhost:1090/jmxconnector" />
</mbeanserver>
```

5.2.2 MBeans and MXBeans

MBeans and MXBeans are the application instrumentation interfaces used by the processes. Both MBeans and MXBeans are defined by the *mbean* tag as shown below.

```
<mbean  id=""
        object-name=""
        interface="(optional)"
        mbeanserver=""  />
```

-*id*: The unique identifier for the MBean, referenced by the processes as an anchor object.

-*object-name*: The string representation of the MBean's ObjectName.

-*interface* (optional): The full class name of the MBean interface used in the proxy generation process. Note that the MBean may not implement the specified interface, but its methods' signatures must look like those of the interface.

-*mbeanserver*: The id attribute of the *mbeanserver* tag that the MBean is registered with.

```
Example:
<mbean id="runtimeMXbean" object-name="java.lang:type=Runtime" mbeanserver="ms1"
        interface="java.lang.management.RuntimeMXBean"/>

<mbean id="mbean1" object-name="starmx:name=mb1,type=xyz" mbeanserver="ms2" />
```

5.2.3 JavaBeans

Beans are simple Java classes used as anchor objects for the processes. Each bean is represented by a *bean* tag.

```
<bean id="" class="" factory-method="(optional)" />
```

-*id*: The unique identifier for the Bean, referenced by the processes as an anchor object.

-*class*: The full class name of the Bean. The class must have a default constructor if it is instantiated by the framework.

-*factory-method* (optional): Instead of direct instantiation, it is possible to use a factory. In this case, the *class* attribute is the factory class (rather than the bean class), and this attribute is the factory method name. The factory method signature must return an instance of the bean, and it can be either parameterless or a single-parameter method accepting the *id* as input. The factory class must have a default constructor unless the factory method is *static*.

5.2.4 Monitor MBeans

Monitor MBeans are special types of MBeans in JMX used to periodically observe an attribute in one or more MBeans and to send a notification if a certain condition is met. Their notifications are often used for the activation of execution chains. They can also be used as anchor objects.

```
<monitor-mbean  id=""
                observed-attribute=""
                granularity-period=""
                object-name=""
                mbeanserver="" >
    <counter-monitor .../> or
    <gauge-monitor .../>   or
    <string-monitor .../>

    <observed-object .../>
    ...
</monitor-mbean>
```

-*id*: A unique identifier for the MonitorMBean.

-*observed-attribute*: The name of the attribute to be observed.

-*granularity-period*: The time interval in milliseconds to observe the attribute.

-*object-name*: The ObjectName of this MBean. The monitor MBean is registered by the framework under this name in the specified MBeanServer .

-*mbeanserver*: The id attribute of the mbeanserver tag, which the MBean should be registered with.

Currently, three different types of monitors are provided by JMX (CounterMonitor, GaugeMonitor, and StringMonitor). The common information of all these monitors is defined in the *monitor-mbean* tag and the monitor specific attributes are declared by one of the *counter-monitor*, *gauge-monitor*, or *string-monitor* tags. It is necessary to use one and only one of these tags. The attributes of these tags correspond to the writable attributes of their MBeans, according to the JDK and JMX documentations.

The list of objects to be observed by the monitor is defined via the *observed-object* tag. Several observed-objects may be defined within the *monitor-mbean*. Each tag contains an *object-name* attribute, which represents a single MBean or a group of MBeans (if it is a pattern). If the ObjectName is a pattern, all available MBeans that match with the pattern are considered for observation.

```
<observed-object object-name="" />
```

Here are some examples:

```
<monitor-mbean  id="monitor1"
                observed-attribute="size" granularity-period="1000"
                mbeanserver="ms2" object-name="starmx:type=monitor,name=cm" >
    <counter-monitor init-threshold="100" notify="true" modulus="5"
                     offset="10" difference-mode="true"/>
    <observed-object object-name="starmx:type=Control,name=abc"/>
    <observed-object object-name="starmx:type=Control,name=xyz"/>
    <observed-object object-name="starmx:type=Control,name=def"/>
</monitor-mbean>

<monitor-mbean  id="monitor2"
                observed-attribute="size" granularity-period="1000"
                mbeanserver="ms2" object-name="starmx:type=monitor,name=gm" >
```

```
    <gauge-monitor high-threshold="100" low-threshold="10"
                   notify-high="true" notify-low="true" />
    <observed-object object-name="starmx:type=Control,name=*"/>
</monitor-mbean>
```

5.2.5 Processes

Processes are the placeholders of management logic and are chained together to support control loop functionalities. At execution time, they are provided with the required set of anchor objects to perform their jobs. Each process may be used in one or more execution chains.

```
<process id=""
         policy-type="(optional)"
         policy-file="(optional)"
         javaclass="(optional)">

    <object name="" ref="" />
    ...
</process>
```

-*id*: This is the unique identifier of the process.

-*policy-type* (optional): If the process is described by a policy language, this attribute shows the policy engine that is used to execute the policy. Based on the value of this attribute, StarMX identifies which policy adapter should be used. The value of this attribute is a string that identifies the corresponding policy adapter class. For example, if you use the XYZ policy engine, you may create the XYZPolicyAdapter class as well; in the configuration, you should use "xyz" as the value of the *policy-type* attribute and define the policy adapter class with the following property, as shown below.

```
<property name="starmx.policy.adapter.xyz">
    mypackage.XYZPolicyAdapter
</property>
```

-*policy-file* (optional): This attribute specifies the policy file name. If the *policy-type* is defined this attribute must also be declared.

-*javaclass* (optional): If the process is implemented with Java (instead of a policy language), its class name is defined by this attribute. Therefore, either the *javaclass* attribute or the *policy-type* and *policy-file* attributes should be defined .

-***object*** (optional): The anchor objects required for this process are defined by this tag. Each tag is related to one anchor object. The objects are chosen from the already defined MBeans and JavaBeans. The *name* attribute is the local name of that object in the process implementation. The *ref* attribute refers to the *id* of an *mbean* or *bean* element. This tag provides a mapping between the name of an object in a process and the id of the anchor object in the configuration.

```
Example:
<process id="policy1" policy-type="spl" policy-file="p1.spl">
    <object name="mb1" ref="myMBean1" />
    <object name="mb2" ref="myMBean2" />
</process>

<process id="sample-proc" javaclass="org.starmx.policy.SampleProcess">
    <object name="mb3" ref="myMBean3" />
</process>
```

5.2.6 Execution Chains

The execution chain's configuration is defined by the *execute* tag. It includes the list of processes and the activation mechanism information.

```
<execute name="(optional)"
         listener="(optional)" >

    <timer-info ... /> or
    <notification-info ... />

    <process .../> or
    <processref refid="" />
    ...
</execute>
```

-*name* (optional): A name for this execution chain

-*listener* (optional): The listener class name for this execution chain. The listener is invoked at the occurrence of different events during the execution chain life cycle.

The *timer-info* or *notification-info* tags are used to define how the execution chain must be activated.

-***timer-info***: This tag is used to configure the chain with a timer-based activation.

```
<timer-info interval=""
            unit="second|minute|hour|day|week|month"
            first-exec-time="HH:mm"
            first-exec-delay=""
/>
```

-*interval*: The time interval between subsequent executions.

-*unit*: The time unit of the interval attribute.

-*first-exec-time* (optional): This specifies the first time that the chain should be executed. It should be represented as HH:mm and is interpreted based on the behavior of the Java SimpleDateFormat class.

first-exec-delay (optional): This attribute can be used instead of *first-exec-time* and defines the amount of delay in seconds, before the first execution of the execution chain. Its minimum and default value is 1 second.

-***notification-info***: This is used to execute the chain upon the receipt of a Notification.

```
<notification-info  emitter-mbean=""
                    event-type=""
                    event-class=""
                    event-handling="synch|asynch"
/>
```

-*emitter-mbean*: The *id* of the *mbean* tag that sends the notification.

event-type (optional): The type attribute of the JMX Notification object. If specified, the notifications are filtered by the mentioned type.

event-class (optional): The class name of the notification object, which is a subclass of *javax.management.Notification*. If specified, only the notification objects with this class (or its subclasses) will activate the chain.

event-handling (optional, default to "synch"): This determines whether the execution should be *synchronous* or *asynchronous*. In synchronous mode, the notification emitter is blocked until the chain is executed, whereas in asynchronous mode, the execution happens in a separate thread.

The processes are defined by either the *process* or *processref* tags. The order of the process determines their execution order at runtime.

process: As described before.

processref: If a process is used in more than one execution chain, its configuration can be reused by defining it with a *process* tag out side of the *execute* tag, and using this tag to reference it. The *refid* attribute refers to the *id* attribute of the *process*.

5.3 Summary

This chapter explains how to develop self-managing software systems using StarMX. The steps to develop these systems includes requirement specification, application instrumentation, management logic development, framework configuration, and final result deployment. It introduces some techniques used to build such systems and shows how these approaches can be realized by the proposed framework. More information about the configuration of the framework is also presented in this chapter.

Chapter 6

Experimental Studies

StarMX is a software framework to build self-managing systems. It provides the fundamental features to develop such systems and a runtime infrastructure to enable the dynamic adaptation behavior in these systems. This chapter aims to provide an evaluation of this framework. The question here is "*what does evaluation exactly mean in the context of StarMX?*".

This framework does not present any algorithm or model for developing self-managing systems, yet it provides the elements that can be utilized by the developer to build a model or mechanism to address self-* properties. Hence, the relationship of StarMX to a self-managing system is similar to that of a programming language to a software program written in that language. This relationship is also comparable to that of the J2EE infrastructure to J2EE-based software. Therefore, *evaluation* in the above question is translated to finding an answer for the following research questions:

- *Are the framework features suitable for developing a self-managing system? Are they enough? How can they be improved?*
- *How well does the framework perform at runtime? What is its impact on the system performance?*

An IP telephony system is used as a case study to evaluate and analyze our framework through a set of experiments that focus on the research questions. The answer to the first question is subjective, and it is not quantitatively measurable. We aim to answer this question with the best of our knowledge and experience throughout developing an adaptable version of the case study. On the other hand, the second question is quantifiable, and we

address this question by testing the case study under different workloads and calculating the framework performance.

This chapter is organized as follows. The first section describes the case study, and it is followed by the design and implementation of our experiments in the next section. The last section discusses the capabilities of the framework and reports obtained performance results.

6.1 Case Study

Call Controller 2 (CC2) [1] is a Voice over IP prototype system, chosen for conducting experiments on building self-managing systems using the StarMX framework. It is deployed on the Mobicents [63] media server and designed based on a service oriented architecture. It basically provides four main services:

- *Regular VoIP calls*: This is the most basic service provided by all VoIP software. A caller can call a callee to establish a conversation.
- *Call Forwarding*: If a callee is unavailable, CC2 will try to forward the call to the callee's backup address, if it has one.
- *VoiceMail*: A caller can leave a voice message if the callee is unavailable and has no backup address, but his/her voicemail is enabled.
- *Call Blocking*: If a caller is in the callee's blacklist, the call will be blocked.

Mobicents is the first and only open source VoIP Platform certified for JSLEE 1.0 [2]. JSLEE (JAIN Service Logic Execution Environment) is the Java implementation of SLEE. In the telecommunications industry, a SLEE is a high throughput, low latency event processing application environment. The JAIN SLEE specification [3] allows popular protocol stacks such as SIP [4] to be plugged in as resource adapters. The extensible standard architecture naturally accommodates integration points with enterprise applications such as Web, CRM or SOA end points.

[1] http://groups.google.com/group/mobicents-public/web/jain-slee-example-call-controller-2
[2] http://java.sun.com/products/jain/article slee principles.html
[3] http://jcp.org/en/jsr/detail?id=240
[4] https://jain-sip.dev.java.net

Mobicents is deployed on JBoss [45] and brings to telecom applications a robust component model and execution environment. It complements J2EE to enable convergence of voice, video, and data in next generation intelligent applications. One of the main components of JSLEE are Service Building Blocks (SBB), which are comparable to Enterprise Java Beans (EJB) in J2EE systems. Mobicents enables the composition of different SBBs such as call control, billing, user provisioning, administration, and presence sensitive features. Monitoring and management of Mobicents components comes out of the box via the SLEE standard, which is based on JMX and SNMP interfaces. In our experiments, we utilize its JMX-based management facility to manage the CC2 system dynamically.

The architecture of the CC2 application consists of three key SBB components to address its main functionalities:

- *ForwardingSBB*, which provides regular VoIP call and call forwarding services
- *VoiceMailSBB*, which is responsible for the voice mail service
- *BlockingSBB*, that enables the call blocking service

The choice of CC2 as our case study is justified by the following characteristics of CC2: It is an open source Java system, which allows us to investigate and modify its source code; It addresses a real business need (VoIP), rather than a hypothetical one; and It is a large-scale system, which utilizes more features of our proposed framework for adaptation.

6.2 Experiment Design

To evaluate the framework, our experiments are divided into two phases with regard to the research problems: *i*) building an adaptive version of the case study, which is capable of tolerating the varying loads while providing its services to users, and *ii*) analyzing the behavior of StarMX while the system is under different workloads.

6.2.1 Making Case Study Self-Managed

The process of making CC2 adaptive is carried out based on the development steps described in Section 5.1. This phase of the experiment helps us assess the suitability of the framework features and its applicability in a real context. We used this phase to to improve the design of our framework and to find and fix the developmental bugs.

The case study is a relatively large system with 171K lines of code, and several new technologies are involved in its design and implementation. Hence, it took a considerable amount of time for us to read the system documents, understand its architecture, and investigate its source code, before starting this phase of the experiment.

In order to make the adaptation scenarios more realistic, we modified CC2 to have users with different privileges, while preserving all other functionalities of the system. In the modified version, users are categorized into three classes: *Gold*, *Silver*, and *Bronze*, from highest to lowest priority. The gold users are the most valuable users of the VoIP system because they produce the most profit for the company. The Bronze users are the least profitable users, and silver users fall between the other categories. The system owner has to guarantee the quality of services provided to different classes of users, according to their contracts; otherwise, the owner is required to pay them a penalty. Moreover, all users are allowed to access all services provided by the system at all times.

Step1: Requirement Specification - The high level business objective of the new adaptable CC2 is to maximize the company's profit at different workload situations. To achieve this objective, the requirement is to maintain service availability such that it always results in the maximum benefit for the company. For this purpose, the system may decide to block access to a service for low-priority users at certain times (*e.g.* very high loads), in order to guarantee service quality for high-priority users. In other words, the service *availability* and *self-optimizing* property are the target objectives.

Step2: Management Interface Instrumentation - Based on the specified requirements, we need to identify and instrument the required set of management interfaces (sensors and effectors). As discussed earlier, the services in CC2 are provided by three SBB components. Hence, we need a set of sensors and effectors to control these SBBs. The Mobicents server allows managing SBBs through MBeans. Three MBeans (one for each SBB) were designed to provide the average *response time* and *throughput* of each service and to *block* or *unblock* the service for each class of users. Note that each MBean acts as both sensor and effector.

Step3: Management Logic Development - The self-management logic of the case study was developed based on the *GAAM* approach, as described in Section 5.1.1. First, several goals have been designed to reflect the properties of different users based on the requirements. The most high-level goal is "self-optimizing" which is decomposed into lower-level goals that deal with more specific concerns. Each goal also defines its activation conditions. For example, one goal is to "minimize the response time" and it is activated "if the response time is greater than 5 seconds". Goal evaluation was performed by means of policies using the IBM ABLE rule engine [40]. About 30 goals, one voter, and several

execution policies that translate the action plans to MBean method invocations, have been designed for this experiment.

It should be noted that the main purpose of this step was to capture a solid understanding of the StarMX features and facilities needed for developing self-managing systems. GAAM was used just as a model to help us in this respect, and its details (*i.e.* the goals structures, their relationships, and the voting algorithm) are not within the scope of this research.

Step4: StarMX Configuration - The outputs of steps two and three were used to configure the framework and to build control loops. Goal policies, voter, and execution policies were grouped together into several execution chains based on their conflict of interests and commonalities. Each policy was also associated with a set of anchor objects, including the MBeans. The construction of execution chains was one of the most challenging parts of this experiment, since we had to take many issues into account. Some of the challenges were: how to assign weights to goals or prioritize them, and how to elicit action preferences for each goal with respect to the requirements. Again, these challenges are not within the scope of this thesis, as they should be discussed in the context of GAAM evaluation.

Step5: StarMX Deployment and Run - This is the next phase of the experiments, and it is discussed in the next section.

Finally, the architecture of the self-managing version of the CC2 system is shown in Figure 6.1. Several control loops are defined within StarMX, which interact with the CC2 system through the MBean-type sensors/effectors. Thick black arrows display flow of data between control loops and MBeans and also between MBeans and SBB components. Blue arrows denote flow of control among control loop processes.

6.2.2 Testing the Self-Managed Case Study

The second phase of the experiment is to run the system and analyze the results. This phase intends to address two research objectives:

1. Analyzing the behavior of StarMX at runtime and its performance overhead
2. Analyzing the behavior of the adaptive CC2 and the effectiveness of GAAM approach

The first objective is concerned with the framework performance and is one of the key parts of this research. We deployed StarMX locally with the adaptive CC2 system on the

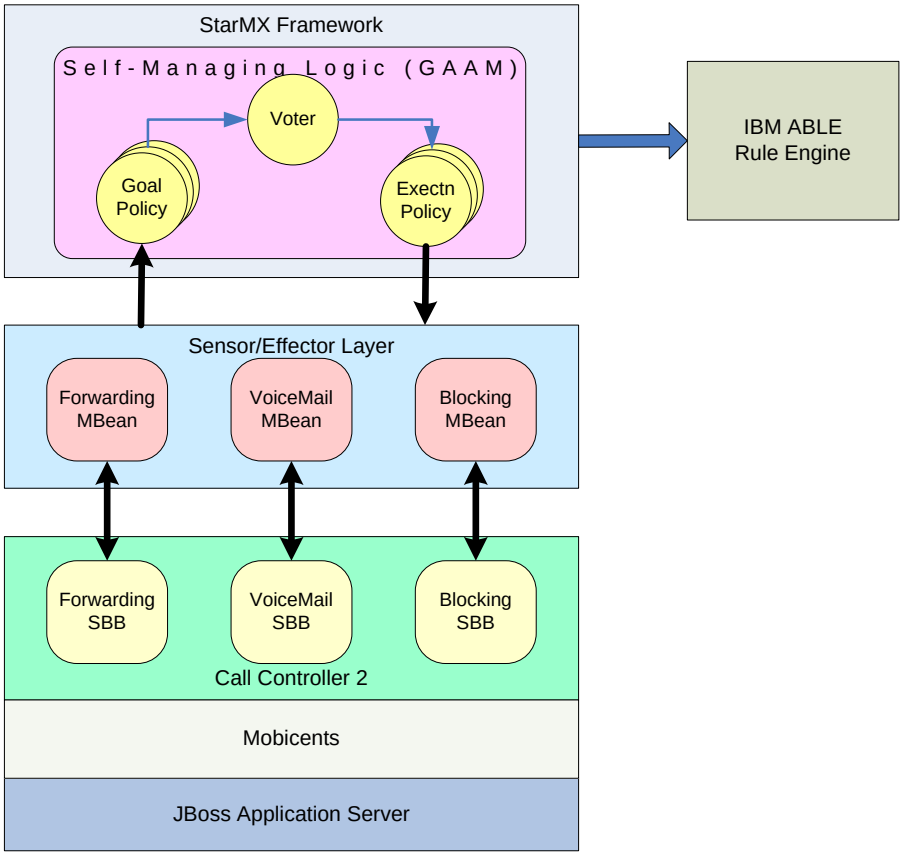

Figure 6.1: Self-Managing CC2 Architecture Enabled by StarMX

same server. We selected the local deployment model, since it has more impact on the system's overall performance due to the resource usage by StarMX and the policy engine.

For the second objective, we had to also run the non-adaptive version of the CC2 system and compare the results with that of the adaptive CC2. The results of this part of experiment were also promising, and they revealed the effectiveness of the GAAM approach. However, these results are not covered in this thesis as they deal with an out-of-scope problem.

Experiment Setup

The experiment was designed to be executed with two different workloads. The results are reported in Section 6.3.3.

- Low load: this workload produces a specified number of requests on behalf of different classes of users (gold, silver, bronze) with a pre-defined time interval between requests. It is designed to be less than the capacity of the system for properly handling workloads without crashing.
- High load: this workload is designed to be above the system capacity for handling workloads by producing requests more frequently. The system should utilize adaptation logic to survive and provide its services to the users.

Furthermore, to minimize the experimental errors due to sporadic events, three replications were conducted for each workload scenario.

A powerful traffic generator should be used to conduct load testing. In this project, we used SIPp 3.1 [5], a free open source load generator for the SIP protocol.

One server and one workstation were used to run the Mobicents server and to generate traffic respectively. The specification of the server was: Windows Server 2003 Standard x64 Edition SP2, Intel Core 2 Quad CPU Q6700 @ 2.66GHz, 8GB of RAM. The workstation was: Windows XP professional SP3, Intel Pentium 4 CPU 3.4GHz, 2GB of RAM. These two machines were connected via 100.0 Mbps Ethernet LAN.

6.3 Discussion and Evaluation

StarMX aims to enable software systems with autonomic capabilities by providing the required set of services for these systems. Evaluation and assurance of autonomic systems has been indicated as one of the major concerns and challenges of these systems [51, 62, 76]. Consequently, the evaluation of development tools and frameworks is a part of this challenge.

To examine the presented approach and to identify its strengths and weaknesses, a set of evaluation criteria or metrics is required. The evaluation criteria will also be helpful for comparing similar solutions and for determining the relative advantages of different approaches. Due to the fact that autonomic computing and self-adaptive systems is an open research area, it still suffers from lack of research results in the evaluation of such systems. There is still no standard mechanism or platform to evaluate the accuracy of these systems, or a generic set of agreed upon evaluation criteria.

[5] http://sipp.sourceforge.net/

In [18], Chen *et al.* present scalability, adaptability, overhead, latency, complexity, and effectiveness as metrics for evaluating dynamic configuration techniques. While the scalability, performance overhead, and complexity metrics are useful in evaluating our framework, the other metrics are concerned with the quality of adaptation technique. McCann *et al.* also define quality of service, cost, granularity/flexibility, failure avoidance, degree of autonomy, adaptivity, time to adapt, reaction time, sensitivity, and stabilization as metric for evaluating the self-management quality of a system [61], such metrics are more suitable for evaluating an adaptation solution like GAAM. In [68], Neti *et al.* propose a list of specific quality criteria to evaluate self-healing systems.

In order to enable the comparison of different self-adaptation approaches, Cheng *et al.* suggest using utility theory to merge several quality dimensions [20]. They introduce the Self-Adaptation Fitness Unit (SAFU) as a utility function, which is the weighted sum of different quality aspects, such as resource overhead and adaptation engineering effort. The problem of this method is difficulty of quantifying quality attributes and calculating SAFU.

However, due to the qualitative nature of the first phase of our experiments, we present a discussion of the framework capabilities, and review its quality attributes, such as scalability and reusability, in the next sections. Then, we report the results of performance evaluation and show how effective the framework is at runtime.

6.3.1 Framework Capabilities Discussion

One of the major research goals of our experimental studies is to understand the suitability and fitness of the framework in a real situation from a user's perspective. Developing a self-managing version of the CC2 system helped us identify and resolve design and implementation issues in our framework and to improve its feature set. At the end of this phase of the experiment, we realized that StarMX satisfies all the needs for enabling adaptation behavior in the selected case study and provides the required features. In our analysis of the experiment results, we were more concerned with self-optimizing and proactive self-healing for improving system availability and avoiding crashes. More experiments with different case studies are required to deal with other self-* properties to reach a more precise understanding of the framework features in addressing different self-managing aspects.

As discussed earlier in this section, the presented evaluation metrics in the literature concentrate more on the quality of adaptation techniques rather than on enabling frameworks and tools. Therefore, we decided to analyze our framework based on a set of suggested parameters, which reflect its capabilities from different aspects that complement

each other. They can also be used by other researchers as the key design issues in building enabler solutions like StarMX. This is not a comprehensive set, but can be used as a basis for comparing different solutions.

- *Degree of autonomy*: The capability of a framework in automating the management process, which ranges from manual to fully autonomic. IBM defines the five levels of *manual*, *instrument and monitor*, *analysis*, *closed loop*, and *closed loop with business priorities*, from lowest to highest degree of automation [41].

- *Control Scope*: The granularity or the scope of what is being managed. From the smallest to the largest, the levels are: *subcomponent* (portion of a resource), *single instance* (an entire resource like an application), *multiple instances of the same type*, *heterogenous instances*, and *business system* (a complete set of hardware and software resources) [41].

- *Self-* properties support*: The capability of the framework in properly addressing self-configuring, self-healing, self-optimizing, and self-protecting properties. This metric shows the generality of a framework in dealing with different self-managing requirements.

- *Management logic expression*: The mechanisms for defining the self-managing requirements. Possible categories are: *descriptive* format (specific or arbitrary policy language) and *programmatic* format (programming language and APIs). This metric affects the usability of the framework.

- *Control loop construction*: The mechanisms and facilities provided to support creating closed control loops. Some frameworks enforce a particular approach or algorithm to address this matter.

- *Monitoring technique*: The capability of the framework in supporting different mechanisms for monitoring or activating control loops. The common techniques are *timer-based* and *event-based*.

- *Data communication facility*: The capability of the framework in facilitating communication between control loops or autonomic elements. These elements need to collaborate with each other at runtime to make the best decision for adaptation. Values of this property fall within the spectrum of *no-support* to *fully-supported*.

- *Remote management*: The ability of the framework to enable managing a system remotely and transferring the adaptation cost to a different machine. This property reveals the scalability of the framework in dealing with management logic.

- *Applicable environment*: The characteristics or specification of the environment and the target systems that the framework can successfully work with.
- *Managing non-Java systems*: The capability of the framework to be used in non-Java environments. This shows how the framework can be used in such contexts, if possible.
- *Runtime updating management logic*: The capability of the framework in allowing runtime modification of the management logic. The system objectives may change during its life cycle, and it is necessary to be able to update them at runtime without interrupting the working system.

Table 6.1 summarizes StarMX capabilities in terms of the suggested parameters. Unfortunately, we are unable to provide a comparison between our framework and similar projects in the literature (like Accord [56], AMT [2], ASF [31], Rainbow [29], and J3 Process [87]). The reason is the lack of public access to those frameworks for practical evaluation. Also, their corresponding papers do not clearly report their capabilities as we have discussed here.

Table 6.1: Framework Capabilities Summary

Parameter	**StarMX capability**
Degree of autonomy	Closed loop
Control scope	Multiple instances
Self-* properties support	Potentially, all self-* properties
Management logic expression	Both descriptive and programmatic
Control loop construction	Flexible architecture
Monitoring technique	Both timer-based and event-based
Data communication facility	Supported by data repositories (memory scopes)
Remote management	Supported
Applicable environment	Any Java-based system
Managing non-Java systems	Via WebServices- or JNI-based anchor objects
Runtime modification	*Under construction*

Regarding the framework support for addressing all self-* properties, StarMX allows the construction of MAPE loops, which interact with the system via a set of sensors and effectors. It is believed that with such an architecture, any self-managing aspect can be addressed [37, 41]. However, in the context of our framework, this should be practically proven by more experimental studies in the future.

6.3.2 Quality Attributes Review

The generality and standard-compliance properties of our framework make it compatible with different Java EE application servers like JBoss [45] and Weblogic [70]; this has been proved by our test results. These properties also help to maintain quality attributes such as reusability, usability, and flexibility. Various attributes have been discussed in the software engineering literature to evaluate the quality of software from different perspectives, for instance in [9, 86]. To analyze this work, we have selected the attributes that we believe are more important for a self-management support framework. Note that this discussion is based on the best of knowledge and experience of the designers and developers of StarMX, and these attributes have been kept in mind while developing this framework.

Flexibility: It is the ability that allows the developer to combine his own mechanism, algorithm, or technique in the design and implementation of the self-management logic. StarMX provides flexibility in dividing the control loop functions into any number of consecutive processes, in constructing autonomic managers statically or dynamically, and in describing the logic as rules, policies, or even code.

Scalability: This attribute shows the capability of a system in properly handling a growing amount of load. This property is important when performance is a critical factor. Because of adaptation, if the performance overhead imposed on the system is significant, the scalability of the framework becomes more important. The framework must be able to transfer the load to another machine or remotely manage the system. The StarMX user can break the management logic down into more than one configuration file and deploy and run multiple instances of the framework locally or remotely, such that each instance is responsible for dealing with a different part of the self-management requirements.

Usability: This attribute determines how easy it is to employ the framework in an application. It can also be considered as the cost of or effort needed for framework adaptation in a particular context. This attribute plays a key role in the success of the software. Many software systems and frameworks are not used due to their difficulty of usage. This attribute is affected by several parameters, such as the newly introduced concepts (model, language, etc.) that should be learned by the users, or the amount of coding and configuration effort required for its application.

Following standards, easy integration with different policy engines, and not enforcing any specific approach to the self-managing problem simplify our framework instantiation and improve its usability. As an example, the policy adapter class for the

Imperius framework was implemented with about 80 lines of code, which reflects the simplicity of integrating the framework with a typical policy engine. The framework configuration is also designed to be as simple as possible: while it allows very detailed configuration, only the key properties are mandatory, and other fields can be left at their default values.

Reusability: This attribute is the ability of using the framework for different systems. A generic design that considers different usage scenarios improves this factor. On the other hand, poor design or any dependency on a particular architectural style, execution environment, other components or frameworks, or a specific self-managing aspect affects reusability. The presented framework is fully reusable and applicable to different Java-based systems, since its interaction with its environment is through standard Java features and interfaces.

Extendibility: This attribute permits the developer to extend the framework to add new features or to integrate it with other frameworks. StarMX is an open-source project with well-defined APIs that support this attribute. Integration with different policy engines is an example of this capability.

6.3.3 Performance Evaluation

Analyzing the framework's performance is our second major research objective. In large-scale and server-based systems, performance is one of the key characteristics. In self-managing systems, the execution of management logic adds extra overhead to the system's performance, and it is desired to reduce this overhead by applying efficient techniques and optimized frameworks.

The first step of performance evaluation is performance tuning. This step is required to discover the bottlenecks in the software and to optimize the source code to resolve the problems. To achieve this, we executed the system under a high volume of workloads and monitored the framework runtime behavior using a specialized profiling tool called JProfiler [48]. Designed for Java-based systems, it is a professional tool that is capable of reporting the memory usage of individual objects and the time consumed in each invoked method in the system. After several rounds of code improvement and execution, we started the next step, which is the detailed analysis of adaptation cost.

The performance of a framework is said to be good if its impact on the overall system performance is negligible. Using the StarMX framework, the execution cost of adaptation

is considered to be that of the execution chains. This cost is composed of the following portions, which except for the last part, are out of the control of the framework:

- *Sensing and Effecting cost*: The amount of time spent in the invoked method of a sensor or effector to get or set an attribute or to execute an operation on the target resource.
- *MBean proxy cost*: If the anchor object is an MBean, this is the time spent in the proxy itself (from the invocation request time to the moment that the MBean is invoked). In the JMX architecture, since all accesses to MBeans are directed through the MBeanServer, this cost will be different in the remote and local access modes.
- *Process cost*: If the process is policy-based, this will be the time spent in the policy engine to execute a policy. Otherwise, this is the time required to execute the process Java code, which is developed by the user. In the remote deployment scenario, this time is spent on another machine and does not affect the system's performance.
- ***Framework cost***: The remaining execution time is spent in the framework from the start of execution to its end, and we aim to minimize this portion of the cost by optimizing our framework. Moreover, in the remote mode, framework cost does not affect the system's performance.

As mentioned before, local and remote deployments have different impacts on performance. In the remote mode, the last three portions of cost will not affect the system's overall performance since they consume another machine's resources. Only the anchor objects (like MBeans) that are collocated with the system components and accessed remotely will impose some performance overhead. Furthermore, MBean proxy cost may considerably affect the total execution cost due to the network latency in remote invocation of MBean objects. On the other hand, all cost portions will affect performance if the framework is deployed locally. Hence, we decided to conduct our experiments in local mode to analyze the framework overhead.

To obtain performance-related data, we enabled sensor/effector MBeans with the facility to compute their execution time. Also, we used the *data gathering* and *logging* services of the framework to collect detailed data on the execution of processes and execution chains. We also developed a set of software tools to analyze the StarMX output log file to calculate the expected cumulative numbers.

Table 6.2 summarizes obtained results for the local deployment of our framework in two different work loads: *Low* and *High*. As mentioned in Section 6.2.2, each work load has

been tested three times, and the mathematical average of the raw data are reported here. The presented performance metrics are divided into two sections. The first half shows the metrics measured directly while the system was running. The second half illustrates the computed results based on the first half data. These metrics reflect the framework overhead more distinctively. Note that all times are reported in seconds.

Table 6.2: Performance Analysis Result

Performance metrics	**Low load**	**High load**
Load test time	732 (sec)	850 (sec)
No. of executed processes	1191	1472
Total adaptation time	2.8183	4.7152
Total process execution time	2.3299	3.8645
Total sensing/effecting time	1.0515	1.3617
Average process execution time	2.3663 E-3	3.2032 E-3
Average framework cost per process	**0.41 E-3**	**0.578 E-3**
Adaptation proportion to total time	0.38%	0.55%

- *Load test time*: The total time the system was under load testing.
- *No. of executed processes*: The total number of Java-based and policy-based processes executed in all execution chains during the load test.
- *Total adaptation time*: The total time spent for adaptation.
- *Total process execution time*: The total time spent for execution of all processes. This value is included in the *total adaptation time*.
- *Total sensing/effecting time*: The total time consumed for sensing and effecting purposes. This time is included in the *total process execution time*.
- *Average process execution time*: The average execution time of a process, calculated by dividing *total process execution time* by the *no. of executed processes*.
- *Average framework cost per process*: The average time spent in the framework for executing a process. It is equal to (*total adaptation time - total process execution time*) / *no. of executed processes*. This is a key parameter, which separates the effect of the framework on overall system performance from other contributing factors.

- *Adaptation proportion to total time*: The percentage of the total time spent on adaptation, computed by dividing *total adaptation time* by *load test time*.

The difference between load test times in two workloads is attributed to the difference between the number of requests and the delay between them in the two test modes. Although the system is expected to work properly without any adaptation in the low test scenario, we observed that it performed some adaptation stemming from false-positives in the implementation of policies and goals. However, the values in the high load column show that a greater amount of adaptation has occurred in the high load scenario.

Besides hardware performance, the total adaptation time is affected by several factors, including: the components that are managed, the adaptation logic (*e.g.* policies), the frequency of adaptation, and the framework itself. The *average framework cost per process* metric shows the performance of the framework, and how effectively it works at runtime. The lower this number is, the more optimized the framework behaves. This metric views the framework performance from a per process perspective. As shown in Table 6.2, this number is about 0.5 milliseconds in the operational environment, which is negligible and meets our performance objective, stated in *NFR1* (See Section 3.2).

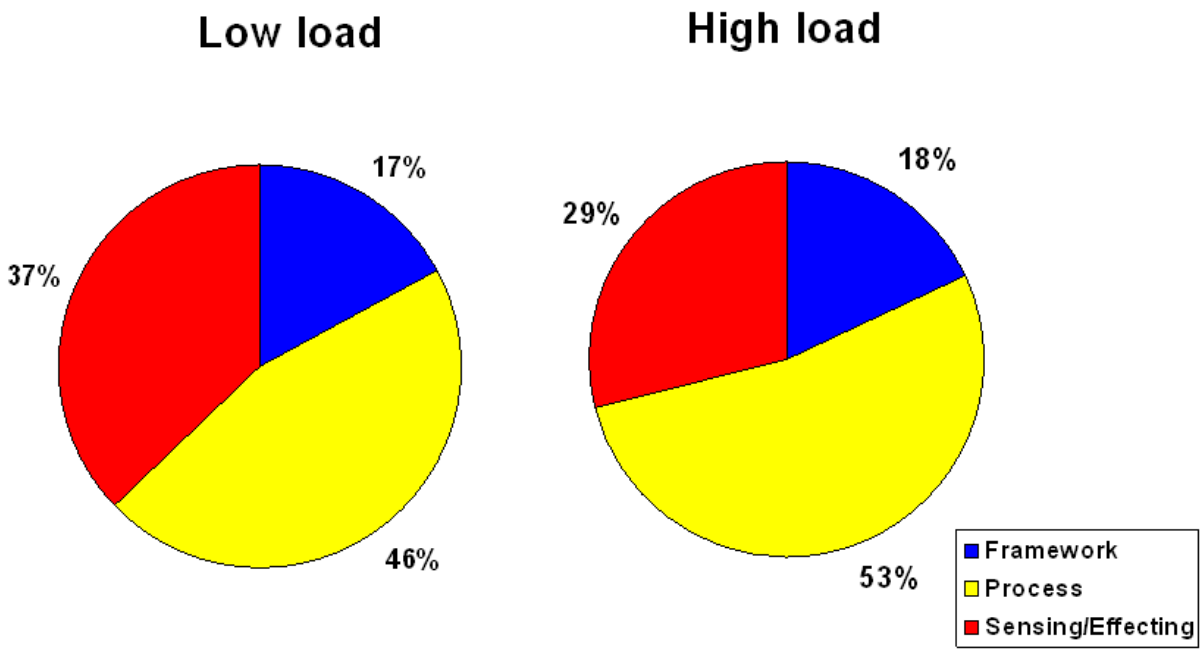

Figure 6.2: Adaptation Cost Distribution

Moreover, Figure 6.2 shows the detailed distribution of the adaptation cost between framework, processes, and sensing/effecting portions in both load scenarios. The majority of the adaptation cost stems from the execution of the processes, which were mostly policy-based. This segment is affected by the complexity of the policies and the performance of the policy engine in executing the policies. We used the IBM ABLE rule engine [40], and we noticed a remarkable difference in its performance for the different load tests. The share of framework cost remains the same (about 18%) in both scenarios, which reveals the consistent behavior of the framework in different work loads.

However, we believe that using the same case study in both phases of our experimental studies (development and testing) is not a threat to the validity of our evaluations due to the following points. First, the testing phase always depends on the development phase, and it is not possible to test a scenario without developing it. Second, the focus of our testing phase is on the performance of the proposed framework, rather than the quality of adaptation in the self-managing system. This matter eliminates the dependency of testing results to the characteristics of self-managing system. Third, to avoid the risk of sporadic events the testing phase has been performed several times, and the aggregated results are reported here.

6.4 Summary

This chapter presents the experimental studies conducted to evaluate the StarMX framework. CC2 is a VoIP system based on JSLEE technology, and was used as a case study in the experiments. The major goals of the study are to assess the suitability of the framework for building self-managing systems and to analyze its performance overhead. The experiment consists of two steps. First, creating an adaptable version of the CC2 system, and then testing the new system under different work loads, thereby simulating the real operational profile. The evaluation results of this experiment are very promising and satisfy our objectives for the development of this framework. The results are reported to reflect the framework capabilities in enabling self-managing properties, the quality characteristics of the framework such as reusability or extensibility, and the performance overhead incurred by the framework itself.

Chapter 7

Conclusion and Future Directions

In this chapter, we summarize the findings of the thesis and outline future directions that can be pursued from this research. Section 7.1 presents the contributions of the work presented in the thesis. Section 7.2 outlines some potential future work for extending this research. Finally, Section 7.3 finishes the thesis with some concluding remarks.

7.1 Contributions

The major contribution of the proposed research is that, it facilitates the development of self-managing Java-based systems by means of a generic software framework. The principle contributions of this thesis were described in Chapter 1. We restate these with more information based on the remainder of the thesis:

- StarMX provides the notion of Execution Chain as a configurable and flexible mechanism to construct control loops that consists of a sequence of processes. Each process may act as a MAPE loop entity, supporting the monitoring, analyzing, planning, and executing functions. This mechanism allows separating management logic from business logic.
- It presents an infrastructure as the container of execution chains that activates them at runtime and maintains their life cycle.
- It defines the concept of anchor object as the interface between the execution chains and the software to be managed.

- It provides a set of services to support the runtime behavior of the framework, including: a) standard access mechanisms to anchor objects; b) several activation techniques to invoke execution chains at appropriate times; c) various communication facilities between execution chains by means of data reporsitories; d) statistical data gathering about the runtime behavior of the execution chains and processes; e) a logging feature to record internal events; f) a method to combine several control loops dynamically and to create a bigger control loop on-the-fly.

- StarMX prepares a platform for developing and testing various adaptation solutions to deal with different self-* properties and system resources. As an example, the GAAM [77] approach has been used to address the self-optimization property in the case study.

7.2 Future Work

The StarMX framework is a newly released system and is in its early stages. There are numerous ways to improve and extend this work. The following outlines several potential directions to enhance this framework:

- One of the best approaches to understand the limitations and shortcomings of a software system is to use it in practice. This framework should be employed in many other adaptation scenarios to discover new features that should be added or the problems to be fixed. Addressing different self-* properties will reveal new directions for improving the framework.

- High-level system objectives change in time; such changes should be reflected in the management logic. The framework should provide mechanisms to modify the configured properties and policies dynamically.

- A framework is a software module, and similar to other resources, it may require adaptation at runtime. Therefore, it should expose a set of sensors and effectors to enable its management by itself or through other means.

- Service oriented computing is a new trend in software development. Web Services realizes this model by constructing a system as a composition of different components, and by providing coarse-grained services. Service-based systems can also be managed by web-services. As described in Chapter 2, WSDM [69] defines the standard for using web-services for application management. In Section 5.1.1, we presented an approach

to realize this idea. However, this facility can be improved in the framework, so that the user only needs to provide the web-service information (*i.e.* WSDL) to the framework; a proxy object is then created for the web-service, which represents it as a simple java object.

- To achieve the best result from using a new technology, practitioners always rely on guidelines and best practices related to the subject. This framework presents a new technology in automated application management, and it therefore needs a set of guidelines to assist the developer in using it.

7.3 Conclusion

In this thesis, we presented the StarMX framework, which aims at supporting the development of self-adaptive systems. It captures the common problems in different adaptation solutions and provides a set of features and a runtime environment to enable self-management properties in a configurable manner. We discussed the internal design and architecture of the framework and defined a process for developing a self-managing system from scratch, or converting a legacy system to a self-managing one. Several sample scenarios have been explained to show how different features of the framework can be utilized.

In the next step, we devised a set of experimental studies to evaluate the work. A sample self-managing system was developed to enhance availability and self-optimizing properties in a VoIP system. This phase of the experiment was important for capturing a sound assessment of the framework's suitability for a real application. It also helped us in improving the design of the framework from different quality perspectives, like usability, reusability, scalability, flexibility, and extendibility. The performance overhead of the framework was also analyzed by running the self-managing VoIP system under different workloads. The performance overhead is related to different components including sensors/effectors, adaptation logic (policies), and the framework modules. Through a set of optimizations, we improved the performance of the framework to minimize its impact on the overall performance. Finally, our observations revealed that the framework is effective in addressing self-managing concerns with negligible performance overhead.

APPENDICES

Appendix A

Sample StarMX Configuration File

The following shows a sample "starmx.xml" configuration file.

```
<!-- THIS IS A SAMPLE STARMX.XML FILE, DESIGNED FOR ILLUSTRATION PURPOSE ONLY -->
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE starmx PUBLIC "-//STAR Lab//StarMX Configuration DTD 1.0//EN" "starmx-1.0.dtd">
<starmx>

    <!-- Creating connection to JBoss MBeanServer using JMX Remote API -->
    <mbeanserver id="jboss_ms" lookup-type="jmx" >
        <jmx-param service-url="service:jmx:rmi://localhost
                        /jndi/rmi://localhost:1090/jmxconnector" />
    </mbeanserver>

    <!-- Anchor objects -->
    <mbean id="mbean1" object-name="starmx:name=mb1" mbeanserver="jboss_ms" />
    <mbean id="mbean2" object-name="starmx:name=mb2" mbeanserver="jboss_ms" />

    <bean id="mailSender" class="util.MailSender" />

    <monitor-mbean  id="gaugeMonitor"
                    observed-attribute="size" granularity-period="1000"
                    mbeanserver="jboss_ms"
                    object-name="starmx:type=monitor,name=gm" >
        <gauge-monitor high-threshold="100" low-threshold="10" notify-high="true" />
        <observed-object object-name="starmx:type=Control,name=*"/>
    </monitor-mbean>

    <!-- A reused policy-based process based on the SPL language -->
    <process id="proc1" policy-type="spl" policy-file="policy1.spl">
```

```
        <object name="mb1" ref="mbean1" />
        <object name="mailSender" ref="mailSender" />
    </policy>

    <!-- A timer-based execution chain containing two processes -->
    <execute >
        <timer-info interval="30" unit="second" />

        <processref refid="proc1" />

        <!-- local process to the chain, described by Drools rule language (drl) -->
        <process id="proc2" policy-type="drl" policy-file="policy2.drl">
            <object name="mb1" ref="mbean1" />
            <object name="mb2" ref="mbean2" />
        </process>
    </execute>

    <!--
        A notification-based execution chain containing two processes,
        activated by the notifications coming from the defined gauge
        monitor. Only the THRESHOLD_HIGH_VALUE_EXCEEDED notification types
        will activate the chain.
    -->
    <execute >
        <notification-info  emitter-mbean="gaugeMonitor"
                            event-type="jmx.monitor.gauge.high" />

        <processref refid="proc1" />

        <!--
            A local Java-based process, which uses only the gauge monitor mbean
            as anchor object
        -->
        <process id="proc3" javaclass="test.MySampleProcess">
            <object name="monitor" ref="gaugeMonitor" />
        </process>
    </execute>

    <property name="starmx.log.level">warn</property>
    <property name="starmx.log.dir">/starmx/log</property>

    <!-- Introducing the policy adapter for Drools rule engine -->
    <property name="starmx.policy.adapter.drl">
                    sample.policy.DroolsPolicyAdapter</property>
</starmx>
```

References

[1] Takoua Abdellatif. Enhancing the Management of a J2EE Application Server Using a Component-Based Architecture. In *EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 70–77, September 2005. 13

[2] Jakub Adamczyk, Rafal Chojnacki, Marcin Jarzab, and Krzystof Zielinski. Rule Engine Based Lightweight Framework for Adaptive and Autonomic Computing. *LNCS*, 5101:255–364, June 2008. 8, 10, 65

[3] Richard J. Anthony. A Policy-Definition Language and Prototype Implementation Library for Policy-based Autonomic Systems. In *IEEE International Conference on Autonomic Computing (ICAC)*, pages 265–276, June 2006. 7

[4] Apache. Imperius. http://incubator.apache.org/imperius. 23, 31

[5] Apache. Log4j. http://logging.apache.org/log4j. 38

[6] Apache. Struts. http://struts.apache.org/. 16

[7] 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 (SEAMS)*, pages 58–67, May 2009. 7

[8] AspectJ. http://www.eclipse.org/aspectj/. 7

[9] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003. 66

[10] Umesh Bellur and Amar Agrawal. Root Cause Isolation for Self Healing in J2EE Environments. In *International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 324–327, July 2007. 11

[11] Philippe Boinot, Renaud Marlet, Jacques NoyC, Gilles Muller, and Charles Consel. A Declarative Approach for Designing and Developing Adaptive Components. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 111–119, September 2000. 7

[12] Dario Bonino, Alessio Bosca, and Fulvio Corno. An Agent Based Autonomic Semantic Platform. In *International Conference on Autonomic Computing (ICAC)*, pages 189–196, May 2004. 11

[13] Dario Bonino, Fulvio Corno, and Laura Farinetti. Dose: A distributed open semantic elaboration platform. In *IEEE International Conference on Tools with Artificial Intelligence*, pages 580–588, November 2003. 11

[14] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. *Software - Practice and Experience*, 36(11-12):1257–1284, 2006. 13

[15] Radu Calinescu. Challenges and Best Practices in Policy-Based Autonomic Architectures. In *IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pages 65–74, September 2007. 7, 32

[16] George Candea, Aaron B. Brown, Armando Fox, and David Patterson. Recovery-Oriented Computing: Building Multitier Dependability. *IEEE Computer*, 37(11):60–67, 2004. 11

[17] George Candea, Emre Kiciman, Shinichi Kawamoto, and Armando Fox. Autonomous Recovery in Componentized Internet Applications. *Cluster Computing*, 9(2):175–190, April 2006. 11

[18] Huoping Chen and Salim Hariri. An Evaluation Scheme of Adaptive Configuration Techniques. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 493–496, November 2007. 63

[19] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Architecture-Based Self-Adaptation in the Presence of Multiple Objectives. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 2–8, May 2006. 8

[20] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Evaluating the effectiveness of the rainbow self-adaptive system. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 132–141, May 2009. 63

[21] Ivica Crnkovic and Magnus Larsson. *building reliable component-based Software Systems*. Artech House, 2002. 16

[22] Eric M. Dashofy, Andre van der Hoek, and Richard N. Taylor. Towards Architecture-based Self-Healing Systems. In *Workshop on Self-healing Systems*, pages 21–26, November 2002. 11

[23] Mikael Desertot, Clement Escoffier, and Didier Donsez. Autonomic Management of J2EE Edge Servers. In *International ACM Workshop on Middleware for Grid Computing*, pages 1–6, November 2005. 13

[24] DMTF. Common Information Model-Simplified Policy Language (CIM-SPL), 2007. www.dmtf.org/standards/published_documents/DSP0231.pdf. 23

[25] Gary Duzan, Joseph Loyall, Richard Schantz, Richard Shapiro, and John Zinky. Building Adaptive Distributed Applications with Middleware and Aspects. In *International Conference on Aspect-Oriented Software Development (AOSD)*, pages 66–73, March 2004. 7

[26] Onyeka Ezenwoye and S. Masoud Sadjadi. TRAP/BPEL: A Framework for Dynamic Adaptation of Composite Services. Technical Report FIU-SCIS-2006-06-02, Florida International University, June 2006. 7

[27] Mohamed Fayad and Douglas C. Schmidt. Object-Oriented Application Frameworks. *Communications of the ACM*, 40(10):32–38, 1997. 16, 17

[28] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995. 7, 29, 31, 34

[29] David Garlan, Shang Wen Cheng, An Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer*, 37(10):46–54, October 2004. 8, 65

[30] David Garlan and Bradley Schmerl. Model-based Adaptation for Self-Healing Systems. In *Workshop on Self-healing Systems*, pages 27–32, November 2002. 11

[31] Ian Gorton, Yan Liu, and Nihar Trivedi. An Extensible, Lightweight Architecture for Adaptive J2EE Applications. In *International Workshop of Software Engineering and Middleware*, pages 47–54, November 2006. 8, 9, 65

[32] Ian Gorton, Yan Liu, and Nihar Trivedi. An extensible and lightweight architecture for adaptive server applications. *Software: Practice and Experience*, 38(8):853 – 883, October 2007. 9

[33] Denis Gracanin, Shawn A. Bohner, and Michael Hinchey. Towards a Model-Driven Architecture for Autonomic Aystems. In *IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 500–505, May 2004. 10

[34] Salim Hariri, Bithika Khargharia, Houping Chen, Jingmei Yang, Yeliang Zhang, Manish Parashar, and Hua Liu. The Autonomic Computing Paradigm. *Cluster Computing*, 9(1):5–17, 2006. 1

[35] Aaron Helsinger, Michael Thome, and Todd Wright. Cougaar: A Scalable, Distributed Multi-Agent Architecture. In *IEEE International Conference on Systems, Man and Cybernetics*, volume 2, pages 1910–1917, October 2004. 10

[36] Hibernate. http://www.hibernate.org/. 16

[37] Michael G. Hinchey and Roy Sterritt. Self-managing software. *IEEE Computer*, 39(2):107–109, Feb 2006. 1, 65

[38] HP. Design Patterns for JMX and Application Manageability, June 2005. http://www4.java.no/javazone/2005/presentasjoner/JustinMurray/Justin_Murray-DesignPatternsForJMX_javazone2005.pdf. 21

[39] Markus C. Huebscher and Julie A. McCann. A Survey of Autonomic Computing—Degrees, Models, and Applications. *ACM Computing Survey*, 40(3):1–28, August 2008. 32

[40] IBM. Agent Building and Learning Environment (ABLE). http://www.research.ibm.com/able. 31, 59, 71

[41] IBM. An architectural blueprint for autonomic computing. White paper, June 2006. http://www-01.ibm.com/software/tivoli/autonomic/pdfs/-AC_Blueprint_White_Paper_4th.pdf. 1, 3, 11, 18, 42, 64, 65

[42] Florian Irmert, Thomas Fischer, and Klaus Meyer-Wegener. Runtime Adaptation in a Service-Oriented Component Model. In *ICSE International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 97–104, May 2008. 13

[43] I. Jacobson, M. L. Griss, and P. Jonsson. *Software reuse: architecture, process and organization for business success*. ACM Press/Addison-Wesley, 1997. 16

[44] Michael Jarrett and Rudolph Seviora. Constructing an Autonomic Computing Infrastructure Using Cougaar. In *IEEE International Workshop on Engineering of Autonomic and Autonomous Systems*, pages 119–128, March 2006. 11

[45] JBoss. Application Server. http://www.jboss.org/jbossas. 58, 66

[46] JBoss. Drools. http://www.jboss.org/drools. 23

[47] Ralph Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988. 15

[48] JProfiler. http://www.ej-technologies.com/products/jprofiler/overview.html. 67

[49] Eser Kandogan, Christopher S. Campbell, Peter Khooshabeh, John Bailey, and Paul P. Maglio. Policy-based Management of an E-commerce Business Simulation: An Experimental Study. In *IEEE International Conference on Autonomic Computing (ICAC)*, pages 33–42, June 2006. 7

[50] Jeffrey O. Kephart. Research Challenges of Autonomic Computing. In *International Conference on Software Engineering (ICSE)*, pages 15–22, May 2005. 3

[51] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, January 2003. 1, 62

[52] Jeff Kramer and Jeff Magee. Self-Managed Systems: an Architectural Challenge. In *Future of Software Engineering (FOSE)*, pages 259–268, May 2007. 12, 42, 44, 47

[53] Heather Kreger, Ward Harold, and Leigh Williamson. *Java and JMX: Building Manageable Systems*. Addison-Wesley, 2002. 21

[54] Alexei Lapouchnian, Yijun Yu, Sotirios Liaskos, and John Mylopoulos. Requirements-Driven Design of Autonomic Application Software. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, pages 80–94, 2006. 43

[55] Timothy C. Lethbridge and Robert Langaniere. *Object-Oriented Software Engineering.* McGraw-Hill, 2001. 16

[56] Hua Liu and Manish Parashar. Accord: A Programming Framework for Autonomic Applications. *IEEE Transactions on Systems, Man, and Cybernetics*, 36(3):341–352, May 2006. 9, 65

[57] Hua Liu, Manish Parashar, and Salim Hariri. A Component-Based Programming Model for Autonomic Applications. In *International Conference on Autonomic Computing (ICAC)*, pages 10–17, May 2004. 9

[58] Joseph P. Loyall, David E. Bakken, Richard E. Schantz, John A. Zinky, David A. Karr, Rodrigo Vanegas, and Kenneth R. Anderson. QoS Aspect Languages and Their Runtime Integration. *LNCS*, 1511:303–318, May 1998. 7

[59] Leonidas Lymberopoulos, Emil Lupu, and Morris Sloman. An adaptive policy-based framework for network services management. *Journal of Network and Systems Management*, 11(3):277–303, September 2003. 7

[60] P. Martin, W. Powley, K. Wilson, W. Tian, T. Xu, and J. Zebedee. The WSDM of Autonomic Computing: Experiences in Implementing Autonomic Web Services. In *ICSE International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, May 2007. 8

[61] Julie A. McCann and Markus C. Huebscher. Evaluation Issues in Autonomic Computing. *LNCS*, 3252:597–608, September 2004. 63

[62] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H.C. Cheng. Composing Adaptive Software. *IEEE Computer*, 37(7):56–64, July 2004. 6, 7, 62

[63] Mobicents. http://www.mobicents.org. 57

[64] Mirko Morandini, Loris Penserini, and Anna Perini. Towards Goal-Oriented Development of Self-Adaptive Systems. In *ICSE International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 9–16, May 2008. 43

[65] Arun Mukhija and Martin Glinz. CASA - A Contract-based Adaptive Software Architecture Framework. In *IEEE Workshop on Applications and Services in Wireless Networks*, pages 275–286, July 2003. 7, 10

[66] Arun Mukhija and Martin Glinz. Runtime Adaptation of Applications Through Dynamic Recomposition of Components. *LNCS*, 3432:124–138, March 2005. 7, 10

[67] Arun Mukhija and Martin Glinz. The CASA Approach to Autonomic Applications. In *IEEE Workshop on Applications and Services in Wireless Networks*, pages 173–182, July 2005. 10

[68] Sangeeta Neti and Hausi A. Muller. Quality Criteria and an Analysis Framework for Self-Healing Systems. In *ICSE International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, May 2007. 63

[69] OASIS. Web Services Distributed Management (WSDM). http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsdm. 8, 47, 73

[70] Oracle and BEA. Weblogic Application Server. http://www.oracle.com/bea/index.html. 66

[71] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime Software Adaptation: Framework, Approaches, and Styles. In *International Conference on Software Engineering (ICSE)*, pages 899–910, May 2008. 7

[72] Nauman A. Qureshi and Anna Perini. Engineering Adaptive Requirements. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 126–131, May 2009. 43

[73] S. Masoud Sadjadi, Philip K. McKinley, Betty H. Cheng, and R. Kurt Stirewalt. TRAP/J: Transparent Generation of Adaptable Java Programs. *LNCS*, 3291:1243–1261, October 2004. 7, 8

[74] Mazeiar Salehie, Sen Li, Reza Asadollahi, and Ladan Tahvildari. Change Support in Adaptive Software: A Case Study for Fine-Grained Adaptation. In *International IEEE Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, pages 35–44, April 2009. 42

[75] Mazeiar Salehie, Sen Li, and Ladan Tahvildari. Employing Aspect Composition in Adaptive Software Systems: A Case Study. In *International ACM Linking Aspect Technology and Evolution Workshop*, pages 17–21, March 2009. 7, 42

[76] Mazeiar Salehie and Ladan Tahvildari. Autonomic Computing: Emerging Trends and Open Problems. In *International ICSE Workshop on Design and Evolution of Autonomic Application Software (DEAS)*, pages 82–88, May 2005. 43, 62

[77] Mazeiar Salehie and Ladan Tahvildari. A Weighted Voting Mechanism for Action Selection Problem in Self-Adaptive Software. In *International IEEE Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 328–331, July 2007. 12, 42, 44, 46, 73

[78] Mazeiar Salehie and Ladan Tahvildari. Self-Adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):1–42, May 2009. 2, 6, 7

[79] Marcio A. S. Sallem and Francisco Jose da Silva e Silva. The Adapta Framework for Building Self-Adaptive Distributed Applications. In *International Conference on Autonomic and Autonomous Systems (ICAS)*, June 2007. 7, 8, 10

[80] Michael E. Shin. Self-healing components in robust software architecture for concurrent and distributed systems. *Science of Computer Programming*, 57(1):27–44, July 2005. 11

[81] Roy Sterritt. Autonomic computing. *Innovations in Systems and Software Engineering*, 1(1):79–88, April 2005. 7

[82] Sun Microsystems. Java Management Extensions (JMX). http://java.sun.com/jmx. 3, 8

[83] Sun Microsystems. Java Management Extensions (JMX) Specification v1.4, 2006. http://java.sun.com/javase/6/docs/technotes/guides/jmx/JMX_1_4_specification.pdf. 21

[84] Emil Vassev and Joey Paquet. ASSL - Autonomic System Specification Language. In *IEEE Software Engineering Workshop*, pages 300–309, Febuary 2007. 12, 42

[85] Emil Vassev and Joey Paquet. Towards an Autonomic Element Architecture for ASSL. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, May 2007. 12

[86] Hans Van Vliet. *Software Engineering: Principles and Practice*. John Wiley & Sons Ltd., 2nd edition, 2003. 66

[87] Jules White, Douglas C. Schmidt, and Aniruddha Gokhale. Simplifying Autonomic Enterprise Java Bean Applications via Model-Driven Development: A Case Study. *LNCS*, 3713:601–615, November 2005. 9, 65

[88] Jules White, Douglas C. Schmidt, and Aniruddha Gokhale. The J3 Process for Building Autonomic Enterprise Java Bean Systems. In *International Conference on Autonomic Computing (ICAC)*, pages 363–364, June 2005. 9

[89] Steve R. White, James E. Hanson, Ian Whalley, David M. Chess, and Jeffrey O. Kephart. An Architectural Approach to Autonomic Computing. In *International Conference on Autonomic Computing (ICAC)*, pages 2–9, May 2004. 7, 13

[90] Yan Zhang, Anna Liu, and Wei Qu. Software Architecture Design of an Autonomic System. In *Australasian Workshop on Software and System Architectures*, pages 5–11, 2004. 12