Evaluating Mission-Critical Self-Adaptive Software Systems: A Testing-Based Approach

by

Sen Li

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, 2010

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

Self-adaptive software is a closed-loop system that tries to manage, direct, or regulate its own behavior dynamically. Such a system aims at providing an automated and systematic approach to handling the increasing complexity of operation management. Mission-critical systems (e.g., e-business and telecommunication systems) are usually large, complex, and distributed. These systems must preserve their Quality of Service (QoS) at runtime under highly dynamic and non-deterministic conditions; therefore, they are suitable candidates for being equipped with self-adaptive capabilities. Although significant efforts have been devoted to modeling, designing, developing and deploying self-adaptive software since a decade ago, there is still a lack of well-established concrete processes for evaluating such systems.

This dissertation proposes a systematic evaluation process for mission-critical self-adaptive software systems. The process is a well-defined testing approach that needs a post-mortem analysis, takes the quantified QoS requirements as inputs, and comprises two main phases: i) conducting system-level testing, and ii) evaluating QoS requirements satisfaction. The process uses Service Level Agreements (SLAs) as quantified QoS requirements, and consequently as the adaptation requirements of mission-critical systems. Adaptation requirements are specific types of requirements used to engineer self-adaptive software. Moreover, for the first phase, the dissertation discusses the uniqueness and necessity of conducting system-level load and stress testing on a self-adaptive software system, for collecting runtime QoS data. In the second phase, the process makes use of utility functions to generate a single value indicating the QoS satisfaction of the evaluated system. The dissertation mainly focuses on evaluating the performance, availability and reliability characteristics of QoS.

An open source service-oriented Voice over IP (VoIP) application was selected as a case study. The VoIP application was transformed into a self-adaptive software system with various types of adaptation mechanisms. A set of empirical experiments was performed on the developed self-adaptive VoIP application, and the proposed process was adopted for evaluating the effectiveness of different adaptation mechanisms. To this end, the dissertation defines a sample SLA for the VoIP application, presents a report on the load and stress testing performed on the self-adaptive VoIP application, and presents a set of utility functions for evaluation. The experiments illustrate the validity, reliability, flexibility, and cost of the proposed evaluation process.

In sum, this dissertation introduces a novel evaluation process for mission-critical self-adaptive software systems, and shows that the proposed process can help researchers to systematically evaluate their self-adaptive systems.

Acknowledgements

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

Special thanks goes to Professor. Kostas Kontogiannis and Professor. Rudolph E. Seviora for their valuable time spent to read the thesis and for the valuable feedback they have provided.

I acknowledge my gratitude to all members of the Software Technologies and Applied Research (STAR) group, especially Dr. Mazeiar Salehie and Mehdi Amoui, for all their tremendous support, guidance, and thoughtful feedbacks throughout all my research.

Lastly, I would like to thank my parents for all they have done for me. I can not put my gratitude to them into words.

Dedication

This is dedicated to the one I love.

Table of Contents

Chapter 1

Introduction

Most software applications are implemented as open-loop systems, which need human oversight to function properly during their operation time [1]. However, with a dramatic increase in the complexity of software systems in the last two decades, management tasks for such systems have become more and more costly, time-consuming and error-prone, especially for large-scale distributed systems. Here are a few facts about nowadays IT operation management nowadays [2]:

- 30% of IT employment is for system administration.
- According to a study in 2002:
  - The labor cost is 3 to 8 times more than equipment costs in IT systems.
  - 1/3 to 1/2 of the total budget goes to human resources in IT.
- The IT operation management forecast for 2007-2011 is 18.1 billion dollars.

The main motivation of large blue-chip companies (e.g., IBM) to do research on *Self-Adaptive Software (SAS)* comes from the ever-increasing cost of operation management complexity [3]. SAS is a closed-loop system, which tries to manage, direct, or regulate its own behavior dynamically. Such a system aims at providing an automated and systematic way of handling the increasing complexity of operation management [4]. Generally, SAS consists of two main modules: the *Adaptable Software* and the *Adaptation Manager*. These two modules play the roles of a controlled plant and a controller, respectively. In SAS research, the major objective is to add runtime variability into software systems so as to provide and navigate heterogeneous adaptation changes. These adaptation changes can be categorized into four aspects: self-configuration, self-optimization, self-healing, and self-protection [5].

Self-adaptivity is mainly suitable for software systems that tend to operate in highly dynamic, non-deterministic, and complex environments. For mission-critical systems (e.g., electronic business and telecommunication systems), preserving their *Quality of Service (QoS)* at runtime under different conditions is important, or even vital [6]. Fowler in [7] defines mission-critical systems as "the ones where a hazard can degrade or prevent the successful completion of an intended operation". Moreover, supporting runtime adaptation is crucial for *mission-critical* systems, because shutting down and restarting such systems may incur unaffordable cost and risk [8]. Thus, mission-critical systems are promising candidates for incorporating self-adaptivity.

1.1 Problem Description

Significant efforts have been devoted to modeling (e.g., [9]), designing (e.g., [10]), and developing (e.g., [11]) SAS systems for about a decade. Scientists and engineers have proposed a variety of methodologies towards SAS system development. As surveyed in [12], several innovative adaptation mechanisms have been invented for controlling the runtime behavior of SAS systems. Meanwhile, a remarkable amount of new SAS systems have been implemented for addressing adaptivity in diverse aspects, such as performance, availability, reliability, and security.

However, there are still many challenges associated with evaluating, comparing, and benchmarking SAS systems or runtime adaptation mechanisms. Most existing research efforts have focused on the development process of SAS systems, and very few efforts have been made towards systematically evaluating these newly emerging systems. The community is eager to seek proper evaluation techniques or processes for such systems. It is noted that three main obstacles are associated with this issue:

- **Lack of Concrete Evaluation Processes:** We lack a well-established concrete process among researchers for evaluating SAS systems. A few attempts (e.g., [13]) have been made to propose feasible metrics for SAS evaluation. Yet, when discussing evaluation issues, people in this area sometimes neglect to answer a fundamental but essential question, "what is the SAS system built for?". Generally speaking, software is developed to fulfill its requirements, which address functional or non-functional concerns. Software systems cannot be properly evaluated without predefined requirements. For addressing the adaptivity concern, each SAS system is built based on special requirements, called the adaptation requirements [12]. Unfortunately, *Requirements Engineering (RE)* for SAS is still at an early stage in its development, and a widely-agreed definition or specification of adaptation requirements is not yet available [14].

- **Lack of Proper Case Studies:** We lack sophisticated and accessible applications that need runtime adaptation and can adapt at runtime. Engineering SAS is a challenging issue, and as pointed out in different works (e.g., [15, 12]), one of the obstacles associated with this issue is the lack of publicly-available and appropriate adaptable software applications as case studies. Such case studies are essential for evaluating different adaptation mechanisms, and for validating even the evaluation process itself. Therefore, we believe that having a well-engineered case study that needs runtime adaptation and has the capability of being adaptable, is a key requirement for studying the evaluation process.

- **Lack of System-Level Testing Processes:** We lack a well-defined process to conduct system-level testing on SAS systems. To evaluate a real system, its runtime log data should be obtained and analyzed. For example, if we want to evaluate system performance, the performance data of the system need to be logged. There are two ways to retrieve these kinds of runtime data: i) running the system in a real operational environment, or ii) simulating runtime workloads by system-level testing. For research projects, the first approach is usually expensive and impractical. In contrast, system-level testing is cheaper, and more flexible and popular. However, to the best of our knowledge, system-level testing on SAS systems for an evaluation purpose, has not yet been explicitly addressed.

These are the three main obstacles in developing an evaluation process for SAS research. In *Computer Science (CS)* or *Software Engineering (SE)* research, people do not believe in the existence of a silver bullet. We do not aim to provide an ultimate solution for addressing all existing issues. Instead, this dissertation only focuses on evaluating mission-critical SAS systems.

1.2 Thesis Contributions

Mission-critical systems have to maintain strict QoS guarantees at runtime [16]. These guarantees are normally specified in the contract signed by service providers and users, which is called the Service Level Agreement (SLA). The major contribution of this thesis is to propose a systematic process for evaluating mission-critical SAS systems. To this end, the contributions of this thesis for proposing such a process are summarized below:

- Propose a systematic evaluation process for mission-critical SAS systems. The process is a testing-based approach that needs post-mortem analysis. It proposes the use of SLAs as the quantified QoS requirements, and consequently as the adaptation

requirements of mission-critical SAS systems. Utility functions are applied to generate a single value indicating the satisfaction of the QoS requirements of the evaluated SAS system.

- Discuss explicitly the uniqueness and necessity of conducting load and stress tests on an SAS system for QoS evaluation.
- Survey the case studies previously used in SAS research, and discuss the desired characteristics of a case study for SAS research. In addition, a service-oriented VoIP platform is selected as our case study.
- Extract the architecture of the selected VoIP platform, and transforms the platform into an SAS system. Compared with most case studies implemented by existing research projects, this SAS system has the following merits: i) it is substantially larger and more practical, ii) it has a considerably greater number of meaningful adaptation scenarios with diverse sensors and effectors, and iii) it is flexible and easy to configure, such that new adaptation mechanisms can be added with minimum effort.
- Report on a set of empirical experiments that validates the feasibility of the proposed evaluation process in practical use. The set of experiments adopts the proposed process to evaluate different adaptation mechanisms on the developed self-adaptive VoIP platform.

1.3 Thesis Organization

The rest of this dissertation is organized as follows:

Chapter 2 provides the background concepts and a literature review of research related to this thesis. It first highlights the basic concepts of SAS systems, such as adaptation requirements, adaptation manager and adaptable software. Then, the concepts of software quality, QoS, and software system evaluation are discussed, as well as background knowledge on SLA, and load and stress testing. Finally, related work in evaluating SAS systems are surveyed.

Chapter 3 presents an overview of the proposed evaluation process. This process takes the quantified QoS requirements as input, and comprises two main phases: conducting load and stress testing, and evaluating QoS requirements satisfaction. Moreover, each phase of the process is elaborated in detail.

Chapter 4 describes the case study selected for this research. It first clarifies our expectation of an adaptable software application for use as a case study for SAS research,

and surveys other case studies previously used in SAS research. Moreover, it provides some background knowledge regarding the Voice-over-Internet-Protocol (VoIP). Next, we provide an overview of Mobicents, and assess it as a potential candidate platform for developing suitable case studies for SAS research. An example VoIP call controller application, deployed on Mobicents, is presented and selected as the case study. At the end, we discuss the suitability of our case study for SAS research, and transform the case study into a self-adaptive VoIP platform.

Chapter 5 reports on the experimental studies conducted for the proposed evaluation process. We conducted a set of experiments to evaluate the validity, reliability, flexibility and cost of the evaluation process. In the experiments, the proposed process was adopted to evaluate three adaptation mechanisms on the developed self-adaptive VoIP platform.

Chapter 6 finishes the thesis by drawing conclusions from the presented research. Furthermore, it highlights the research contributions and outlines some potential future directions that could be pursued from this research.

Chapter 2

Background Concepts and Related Work

This chapter provides the background concepts and a literature review of research related to the thesis. Section 2.1 presents a few concepts knowledge in the context of *Self-Adaptive Software (SAS)*. Section 2.2 discusses the definitions and issues regarding software quality and evaluation. The concepts of *Service Level Agreement (SLA)* and load testing are described in Section 2.3 and Section 2.4. At last, Section 2.5 surveys the related work in evaluating SAS systems.

2.1 Self-Adaptive Software (SAS)

The definition of self-adaptive software, given in a Broad Agency Announcement (BAA) provided by the Defense Advanced Research Projects Agency (DARPA), is as follows [17]: "Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible."

An SAS system, which is engineered to fulfill its adaptation requirements, embodies a closed loop and consists of two main components: a controlled plant and a controller. The reference architecture proposed by IBM in 2003 follows this basic composition and introduces the terms: autonomic manager and managed software [5]. The main rationale behind this architectural design is the separation of the adaptation logic (autonomic manager) and application logic (managed software). In some other works (e.g., [12]), autonomic manager and managed software are also called adaptation manager and adaptable software.

In this thesis, we follow the terminology of adaptable software and adaptation manager for the controlled plant and controller respectively [12]. These two entities are connected

through the sensor and effector interfaces provided by the adaptable software, as illustrated in Figure 2.1.

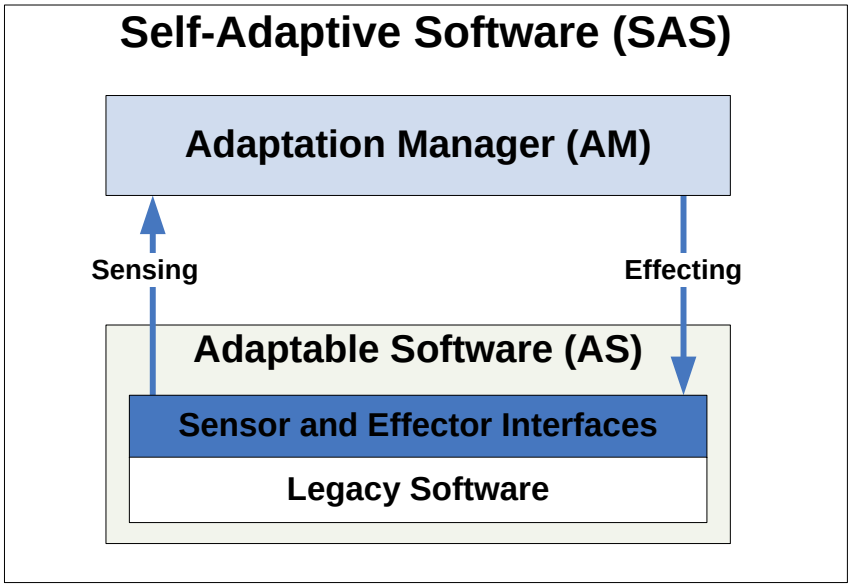

Figure 2.1: The Reference Architecture of Self-Adaptive Software [12]

This section presents some basic knowledge on self-adaptive software. Specifically, the concepts of adaptation requirements, adaptable software, and adaptation manager, which are used in the rest of the thesis, will be defined.

2.1.1 Adaptation Requirements *(ARs)*

In software engineering, we are dealing with an abstract world known as the *Application Domain (AD)*. A software system effects its *AD* by changing some phenomena in this domain [18]. A phenomenon is any observable occurrence in the world. Such phenomena are shared between the software system and its *AD*.

To engineer a software system, a collection of *requirements* is needed to address various functional and non-functional characteristics of the desired system. These requirements describe the effects the system should have on the *AD* in the form of changes to shared phenomena. Functional requirements specify the desired changes to those *AD* phenomena that are shared with the system through the application's input and output interfaces. The system observes a change in the *AD*'s phenomena via an input interface, and changes some shared phenomena via an output interface. In contrast, non-functional requirements specify changes to phenomena that are either: i) part of the *AD*, but are not shared

phenomena and are changed as a side-effect of the application's behaviour (e.g., Quality of Service requirements), or ii) part of other domains (e.g., business domain, test domain, and process domain).

Adaptation Requirements (ARs) are specific types of requirements that are used to engineer a software system that supports self-adaptive behaviors, so called an SAS system. Such a system can: (1) recognize the changes in its *AD*, (2) determine the required changes to be made to the system based on changes in the *AD*, and (3) make changes to itself in order to generate an alternative system behavior [19].

2.1.2 Adaptable Software *(AS)*

An *AS* is a legacy software with exposed and required sensors and effectors, which are specified in the *ARs*. Thus, we can define an *AS* as:

Given the ARs, a software system is considered to be an AS if and only if it exposes the required monitored data via its sensor interfaces, and exposes the required effecting operations via its effector interfaces.

To engineer the *AS* for the given *ARs*, we can either develop a new software application from scratch, or retrofit a legacy software system to comply with the required specifications of sensor and effector interfaces. In other words, adaptable software can be built in an engineering or re-engineering manner [20]. The *ARs* may change over time; a changed *AR* might require additional sensors and effectors, which can be added to the *AS* by a set of evolution and maintenance changes. In all cases, if the candidate software for self-adaptation is maintainable and reusable, it can be retrofitted to an *AS* more easily.

There are various techniques for realizing sensors and effectors. The most common set of sensors and effectors in *AS* is listed in Table 2.1 by Salehie *et al.* [12].

Sensors

A sensor, in an *AS*, is a component that measures various runtime metrics of the software (e.g., response time, throughput). These metrics might be stored into a place that can be checked periodically by the adaptation manager. Logging is probably the simplest and most widely-used sensing technique. There are many tools that support filtering, processing, and analyzing log files (e.g., the Generic Log Adapter and Log Trace Analyzer developed by IBM [21]).

The sensors of an *AS* may also be implemented by utilizing various existing models, protocols, and standards from other areas [12], examples of which are listed in the rows

Table 2.1: Different Techniques for Realizing Sensors and Effectors [12]

Entity	**Technique**	**Example**
Sensors	Logging	GLA (Generic Log Adapter), LTA (Log Trace Analyzer) [21]
	Monitoring & events information models	CIM (Common Information Model) [22], CBE (Common Base Events) [21]
	Management protocols and standards	Simple Network Management Protocol[23] , Web-Based Enterprise Management[24] , Application Response Measurement[25], Siena [26]
	Profiling	JVMTI (JVM Tool Interface) [27]
	Management frameworks	JMX (Java Management eXtension) [28]
	Aspect-oriented programming	BtM (Build to Manage) [29], JRat (Java Run-time Analysis Toolkit) [30]
	Design patterns	Wrapper (Adapter), Proxy, Strategy Pattern [31]
	Architectural patterns	Microkernel pattern, Reflection pattern, Interception pattern [32, 33]
	Autonomic patterns	Goal-driven self-assembly, self-healing clusters and utility-function driven resource allocation [15]
Effectors	Middleware-based effectors	Integrated middleware, Middleware interception [34], Virtual component pattern [35]
	Dynamic aspect weaving	JAC [36]
	Meta-object protocol	TRAP/J [37]
	Function pointers	Callback in CASA [38]

"Monitoring & events information models" and "Management protocols and standards" of Table 2.1. In addition, profiling techniques (e.g., Java Virtual Machine Tool Interface [27]) and enterprise management frameworks (e.g., Java Management eXtensions [28]) can facilitate the instrumentation of sensors.

Due to its flexibility, the use of AOP (Aspect-Oriented Programming) for sensing has been extensively studied. In Chan and Chieu's work [39], they propose to use AOP for the development of an *AS*, and claim that AOP's biggest advantage is that it can be used even when the application source code is not available. Blair develops a practical framework where dynamic aspects are used for monitoring [40]. For example, BtM (Build to Manage) [29] and JRat (Java Runtime Analysis Toolkit) [30] are two monitoring tools that make use of AOP.

Effectors

An effector, in an *AS*, is a component that can be used to produce a desired change in the software. Normally, it is more difficult to instrument effectors than sensors, because monitoring may at most cause some performance rather than changing any the system's functionality. However, effecting will not only cause additional cost, but will also influence the stability and functionality of the software.

By introducing a specific design pattern(e.g., wrapper, proxy, strategy [31]), developers can instrument effectors by applying source code transformations. One example is the work of Landauer *et al.*, which uses wrapper design patterns for developing an SAS system [41]. Furthermore, architectural patterns (e.g., microkernel, reflection, interception [32, 33]) and autonomic patterns (e.g., goal-driven self-assembly, self-healing clusters [15]) are also widely utilized for this purpose [12].

With respect to the generic reference architecture for SAS systems proposed in [5], the boundary of an *AS* with its domain can be narrowed down to a single module, or it can be extended to cover application containers or middleware. Therefore, we may consider an *AS* as an application or service, in conjunction with its constituent layers. This viewpoint enables us to take advantage of possible effecting facilities provided by the application's middleware. Thus, middleware-based effectors are an important set of effecting techniques as well.

Additionally, many studies have been carried out that use dynamic aspect weaving for implementing *AS*'s effectors. For example, Blair's framework allows adaptive behavior to be applied to systems by using a combination of dynamic aspect weaving, parameterization, and policies [40]. The framework applies the frame technology and can automatically generate effecting aspects at runtime. Other than this framework, dynamic aspect weaving was also been designed and/or used in different applications to achieve adaptability,

such as: J2ME applications [42], J2EE applications [43], conscientious software [44], web services [45] and OS kernels [46].

Finally, effectors may also be implemented using the meta-object protocol (e.g., TRAP/J [37]) and the function pointers technique (e.g., CASA [38]) [12].

2.1.3 Adaptation Manager *(AM)*

An *AM* is the key component of an SAS system, and is responsible for tracing software changes and making management decision accordingly. In operation time, an *AM* takes the *AS*'s runtime data as its inputs, and applies adaptation actions on the *AS* as its outputs. As shown in Figure 2.2, the four main obligations (a.k.a., MAPE loop) for an *AM* are [47]:

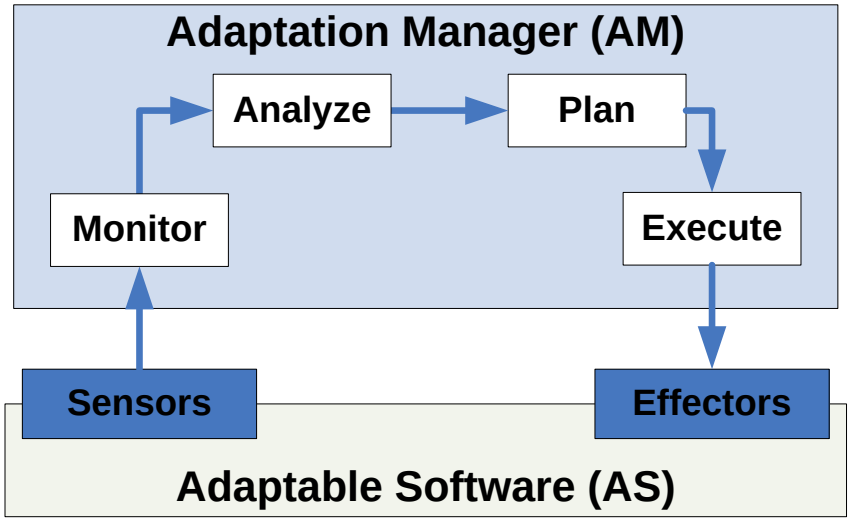

Figure 2.2: The MAPE Loop of Self-Adaptive Software (SAS)

- **Monitor:** An *AM* should collect and correlate runtime data through sensors provided by the *AS*. These data may be collected by periodic polling or by event-driven monitoring.
- **Analyze:** An *AM* should compute different metrics (e.g., performance metrics) based on the *AS*'s collected runtime data, or convert these data into behavioral patterns. Moreover, it should detect whether an adaptation change is required for the *AS* using the computed metrics or patterns.

- **Plan:** An *AM* should decide what adaptation actions need to be applied, and how to apply these actions in order to better satisfy the predefined service level objectives (i.e., *ARs*).
- **Execute:** An *AM* should apply the decided adaptation actions dynamically on the *AS*. This task can be accomplished by sending adaptation commands through the effectors provided by the *AS*.

An *AM* is built based on a particular adaptation mechanism, which is designed to properly capture the predefined *ARs*. The adaptation mechanism realizes the four aforementioned obligations of the *AM* [12]. Most existing adaptation mechanisms can be categorized into three groups:

- **Rule-based:** The adaptation logic in a rule-based *AM* (e.g., [48, 49, 50, 51, 52]) is formulated using condition-action rules (i.e., sets of if-then statements). The development of a rule-based *AM* is straightforward, but requires a deep understanding of the low-level details of system functions [53].
- **Goal-based:** At runtime, a goal-based *AM* (e.g., [54, 55, 56, 57]) makes decisions in order to achieve high-level goals, which in most cases are pre-defined Quality of Service (QoS) objectives. Compared to rule-based *AMs*, this type of *AM* can be tuned more easily based on its *ARs* [57].
- **Utility-based:** Making use of utility-based mechanisms (e.g., [58, 59, 60]) is an even higher-level way to reason about adaptation logic compared to the goal-based approach. A utility-based *AM* makes use of optimization theory, and computes realtime utility values on-the-fly, for the purpose of selecting the most beneficial action at a given time [60].

Goal-based and utility-based *AMs* are designed based on more advanced techniques and theories, whereas the rule-based approach is the most widely-used adaptation mechanism due to its simplicity and efficiency.

2.2 Software Quality and Evaluation

2.2.1 Software Quality

In the IEEE (Institute of Electrical and Electronics Engineers) standard glossary of SE terminology [61], *quality* is defined as: "The degree to which a system, component, or process meets specified requirements (or user expectations)." Requirements and expectations

are multidimensional concepts that reflect the diverse interests of the stakeholders. Thus, software quality comprises various characteristics, each of which can represent one quality aspect.

Many standards exist for software quality evaluation. ISO 9126 is probably the most famous standard and it has been widely adapted and applied in industry. It contains the following six characteristics [62]: *functionality*, *reliability*, *usability*, *efficiency*, *maintainability*, and *portability*. And as defined in [63], *security* can be considered as a part of *functionality*. Another well-known standard is IBM CUPRIMDA. The users' satisfaction plays a key role in the market driven quality strategy of IBM, which monitors user satisfaction in terms of seven characteristics [64]: *capability* (similar to *functionality*), *usability*, *performance* (similar to *efficiency*), *reliability*, *installability* (contains *portability*), *maintainability*, *documentation*, and *availability*.

The contents of these two standards overlap a great deal. In this thesis, we merge the characteristics of the two standards into a more general standard, which contains nine characteristics for evaluating software quality. They are: *capability*, *usability*, *performance*, *reliability*, *portability*, *maintainability*, *documentation*, *availability*, and *security*. Furthermore, Haines advocates categorizing software quality characteristics into three sets [65]: Functionality, Quality of Service (QoS) and Cost, as shown in Table 2.2.

Table 2.2: Three Sets of Software Quality Characteristics [65]

Functionality	Capability
QoS	Usability, Performance, Reliability, Availability, Security
Cost	Portability, Maintainability, Documentation

2.2.2 Quality of Service (QoS)

QoS is a well-known concept in the fields of computer networking and telecommunication networks. Gozdecki *et al.* state that "the term QoS is used in many meanings ranging from the users perception of the service to a set of connection parameters necessary to achieve particular service quality" [66]. Another definition of QoS from CISCO is "QoS refers to the capability of a network to provide better service to selected network traffic over various technologies" [67]. In addition, QoS management (or preservation) in networking and distributed systems is widely performed by using policy-based (i.e., rule-based) mechanisms [68].

Besides the networking and telecommunication fields, the concept of QoS is also used in other research areas. In SE, QoS often refers to non-functional requirements, and is usually linked with five high-level quality characteristics: *usability*, *performance*, *reliability*, *availability*, and *security* [65]. These characteristics are very abstract, and represent the various aspects of QoS from the highest level viewpoint. To actually evaluate the QoS of a software system, each of these characteristics needs to be quantified by low-level metrics. Table 2.3 presents some examples of low-level metrics for each of the high-level characteristics of QoS.

Table 2.3: Examples of Low-level Metrics for QoS Characteristics

Characteristics	**Examples of Metrics**
Usability	Efficiency in use, Learnability, User satisfaction [69, 70]
Performance	Response time, Throughput, Resource utilization [71]
Availability	Uptime percentage [71]
Reliability	Frequency/Severity of failure, Accuracy, Mean time to failure [72]
Security	Confidentiality, Data integrity, Non-repudiation [73]

The metrics listed above are only sample reflections of the five QoS characteristics. For a particular software application, the application domain should be taken into account, and the low-level metrics should be chosen from the specified domain. In other words, different systems may need different sets of low-level metrics for evaluating the same QoS characteristics.

2.2.3 Software System Evaluation

The IEEE defines *testing* as "the process of analyzing a software item to detect the differences between existing and required conditions (that is, to detect bugs) and to evaluate the features of the software items" [61]. From this definition, we can see that *testing* can be considered as one approach for *software system evaluation*.

In SE, the term *evaluation* is normally linked with performance, which is an aspect of quality. The IEEE definition of *computer performance evaluation* is "an engineering discipline that measures the performance of computer systems and investigates methods by which that performance can be improved" [61]. Generally speaking, the goal of evaluating

a software system is to determine "how good the software system is", or, in other words, "what is its *quality*".

Thus, by using the definitions of computer performance evaluation and software quality, *software system evaluation* can be defined as "an engineering discipline that measures the degree to which a software system meets specified requirements (or user expectations) and investigates methods by which that degree can be improved".

2.2.4 QoS vs. Cost

This thesis focuses on issues regarding software system QoS evaluation, whereas the concept of cost is as crucial as QoS in reality. More specifically, "improving QoS" is just the first half of the picture, and the other half is the cost of having this improvement. For example, when a manager wants to upgrade a system, she or he should have a clear idea about the cost and profit for this upgrade. It does not make any sense to spend $1 million for increasing the total net income by $100. In accounting, people name this concept "Return on Investment" (ROI), where $ROI = return\ profit/investment\ spent$. In engineering, we call it "Benefit/Cost" or "Cost-Effectiveness".

Many researchers (e.g., [74, 60, 53]) support this point of view, and state that the ultimate goal of developing SAS is to maximize the "Benefit/Cost" ratio. In this context, "benefit" means "improvement of QoS", and "cost" means how much it takes to replace the non-adaptive system by a self-adaptive system. Three metrics can be concluded from the literature for measuring the replacement cost:

- **Cost of conversion:** The cost of converting a system to be self-adaptive (or related to portability). This can be measured in monetary units or man/hour.
- **Extra overhead:** The extra overhead required to run the system in a self-adaptive manner (or related to maintainability). This can be measured using different performance metrics, such as extra latency or extra resource utilization.
- **Cost of maintenance:** The cost to maintain or modify the self-adaptive system after installation, for example, by adding a new policy to the existing system (or related to maintainability). This can be measured in monetary units or man/hour.

To evaluate an SAS system using the "Benefit/Cost" ratio, developers can: i) compute a value that reflects the users satisfaction in monetary units; ii) use the three cost metrics above to attain the total cost, and convert it into monetary units; and finally, iii) calculate the ratio.

2.3 Service Level Agreement (SLA)

Due to the exponential growth of Internet usage in the past decades, economic globalization has become an irreversible trend, and mission-critical systems are playing an essential role in our daily lives. These rising business systems have special requirements for QoS guarantees [75]. For each class of service, most e-business service providers agree to provide a certain guaranteed QoS level, which is specified by a formal contract, called an SLA.

2.3.1 SLA Properties

An SLA contains guarantee clauses signed between the service providers of mission-critical systems (e.g., e-business [76, 77] and telecommunications [78]) and their customers. These guarantees indicate what services need to be provided to users and how well they should be provided [79]. In other words, the system behavior both with respect to functionality and QoS aspects are guaranteed in such agreements.

Mutual service characteristics, such as usage, service level, runtime behaviors and penalties, are captured by SLAs from the perspectives of both service providers and service consumers [80]. A sample SLA from [81] contains the following three clauses:

- "Service availability should be greater or equal to 99%, weekdays 9a.m.−5p.m."
- "Service availability should be greater or equal to 95%, at all other times."
- "Availability metric is measured over each calendar month; penalty for SLA violation: refund to customer monthly fee."

An SLA explicitly defines the guaranteed QoS that a service provider should provide to its user. Some approaches even use the terms "SLA" and "QoS objectives" interchangeably [82]. Furthermore, there are three essential properties that an effective SLA should have [65]:

- **Specificity:** It must contain specific values for metrics. Fuzzy words like "throughput is low", or "error probability is high" should not appear in an SLA. To evaluate the QoS, we are looking for specific values that can be easily verified. Therefore, as a contract, an effective SLA should avoid the use of vague words.
- **Flexibility:** It must allow a measurable degree of flexibility. For example, a website may normally respond in 2 seconds, but it is almost impossible for a service provider to guarantee that it will always respond in 2 seconds every time. Thus, a clause saying "95% of the time the search engine responds in 2 seconds" makes much more sense.

- **Realism:** It must be realistic and reasonably achievable. Both the application business owner and the application technical owner should be involved and should play a key role in defining the SLA. An unrealistic SLA will give you more harm than benefit.

In one system, the same service may be associated with more than one SLA. This may be because the service provider may sign distinct contracts with different users to guarantee various QoS levels. For example, a web company could have three user levels: bronze, silver and gold. The company may promise that the response time of logging into its homepage for bronze, silver, and gold users is within 6 sec, 5 sec, and 4 sec, respectively. In this case, although the company provides only one service, three SLAs will be required to specify different contracts for ensuring the three QoS levels.

2.3.2 SLA Violation

Mission-critical systems need to preserve their quality attributes at runtime according to the SLA. Similar to any business contract violations, SLA violations (i.e., failing to meet the SLA) can cause serious financial consequences and penalties [65, 79]. Depending on the type of industry, the down-time cost per hour may range from thousands to millions of dollars. Like in credit card transactions, the average hourly down time cost is estimated to be 6.5 million dollars [83]. Thus, to maximize a company's revenue, SLA violations should be minimized.

There are two main ways to determine whether SLA violations occur [84]: i) "SLA Evaluation", which checks SLA violations only once for each accounting period, and ii) "Detection of SLA violation", which checks SLA violations more frequently and within the accounting period. From the perspective of customers, "SLA Evaluation" is good enough for calculating the total penalty at the end. However, no avoidance actions can be performed by service providers to minimize violations. In contrast, the "detection of SLA violations" approach is more proactive and may help to prevent or at least reduce the penalty.

The impact level (or penalty) for different types of SLA violations may be different. For instance, the impact of the whole system crashing for 1 hour is always more severe than a single request's response time exceeding 1 sec. We need measurement techniques to determine the seriousness of each violation, because: i) In the "detection of SLA violation" approach, if two violations occur simultaneously, the administrator should know which one he/she is of higher priority and needs to fix first, and ii) for the "SLA Evaluation" approach, it is necessary to know the penalty of each individual SLA violation, in order to calculate the total penalty incurred. Fortunately, several formulas exist for calculating the penalty of SLA violations (e.g., [84]).

2.4 Load and Stress Testing

Load testing is the process of emulating customer behavior at different load levels, and assessing the system quality under each load level [74]. A load level, corresponding to the request arrival rate, is the expected system workload described by an operational profile [85]. In short, load testing is one type of system-level testing, whose goal is to reveal functional and non-functional problems under various loads.

The execution time for a load test usually ranges from a few hours to several days. During the testing, one or more load generators send vast amounts (from tens to millions) of requests simultaneously for mimicking the behavior of customers. Meanwhile, the system is monitored and a large volume of runtime data is collected for further analysis [6, 85]. Load testing aims at simulating real runtime workloads to evaluate the system. Therefore, the types of runtime data collected during such tests should be comparable to the ones generated by real usages of the system. Thus, runtime QoS data regarding *performance*, *availability*, and *reliability* can be obtained by running load testing.

On the other hand, stress testing assesses the robustness, availability, and error handling of a system under extreme or unreasonable load levels [86]. It can be viewed as a special case of load testing, when the load level is set to be extreme.

2.4.1 User Patterns and Workloads

To mimic heterogeneous user activities at different load levels, a load or stress test should contain several tunable parameters. In [74], Menasce argues that three main important parameters may be varied during a load test: *workload intensity*, *workload mix*, and *customer behavior parameters*. Workload intensity, measured in *requests/sec*, indicates the arrival rate of user requests. When the value of the workload intensity is extremely abnormal, the load testing is called stress testing. In addition, diverse user behaviors are characterized by a workload mix (e.g., the combination of request types) and customer behavior parameters (e.g., abandonment threshold and think time).

In contrast, Haines suggests to use two, instead of three, parameters to tune a load test. He groups the workload mix and customer behavior parameters as one parameter called *user patterns*, and refers to workload intensity as *user loads* or *workloads* [65]. This thesis supports Haines' point of view. More specifically, we use *workloads* and *user patterns* as the two tunable parameters for a load test.

User Patterns Formalization

The most significant feature of a user pattern is its realness: i.e., how it reflects a real user's behavior. Many researchers have argued the importance of the realness of the representative user pattern:

- "If your load is not balanced and representative of actual end-user behavior, then you cannot have confidence that you have identified performance bottlenecks [65]."
- "A key concept in load testing is the notion of a virtual user. A virtual user emulates what a real user does. A load test is only valid if the behavior of virtual users has characteristics similar to those of actual users [74]."
- "Failure to mimic real user behavior can generate totally inconsistent results [87]."

User patterns may be attained in three ways [87]. For existing applications, either access log file analysis or end-user monitoring devices may generate helpful information for formalizing user patterns. For new applications without log files or real users yet, developers can only predict system usage through use case scenarios. After retrieving the paths and statistics of customer behaviors, we can use the Customer Behavior Model Graph (CBMG) to formally represent user patterns for re-generation later. Introduced in [83], CBMG is an interaction model used for modeling user patterns. It is basically a stochastic graph where nodes are states and arcs are transitions annotated by a probability value.

Workload Characterization

Another step of load testing is to determine the load levels, by characterizing the system's workload levels. Mission-critical systems, which need to handle user requests simultaneously, have strong requirements for scalability. Thus, as advocated in many approaches (e.g., [65, 83]), understanding and characterizing workloads is a crucial step in any QoS evaluation study of such systems. Workloads reflect the intensity of user access, and are usually measured as *request/sec* or *session/sec*.

To characterize the workload levels of a system, one should at least properly determine the expected load and the capacity of the system. The expected load is the expected workload that the system will face at runtime, and can be determined based on the predefined system requirements. On the other hand, the capacity is the maximum workload the system can accept before it enters the bucklezone, that is, before the system's saturation limit is reached and it starts crashing. The capacity of the system can be determined by gradually increasing the workload until the system resources are saturated.

2.4.2 Load Generator Configuration

To conduct load or stress testing, a powerful load (traffic) generator must be available. The selected load generator should be configured based on the formerly defined user patterns and workloads. Additionally, the load generator should be setup in an appropriate way to generate traffic. As stated in [83], there are two common approaches for generating traffic in load generators:

- **Trace-based approach**: Using traces from actual traffic, and either samples or replay traces to generate traffic [88].
- **Simulation approach**: Using mathematical models to represent the characteristics of the traffic, and then generating requests that follow the models.

We may be able to use both approaches together to produce better results. Section 2.4.1 has emphasized the importance of mimicing real user patterns, so it is better to apply the trace-based approach to recreate user patterns. However, unlike user patterns that are predictable, runtime workloads are highly variable and unpredictable. For example, the Page Views (PV) of a news website might suddenly rise by an order of 1000 in a few minutes, when some breaking news is announced. Using the trace-based approach may not always be possible or valid. Therefore, we believe the simulation approach is more suitable for determining the workload levels. The characteristics of simulated workloads can be presented by statistical distributions: more specifically, various Probability Density Functions (PDFs), including Lognormal Distribution, Pareto Distribution, and Exponential Distribution [83].

Some examples of free open source load generators, which support both approaches, are JMeter [89] and Grinder [90] for J2EE applications, and SIPp [91] for SIP servers.

2.5 Evaluating Self-Adaptive Software

Perhaps one of the earliest attempts at addressing the evaluation issues in Self-Adaptive Software (SAS) is [13], published in 2004. Several high-level attributes are given for evaluating an SAS, such as QoS, cost, granularity/flexibility, robustness, adaptivity, sensitivity, and stabilistation. These attributes guide researchers in designing customized low-level metrics, which can be measured and used for evaluation. However, [13] does not discuss any actual evaluation techniques for SAS.

In software system evaluation, the three most well-known basic evaluation techniques are *Analytical Modeling*, *Simulation*, and *Measurement* [71]. A number of considerations listed in Table 2.4 can help in selecting from these techniques.

Table 2.4: Criteria for Selecting an Evaluation Technique [71]

Criterion	**Analytical Modeling**	**Simulation**	**Measurement**
Stage	Any	Any	Postprototype
Time required	Small	Medium	Varies
Tools	Analysts	Computer languages	Instrumentation
Accuracy	Low	Moderate	Varies
Trad-off evaluation	Easy	Moderate	Difficult
Cost	Small	Medium	High
Saleability	Low	Medium	High

In SAS research, people propose various evaluation techniques based on the three basic evaluation techniques.

- **Analytical Modeling:** In [92], Litoiu proposes a comprehensive performance model, a workload model, and an analysis engine for SAS. Furthermore, a set of analytical performance models are presented in [58] by Bennani et al. Utility theory is applied in these performance models to calculate the utility value based on the response time, throughput, and probability of rejection metrics derived from SLA.

- **Simulation:** Using Analytical Modeling can only evaluate system performance. To evaluate other quality aspects, Simulation or Measurement techniques should be selected. In the SAS research area, building case studies using simulation tools (e.g., Mathlab) is not common, and is used in only a few approaches (e.g., [93, 57]). Simulation is cheap but provides less accurate results.

- **Measurement:** The Measurement approach is appealing, but has the drawback of being too costly for academic research [71]. Hence, most approaches in this field propose to combine the Simulation and Measurement techniques. In other words, their case studies are real software systems, but workloads are generated by simulation tools. Approaches that use the combination methods mainly focus on evaluating how one quality aspect (e.g., response time [1, 94]) is improved.

Other than evaluation techniques, only few evaluation processes have been proposed for determining the effectiveness of a SAS system in maintaining quality attributes. In [95], Cheng, Garlan, and Schmerl introduce a light-weight self-adaptive news website, Znn.com, which is a web-based client-server system with added self-adaptive capabilities. The system can self-configure its server pool size and content mode, in order to achieve better utility

value based on different quality objectives. Although the work proposes an evaluation process, their case study is simple. Moreover, their process may not be comprehensive and general enough for evaluating other systems.

To the best of our knowledge, there is still no agreed-upon concrete process for evaluating SAS systems. To propose such an evaluation process, three main issues need to be addressed: i) what are the SAS system requirements that can be used for evaluation? ii) how should system-level testing be conducted on SAS systems to collect runtime data? and iii) how should data analysis be performed for evaluating SAS systems? In the rest of this section, we will summarize how the first and third questions are addressed in the literature. For the second question, we could not find any related work, so we have tried to address this issue in our proposed evaluation process in the next chapter.

2.5.1 Adaptation Requirements for Evaluation

In Section 2.2.3, we defined *software system evaluation* as "an engineering discipline that measures the degree to which a software system meets specified requirements (or user expectations) and investigates methods by which that degree can be improved". Thus, the first obstacle for defining an evaluation process for SAS is determining what its system requirements are. Moreover, in Section 2.1.1, *Adaptation Requirements (ARs)* are defined as the specific types of requirements that are used to engineer a software system that supports self-adaptive behaviors. Therefore, to evaluate an SAS system, we need to measure the degree to which the system meets its *ARs*.

Unfortunately, Requirements Engineering (RE) for SAS is still in the early stages of its development, so a widely-agreed definition or specification of adaptation requirements does not yet exist [14]. However, many researchers propose to use some existing requirements documents as the source for *ARs*. To this end, such requirements documents need to have quantified clauses, so that they can be properly evaluated. Service Level Agreement (SLA) is one type of requirement document that is widely used as the main source for specifying *ARs* [96]. SAS systems built for better satisfying their SLAs are called SLA-driven SAS systems. For example, Bobroff *et al.* propose an algorithm for the dynamic placement of a virtual machine for managing SLA violations [97]. In addition, Gambi *et al.* develop a self-adaptive virtualized data center to protect its SLA.

2.5.2 Trade-Off Analysis for Self-Adaptive Software

In an SAS system, runtime data cannot be easily analyzed for evaluating the system, because a trade-off analysis often needs to be performed between several potentially conflicting goals [14]. For example, there are tradeoffs between QoS metrics (e.g., response

time vs. availability, response time vs. security) [98]. If one goal of an SAS system is to improve some QoS elements (e.g., reducing response time), the system may need to sacrifice another goal (e.g., increasing availability). Due to potentially conflicting metrics, managing or evaluating the total QoS satisfaction of a system is often not a straightforward task [99]. Instead of separately evaluating each metric, the QoS of an SAS system needs to be considered as a whole, by using utility functions.

From artificial intelligence [100] to economics [101], utility functions are widely recognized as a form of preference specification. A utility function is an objective function with several attributes, and can compute a utility value that may be expressed in any suitable unit (e.g., monetary units) [53]. Many studies have discussed how to use utility values to reflect Service Level Agreement (SLA) satisfaction (e.g., [102, 103, 81, 104]). Yet, none of these studies are in the SAS research area, nor do they use the utility values for evaluation.

In the field of SAS, utility functions allow stakeholders to map multiple (possibly conflicting) runtime metrics into a real scalar value, by which the software system can be evaluated [60]. The responsibility of the adaptation manager is to optimize this utility [99]. For instance, Walsh et al. in [53] state that, "Given a utility function, the system or component must use an appropriate optimization technique in conjunction with a system model to determine the most valuable feasible state and the means for achieving it. Typically, these means may include tuning system parameters or reallocating resources." Thus, utility theory has been used in many projects for managing (e.g., [105, 53]) or evaluating (e.g., [95, 106]).

2.6 Summary

This chapter has presented different background concepts and related work for the subject of this thesis. First, we described the SAS system, its relationship with adaptation requirements, and its two main components. Then, Section 2.2 defined the terms software quality, QoS and software system evaluation in this context. Furthermore, we overviewed some concepts regarding the SLA, and load and stress testing processes. Finally, the literature of SAS evaluation has been surveyed.

To the best of our knowledge, an agreed-upon concrete evaluation process for SAS systems still does not exist. In Chapter 3, we will propose an approach for mission-critical SAS evaluation. More specifically, an systematic process will be presented for evaluating mission-critical SAS systems based on their adaptation requirements.

Chapter 3

Proposed Evaluation Process

In Chapter 2, we stated that to evaluate a Self-Adaptive Software (SAS) system, the degree to which the system meets its *ARs* needs to be measured. Mission-critical systems have to maintain very stringent QoS requirements at runtime. In most mission-critical SAS systems, adaptivity is added for better satisfying the predefined QoS guarantees between service providers and users. Thus, QoS requirements are the main source for specifying the *ARs* of mission-critical SAS systems. As a result, we argue that the evaluation process of a mission-critical SAS system is a systematic approach for measuring the degree to which the system meets its QoS requirements.

The main contribution of this thesis is to propose a systematic process for evaluating mission-critical SAS systems. The process is a testing-based approach that needs a post-mortem analysis.

3.1 Overview of the Evaluation Process

Figure 3.1 depicts an overview of the proposed evaluation process, and the sub-steps for each main phase.

At first, the process requires the quantified QoS requirements of the evaluated system to be provided as inputs. To measure the degree to which the system meets its QoS requirements, these requirements should be first quantified. For example, assuming one requirement is that "the response time shall be low", it would be difficult to determine how well this requirement is satisfied, due to the unquantifiable adjective "low". Moreover, as Figure 3.1 shows, the quantified QoS requirements will be used as inputs in the later two phases.

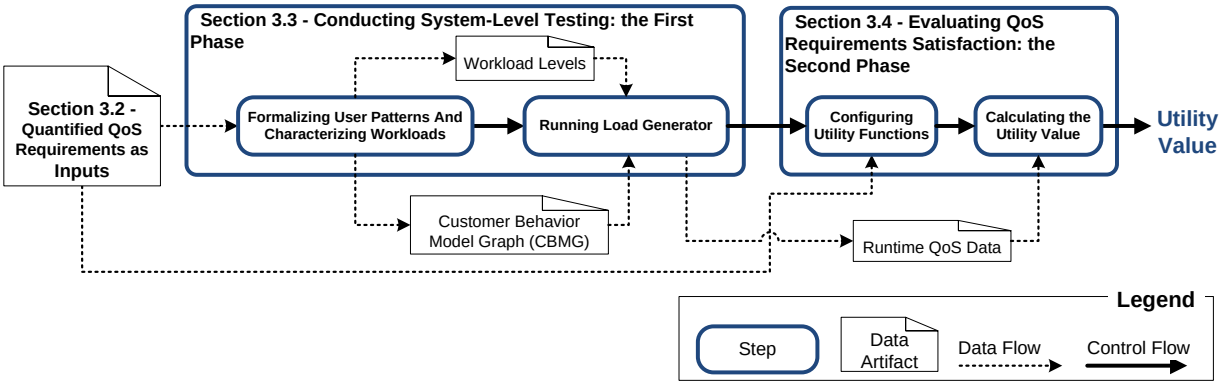

Figure 3.1: Overview of the Proposed Evaluation Process

After providing the quantified QoS requirements as inputs, the first phase of the process is to conduct system-level testing against the evaluated system. Only having the QoS requirements is not sufficient for evaluating the system. We also need to know how well the system operates against the requirements at runtime, by obtaining the runtime QoS data. To collect the data, we can either let real customers use the system, or conduct system-level testing to simulate user traffic. The latter method is definitely cheaper and more practical for most research-based projects. This process selects load and stress testing as the system-level testing techniques. To this end, user patterns and workloads should be formalized and characterized.

User patterns can be represented as the Customer Behavior Model Graph (CBMG) [83]. On the other hand, the workload levels should be determined based on the quantified QoS requirements. Finally, the load (traffic) generator will be configured and run in order to collect the runtime QoS data.

In the second phase, a set of utility functions should be selected and configured according to the quantified QoS requirements. These utility functions are used to compute the degree to which the system meets its QoS requirements. In other words, a utility value, which indicates the satisfaction level of the QoS requirements, will be calculated based on the quantified QoS requirements and the runtime QoS data. This utility value is the final evaluation result for the system.

The rest of this chapter will be dedicated to elaborating each phase of the evaluation process. Section 3.2 will discuss the suitability of using SLAs as the quantified QoS requirements. Section 3.3 will present a systematic process for conducting system-level load and stress testing against SAS systems. Finally, the process of evaluating QoS requirements satisfaction by using utility functions will be illustrated in Section 3.4.

3.2 Quantified QoS Requirements as Inputs

To engineer a mission-critical SAS system, QoS requirements should be specified for addressing various QoS characteristics of the desired system. Not only do these QoS requirements need to be explicitly defined, but they also have to be quantified. A collection of quantified QoS requirements is crucial for determining the degree to which the system satisfies these requirements (i.e., for evaluating the system). In practice, most enterprise mission-critical systems would quantify their QoS requirements, and represent them as Service Level Agreements (SLAs) [65].

SLAs, as formally-defined contracts, are strong requirements indicating the expectation of clients and the guarantee of providers with regards to QoS [65]. Usually, the application business owner (or the application product manager) is responsible for analyzing business use cases and client requirements in order to generate SLAs. Hence, theoretically, we can infer that customer needs will be satisfied if the SLAs are satisfied. Technically, because SLAs are the legal contracts between two parties, the service provider has no responsibility to pay any penalty if no SLA violation occurs. That is, as long as the SLA is satisfied, one can be confident of the system's QoS from both the user and finance points of view.

In [65], Haines claims that "You need a formally-defined SLA to accurately assess if a request is performing as expected. Without a formally defined SLA, your assessment is essentially a stab in the dark – the best you can do is assume that because a service request is taking a long time to run that it is problematic, without knowing how the request is supposed to respond." In conclusion, the SLA is essential for evaluating an enterprise mission-critical system. The QoS of the system can be evaluated, by checking how well it satisfies the SLA (i.e., how well the penalty of SLA violations is minimized).

In addition, the invention of SLA formal descriptive languages facilitates the evaluation process based on the SLA. As illustrated in Figure 3.1, the quantified QoS requirements need to be used throughout the rest of the evaluation process, specifically, for characterizing workloads and for configuring utility functions. Several formal languages have been developed for specifying SLAs, with the goal of expressing the content of SLAs in a proper and analyzable formalism [107]. For example, SLang proposed by Lanmanna *et al.* in [77] is one of the most famous XML-based SLA languages. Another example is an XML-based SLA language developed by Ludwig *et al.* from IBM [108]. Briefly, by specifying an SLA in a formal language, its content can be more easily parsed and used in various formulas (or utility functions) in order to automatically calculate the penalty of SLA violations.

Although using SLAs as *ARs* to develop mission-critical SAS systems is common and promising, certain mission-critical systems may not have SLAs. For those systems, other forms of quantified QoS requirements must be available. In that case, such alternate quantified QoS requirements could take the place of SLAs, and be used in our evaluation process.

3.3 Conducting System-Level Testing: the First Phase

Based on the five defined characteristics (*usability*, *performance*, *availability*, *reliability*, and *security*) of QoS, five types of runtime QoS data should be collected for fully evaluating the QoS of a system. Real runtime QoS data may be directly retrieved by letting real customers use the system for a period of time. This is, however, costly and not feasible in many situations. In most cases, system-level testing is conducted to collect these runtime data.

There are various types of system-level testing that can evaluate different QoS requirements, such as usability testing and security testing [109, 110, 111]. Security and usability are case-specific (or system-specific) attributes. For example, it is difficult to objectively argue if a system has a higher usability than another one. Due to their uniqueness, security testing and usability testing are out of the scope of this dissertation. Our proposed evaluation process only focuses on the QoS data related to performance, availability, and reliability.

Load and stress testing are two popular types of system-level testing. Supporting the concurrent access of hundreds or thousands of users is a common requirement for large-scale mission-critical systems [6]. Thus, load and stress testing should be conducted to ensure the scalability of these systems. Moreover, as discussed in Section 2.4.1, the three types of runtime QoS data (*performance*, *availability*, and *reliability*) can be collected by conducting load and stress testing.

3.3.1 Formalizing User Patterns and Characterizing Workloads

To conduct a load or stress test, two tunable parameters need to be determined: user patterns and workloads. Formalizing user patterns for mission-critical SAS systems is the same as the formalization procedure mentioned in Section 2.4.1. Thus, we will not discuss the procedure again here.

In contrast, workloads need to be elaborately characterized for conducting load testing against mission-critical SAS systems. As the workload increases, there are three checkpoints which should be given extra attention:

- **Expected Load Point (ELP)**: This is the expected workload level for the system.
- **SLA Violation Point (SVP)**: Starting from this point, the system will exceed its SLA. That is, the SLA must be violated if the workload is over the SVP.
- **Saturation Point (SAP)**: After this point, the system will enter the so-called bucklezone, where the system's saturation limit is reached. This will result in an increase in response time, a decrease in throughput, and resource saturation.

The ELP checkpoint may be predicted or measured by the administrator based on past experiences. In addition, the SVP and SAP checkpoints are often obtained via graduated load testing. More precisely, we can first configure the workload to climb to the ELP, and gradually increase the workload until the SLA is violated or the resources are saturated. The SAP is also regarded as the capacity of the system.

The SVP and SAP of non-adaptive systems are different from those of SAS systems. Usually, SAS systems have better scalability, so their SVP and SAP are higher than those of non-adaptive systems. In this thesis, we use the terms SVP and SAP to refer to the SVP and SAP of the non-adaptive system. Figure 3.2 illustrates the relationships among these three checkpoints.

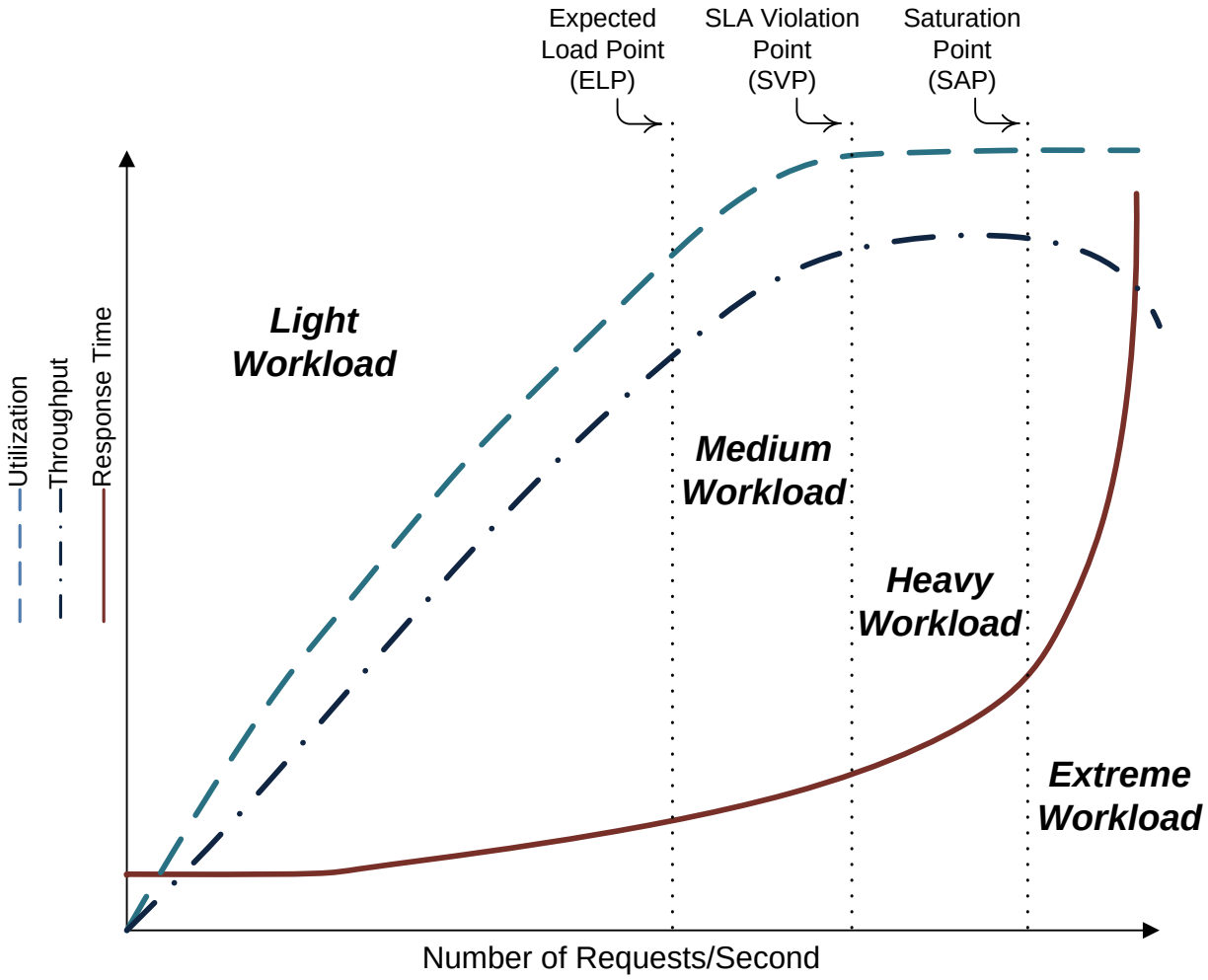

Figure 3.2: A Loaded Mission-Critical System follows this Typical Pattern [65]

To evaluate the QoS of a mission-critical SAS system, we argue that experiments should

be conducted under at least four workload levels: *light*, *medium*, *heavy* and *extreme*. The levels reflect the capacity of the non-adaptive system (i.e., the legacy system). In addition, load testing with extreme workload levels can also be viewed as stress testing:

- **Light Workload**: This workload is below the expected load point (ELP). The aim of evaluating the system under the light workload is to show that compared to a non-adaptive system, the SAS system will not cause dramatic overheads. When the workload is light, both non-adaptive and self-adaptive systems should be able to handle requests within the guaranteed QoS, without using any adaptation actions. In this case, the QoS difference of these systems can be explained as the extra runtime overhead for adding autonomy capability into the legacy system (e.g., monitoring overhead, and adaptation manager overhead).

- **Medium Workload**: This workload is between the expected load point (ELP) and the SLA violation point (SVP). Since the SVP has not been reached yet, the non-adaptive system should still work properly without violating the SLA. The test results of this workload can be used to determine whether the SAS system will generate too many false positives. In other words, the workload aims to test if the SAS system will wrongly decide to execute adaptation actions, even though no adaptation is needed.

- **Heavy Workload**: This workload is between the SLA violation point (SVP) and the saturation point (SAP). Hence, it will not crash the non-adaptive system but definitely decrease its performance. More precisely, the workload does not reach the saturation point, but violate the SLA. When the response time is high and the throughput is low, the system should decrease its service levels in order to increase its capacity. After the capacity is increased, the response time and throughput will remain normal. In short, the objective of this workload is to show that the SAS system can improve its QoS in contrast to the non-adaptive system.

- **Extreme Workload**: This workload is over the saturation point (SAP) and in the buckle zone. The non-adaptive system will crash under this workload while the SAS system should survive. Protecting the system from crashing under extreme abnormal conditions is one of the most remarkable advantages of adaptive systems. Famous examples are several major news websites that crashed on 9/11. By using service-level degrading, like what CNN (Cable News Network) preformed manually, such websites could have survived on that day and the week after [1]. This kind of abnormal situation can also be caused by malicious intruders through a "flooding attack". Therefore, the SAS system should be more robust than the non-adaptive system under extreme workload.

After user patterns are formalized and work loads are characterized, these two types of parameters can be used to configure the load generator.

3.3.2 Running the Load Generator

Finally, the configured load generator can be run for emulating user traffic against the SAS system, during which runtime QoS data will be collected for evaluating the system.

In conclusion, to conduct load or stress testing for evaluating the SAS system, our proposed process includes the following general steps:

- **Formalizing User Patterns**: The three formalization methods are: i) access log file analysis, ii) using end-user monitoring devices and iii) using reasonable predictions.
- **Storing User Patterns**: The formalized user patterns need to be stored in some data format for future re-generation. This can be accomplished by formally representing the user patterns information in CBMG or other formats.
- **Determining Three Checkpoints**: The three checkpoints (expected load point, SLA violation point, and saturation point) of the system need to be determined via graduated load testing.
- **Characterizing Workloads**: The four workloads (light, medium, heavy, and extreme) of the system need to be characterized based on the three checkpoints. Each of these workloads is designed to evaluate one aspect of the system.
- **Configuring the Load Generator**: The load generator needs to be configured based on the CBMG, appropriate probability density functions, and the characterized workloads. Then, the load generator needs to be run.
- **Recording Results**: The results of load or stress testing (i.e., the runtime QoS data) need to be recorded.

The recorded runtime QoS data will be used for evaluating how the system meets its QoS requirements. In other words, the QoS requirements and the runtime QoS data will be passed as inputs for computing the utility value in the next section.

3.4 Evaluating QoS Requirements Satisfaction: the Second Phase

As mentioned before, mission-critical systems must satisfy strong QoS requirements, which in most cases are SLAs. However, it is not easy to determine whether the SLA satisfaction is improved; that is, whether the SLA violations are minimized. For example, assume that there are two systems: a non-adaptive system A, and an SAS system B. The SLA includes the following requirements:

1. The average response time (RT) should be less than 6 sec

2. The average down time (DT) should be less than 1%

Assume that the runtime QoS data obtained for A are: average RT is 7 sec, and average DT is 0.5%; and those obtained for B are: average RT is 5 sec, and average DT is 2%. It is not straight-forward to determine which data satisfies the SLA better.

SAS systems are typically designed for managing and satisfying potentially conflicting goals. In this example, the actions of reducing RT and reducing DT may conflict with each other. The importance of utility functions in evaluating SAS systems has been mentioned in Section 2.5. In our proposed approach, utility functions are needed to fuse the obtained runtime QoS data into a single utility value, which the SLA satisfaction evaluation can be based on.

There are three main ways of creating QoS utility functions: i) by administration experts, ii) deriving them from a formal contract (SLA), or even iii) deriving them from another utility function [53]. A QoS utility function usually consists of two elements. The guaranteed values of QoS elements in the SLA will appear in the utility functions as constant values. And the values of runtime QoS data will be taken as input parameters. These utility functions, therefore, disclose the differential rates between guaranteed values and the obtained values.

Our evaluation process adopts the utility functions proposed by Menasce *et al.* in [99, 59, 60]. They claim that an SAS system should contain $n + 1$ utility functions, where the n functions correspond to n QoS elements guaranteed by the SLA. The last function is used to aggregate the n utility values in order to produce one global utility value, which reflects the total QoS satisfaction of the system. However, Menasce *et al.* have only applied their utility functions for implementing utility-based adaptation managers. In this thesis, we make use of these utility functions in the evaluation process rather than in the adaptation managers development process.

3.4.1 Configuring Utility Functions

For a specified SAS system, each of the utility functions should be configured based on the system's quantified QoS requirements (i.e., SLA). The four utility functions used in our evaluation process will be presented in this subsection. They are adopted from [60] and are configurable for different SLAs. Each of the first three utility functions is for calculating the utility value of one QoS element specified in the SLAs. The last one is for aggregating all calculated utility values in order to compute one single global utility value.

Utility Function for Response Time

The first utility function, $U_{RT}(rt)$, is for determining the *response time* utility value:

$$U_{RT}(rt) = \frac{K_{RT} \times e^{-rt+\beta_{RT}}}{1+e^{-rt+\beta_{RT}}}$$

where rt is the actual response time obtained at runtime, and β_{RT} is the response time guaranteed by the SLA. In addition, $K_{RT} = 100(1 + e^{\beta_{RT}})/e^{\beta_{RT}}$ is a scaling factor that makes $U_{RT}(0) = 100$. Figure 3.3 shows an example of $U_{RT}(rt)$ when $\beta_{RT} = 4$ sec, 2 sec, and 1 sec.

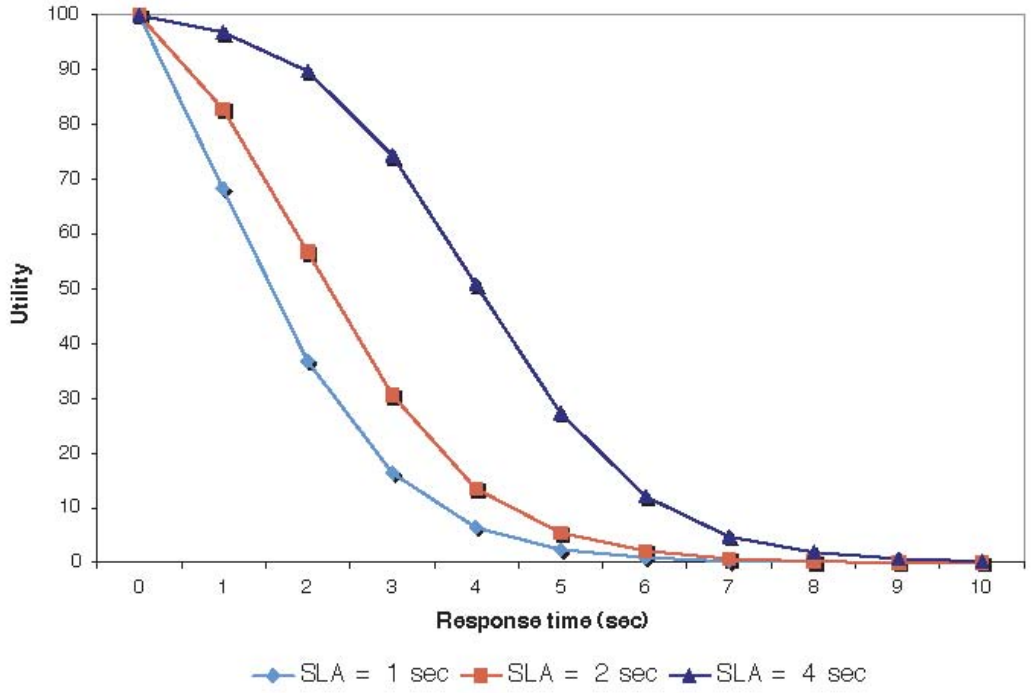

Figure 3.3: Utility vs. Response Time [60]

Utility Function for Throughput

The second utility function, $U_X(x)$, is for determining the *throughput* utility value:

$$U_X(x) = K_X \times \left(\frac{1}{1+e^{-x+\beta_X}} - \frac{1}{1+e^{\beta_X}}\right)$$

where x is the actual throughput obtained at runtime, and β_X is the throughput guaranteed by the SLA. Similar to K_R, $K_X = 100(1 + e^{\beta_X})/e^{\beta_X}$ is a scaling factor that makes $lim_{x\to\infty}U_X(x) = 100$. Figure 3.4 shows an example of $U_X(x)$ when $\beta_X = 4$ tps, 2 tps, and 1 tps (tps = Transactions Per Second).

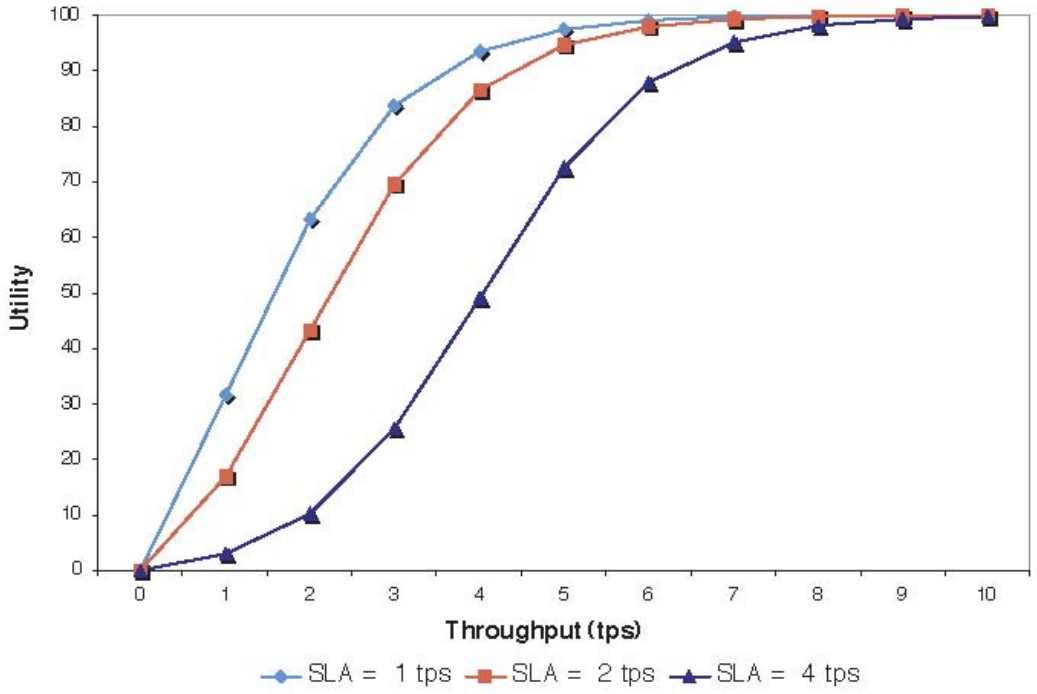

Figure 3.4: Utility vs. Throughput [60]

Utility Function for Error Probability (Error Rate)

The third utility function, $U_{EP}(ep)$, is for determining the *error probability (error rate)* utility value:

$$U_{EP}(ep) = \frac{1-ep}{\frac{1}{100}+\beta_{EP}\times ep}$$

where ep is the actual error rate obtained at runtime, and β_{EP} is the error rate guaranteed by the SLA. The maximum utility value computed by $U_{EP}(ep)$ is 100, when $ep = 0$. Figure 3.5 shows an example of $U_{EP}(ep)$ when $\beta(ep) = 0.05$, 0.1, and 0.2.

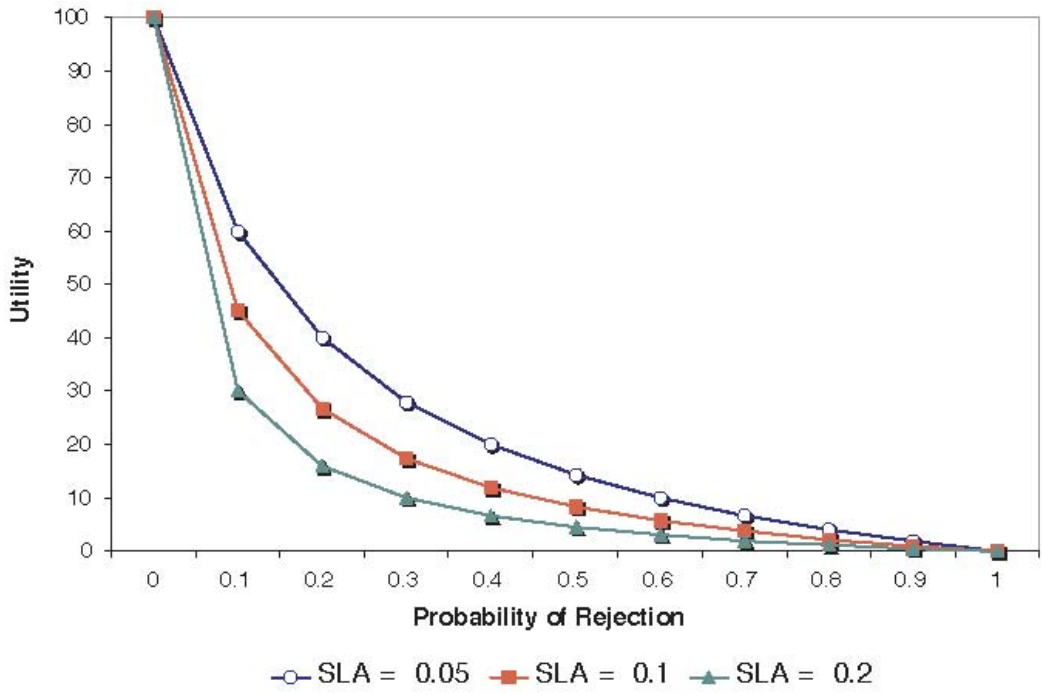

Figure 3.5: Utility vs. the Error Probability [60]

Utility Function for Global QoS

The last utility function is for determining the global utility value:

$$GlobalUtilityValue = w_1 * u_1 + w_2 * u_2 + ... + w_n * u_n$$

where u_i is the utility value returned by the utility function U_i, which represents the satisfaction of the i^{th} QoS element guaranteed by the SLA; and w_i is the weight for u_i, the magnitude of which reflects the importance of the i^{th} QoS element in the SLA.

Other Utility Functions

Although only four utility functions are shown in this section, many other QoS utility functions can be easily derived from these four. For example, any percentage-based probability metric (e.g., availability) may directly use the formula of $U_{EP}(ep)$, or a slight modified version of $U_{EP}(ep)$. Moreover, in many cases, utility functions are simply linear functions, like the utility function for the global QoS.

3.4.2 Calculating the Utility Value

After configuring all of the utility functions based on the quantified QoS requirements (i.e., SLAs), the recorded runtime QoS data from the first phase can be used as the inputs for the functions. Finally, a single global utility value is computed, which indicates the degree to which the system meets its QoS requirements. The global utility value is the final result of the proposed evaluation process.

Currently, this evaluation process is only suitable for mission-critical SAS systems. This is because, for evaluating other types of SAS systems (e.g., safety-critical), QoS requirements may not be the only source for specifying adaptation requirements. We may need to use other functional or non-functional requirements as inputs to the evaluation process. In addition, other types of system-level testing techniques may be involved, and more utility functions may need to be devised.

3.5 Summary

This chapter presents the proposed evaluation process for mission-critical SAS systems. The process takes the QoS requirements of the evaluated system as inputs, and consists of two main phases. First, load and stress testing is conducted against the system to collect the runtime QoS data. Second, the quantified QoS requirements and runtime QoS data are input to the utility functions, and one single utility value is computed. Finally, this utility value is used to evaluate the mission-critical SAS system.

In Chapter 4, an overview of a VoIP platform, Mobicents, will be presented. We select Mobicents as the case study for examining the proposed evaluation process. Furthermore, Mobicents will be transformed to become a self-adaptive VoIP platform that can dynamically enable and disable its services to better satisfy its SLA.

Chapter 4

Case Study

Selecting an appropriate case study for Self-Adaptive Software (SAS) research is important. Such an application is essential for evaluating different adaptation mechanisms, and for validating the evaluation process. One main contribution of this thesis is presenting a service-oriented Voice-over-Internet-Protocol (VoIP) platform as the case study, extracting its architecture, and transforming it into an SAS system.

This chapter first proposes the desired characteristics of a case study for SAS research, and surveys different existing case studies in this area. And, it provides some background concepts regarding VoIP. Next, we present an overview of Mobicents, and analyze it as a potential candidate platform for developing suitable case studies. Moreover, an example VoIP application, deployed on Mobicents, is presented as the case study. Finally, we will discuss the suitability of our case study for SAS research, and will transform it to become a self-adaptive VoIP platform.

4.1 Case Studies for SAS Research

The authors of [95] proposed a valuable list of properties for *Adaptable Software (AS)* as benchmarks. Besides benchmarking, *AS* for SAS research may be used for other purposes. Research on the engineering process, and cost and effort estimation for SAS systems are examples for other important applications of *AS* as case studies. Thus, we need a set of general characteristics for multi-purpose case studies. With respect to [95] and the stated background concepts of SAS, we can list the major characteristics of an appropriate case study as:

- **Having a meaningful *AD* for SAS:** As with any technology, self-adaptivity should not be treated as a silver bullet to boost the quality and performance of software sys-

tems. Adaptation is mainly suitable for software systems that tend to operate in highly dynamic, non-deterministic, and complex environments. Without a meaningful *AD*, it would be difficult to define proper *ARs* for the case study.

- **Facilitating sensing and effecting capabilities:** As we defined in Section 2.1.2, an *AS* should be able to expose the required sensors and effectors, specified by the given *ARs*, to *AM* through sensor and effector interfaces.
- **Being accessible:** To retrofit a current software system into an *AS*, we usually change the application's resources. Design documents, source code, or even binaries might be needed for applying necessary changes to candidate case studies.
- **Being well-supported:** Having good support means that there are professional teams for maintaining and updating the application, as well as its design documents. This ensures that we will stay updated on new application changes, and that we will focus our time on incorporating adaptability, but not on fixing application bugs.

The rest of this section will exhibit some case studies used in current SAS academic research, and try to analyze their advantages and disadvantages based on the above characteristics.

4.1.1 SAS Case Studies: State of the Art

In early SAS system research, the importance of the *Adaptable Software (AS)* was not well-understood. Most research efforts only focused on the *Adaptation Manager (AM)*, and the *AS* was treated as a side artifact, or even neglected altogether. Researchers would normally choose simple stand-alone programs to demonstrate their ideas. For example, [8] written by Oreizy *et al.* is one of the foundation papers for architecture-based runtime adaptation. The concepts and the *ArchStudio* tool suite proposed in the paper are very novel, but their case study, a *Cargo Routing System*, is just a simple logistics system.

However, the *AM* requires a plant to evaluate its functionality and effectiveness. Therefore, the community started to pay attention to benchmarking autonomic capabilities. Existing benchmarks for system performance are considered as potential candidates, and researchers have tried to renovate these classic benchmarks into *AS*. *TPC-W* defined by the Transaction Processing Performance Council (TPC) is perhaps the most widely known benchmark [15]. TPC-W, which implements a bookstore application in Java EE, is a typical web e-Commerce environment, designed for the performance evaluation of application servers [112]. Salehie *et al.* re-engineered TPC-W into an *AS*, with several sensors (e.g., response time) and effectors (e.g., changes to its search engine algorithms) [113, 1, 94]. They instrumented the sensors and effectors using Aspect-Oriented Programming (AOP).

Another famous benchmark for application servers is *SPECjAppServer*, defined by the Standard Performance Evaluation Corporation (SPEC) and used by [114] to implement a self-healing autonomic system. These benchmarks are mature open source projects, which are accessible and well supported. However, the main challenge in using them as AS is that they are not originally designed for self-adaptive software research. That is, these benchmarks are short of available sensors and effectors, as well as diverse adaptation scenarios. Hence, the effort of re-engineering them to be adaptable is great.

Other than transforming existing legacy applications, researchers have also tried to develop new AS from scratch. Cheng, Garlan, and Schmerl introduced a self-adaptive news website, *Znn.com*, in [95] to the community. Znn.com is a web-based client-server system, with added self-adaptive capabilities using the Rainbow framework. The system self-configures its server pool size and content mode to achieve the quality objectives. The design of Znn.com is concise, and is helpful for evaluating different adaptation mechanisms. More importantly, the whole project is made available online with detailed documentation. In our opinion, Znn.com seems to be easy to use and has all the defined characteristics to be a good case study. Nevertheless, Znn.com does not perform coarse-grained changes, such as component-level adaptations. In addition, it is not as complex as Mobicents applications in providing various adaptation scenarios. Thus, Mobicents applications might be more suitable for evaluating adaptation mechanisms, which require diverse sensors and effectors.

Nowadays, as more and more companies sing from the SOA songbook, building service-oriented adaptable software systems is becoming a trend. For example, authors of [115] demonstrate their adaptation mechanisms on a resilient service-based operating environment, *ServiceEcosystem*, which provides components for on-line monitoring and migration. The second example, called *Virtual Tour Guide (VTG)*, is proposed by Lorenzoli *et al.* in [116] to validate their self-adaptive approaches. Heterogeneous remote services (such as web services, grid services, or remote components) are integrated by VTG, to provide audio and video media-contents about a user-requested cultural tour. Besides VTG, Lorenzoli and other colleagues have also implemented another self-adaptive SOA-based system, *Personal Mobility Manager*, as the case study in [117]. These three examples are lightweight applications with limited sensors and effectors. Furthermore, there is no obvious evidence showing that they are publicly available.

In conclusion, most case studies used in current research projects may have one or more of the following disadvantages: i) need great customization efforts, ii) are relatively small and impractical, or iii) lack accessability and support.

4.2 Voice-over-Internet-Protocol (VoIP) Platforms

VoIP, including a set of transmission technologies, allows people to use the Internet for voice communication [118]. Two famous open VoIP technologies are the Session Initiation Protocol (SIP) and the Real-time Transport Protocol (RTP). As a text-based signaling protocol, SIP is widely applied in the creation, modification, and termination of multimedia communication sessions, such as voice, video, online games, and other services [119]. Furthermore, as stated in [120], "RTP provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video".

In contrast to the original telephones (Hardphones) that use the Public Switched Telephone Network (PSTN), VoIP-based phones (Softphones), which can initiate and receive voice and video communications over the Internet, are end-user-based clients that provide telephony services. As an infrastructure on which different services are deployed, a VoIP platform is the telephony service provider for VoIP-based phones.

4.3 Mobicents Platform

Mobicents is owned by Red Hat, and is the world's only open source communication platform certified for JSLEE (JAIN Service Logic Execution Environment) 1.0 compliance. JAIN stands for Java APIs for intelligent networks, and is developed by SUN Microsystems [121]. Mobicents is an event-driven platform, which facilitates the creation, deployment, and management of service-oriented applications [122].

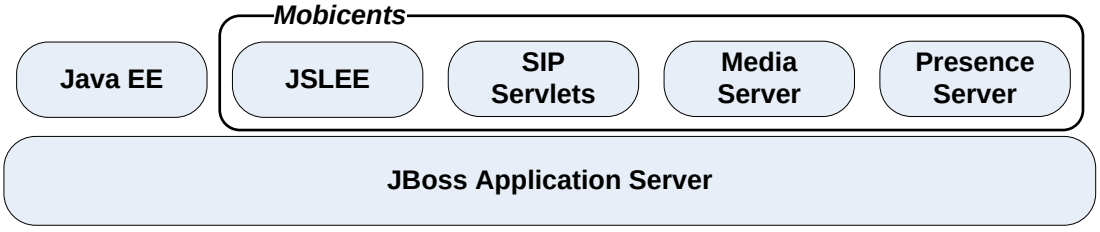

Figure 4.1: Top-level View of Mobicents [122]

As shown in Figure 4.1, the Mobicents project contains four main subprojects [122]: *Mobicents JSLEE*, *SIP Servlets Server*, *Media Server*, and *Presence Server*. Furthermore, Mobicents runs on top of the JBoss [123] application server [124], and uses four JBoss components: JMX Microkernel (for management), JNDI (for service registration), JTA (for transaction management), and TreeCache (for high availability state replication) [125]. Among the four subprojects, Mobicents JSLEE is the most suitable for adding autonomic

features. There are two main reasons: First, Mobicents JSLEE is the core of Mobicents and includes most of the business logic of Mobicents. Second, Mobicents JSLEE is designed with a loose-coupling architecture, and is built with adaptability. In section 4.3.1, Mobicents JSLEE will be discussed in detail.

4.3.1 Mobicents JSLEE

At the heart of Mobicents is its high-quality implementation of the JSLEE specification [126, 127]. Similar to Java EE, JSLEE is a component-based application execution framework. Nevertheless, JSLEE was not created to replace or even compete with Java EE. More precisely, event-driven applications (e.g., communication applications) have strict requirements for performance and availability, and such requirements are not covered by Java EE. JSLEE addresses these requirements thereby complementing the Java EE platform. That is, it aims at providing a high throughput and low latency execution environment for asynchronous event-driven applications. Therefore, JSLEE and Java EE can coexist seamlessly, and their integration leads to converged applications with rich services [121].

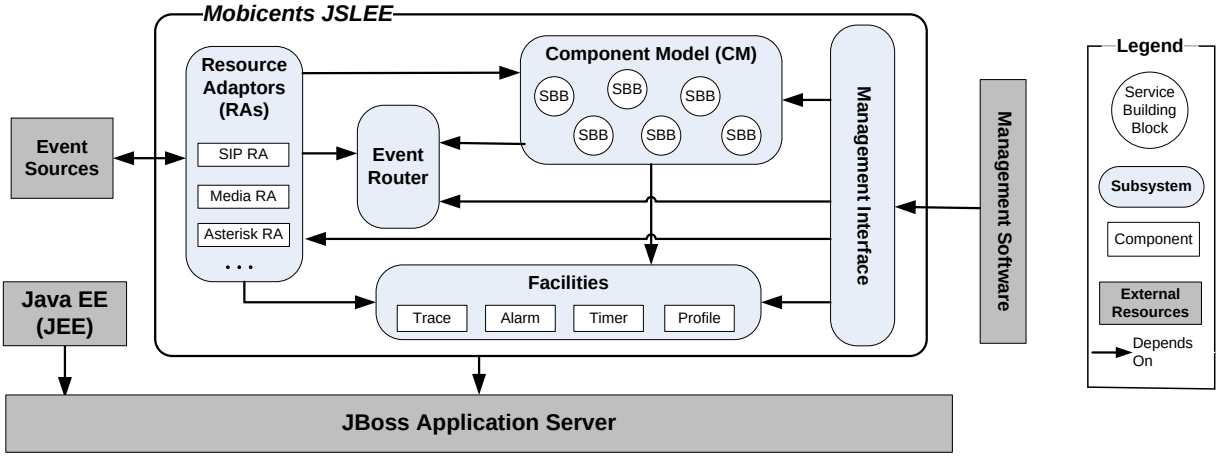

Figure 4.2: The Architecture of Mobicents JSLEE

The architecture of Mobicents JSLEE is presented in Figure 4.2. This architecture has been extracted based on design documents and source code [121, 122, 125, 127, 128]. As Figure 4.2 shows, Mobicents JSLEE runs on top of the JBoss application server, consists of five subsystems, and interacts with two external resources:

- **Component Model (CM)** is the container for *services*, each of which is bundled as a ".jar" file, and consists of one or more Service Building Blocks (SBBs). SBBs, which

are similar to EJBs (Enterprise JavaBeans), implement the business logic of the service and can naturally be integrated with Service Oriented Architecture (SOA) end points. Each SBB can accept certain event types, and has event-handler methods for processing events. In short, SBBs are event processing units that can collaborate with each other to provide various services.

- **Resource Adaptors (RAs)** are used to handle the low-level protocols (e.g., Internet protocols) for the high-level business logic (i.e., services). RAs can: i) receive incoming events emitted by *Event Sources*, ii) convert them into semantically equivalent Java events that the services can "understand", and iii) fire the events to the appropriate SBBs. In other words, RAs are bridges between SBBs and event sources.

- **Management Interface** enables external applications to manage SBBs and RAs via Java Management eXtension (JMX) [28]. As a management framework included in JDK, JMX technology is widely used in Java environments for managing and monitoring resources (e.g., devices and applications). In JMX, each resource is instrumented by one Java object called MBean (Managed Bean), which includes application code for managing and monitoring the resource. External applications can access MBeans via the JMX MBean server.

- **Event Router** redirects the typed events generated by the RAs to the SBBs, similar to a switch station.

- **Facilities** contain a number of utility methods used by the SBBs and RAs, such as Trace, Alarm, Timer, and Profile.

- **External Event Sources** can be any software or hardware that can emit events. The types of events (e.g., telephony events) that a system can accept reflects the application domain (e.g., telecommunications) that the system belongs to.

- **External Management Software** is an external component (e.g., adaptation manager) that connects to the *Management Interface* for monitoring and changing applications deployed on Mobicents JSLEE.

The collaboration among the RAs, Event Router, and SBBs is as follows: i) an RA receives an event from an external resource, ii) the RA reacts to the incoming event and produces a typed event to feed Event Router, iii) Event Router delivers the event to one SBB, which accepts this type of event, and iv) after the SBB finishes processing the event, it returns a response to the external resource via the RA. In short, SBBs can be thought of as event sinks that consume the typed events thrown from Event Router.

4.3.2 The Adaptivity of JSLEE Applications

A JSLEE application is a collection of services, each of which consists of several SBBs. Mobicents JSLEE has a low coupling and high cohesion architecture, which integrates with JMX and acts as a platform for service-oriented applications. More specifically, JMX provides a management interface, and SOA facilitates the dynamic composition of loosely coupled modules. These modules normally have well-defined contracts, and are configurable in supporting containers. These features provide flexibility in the composition, orchestration, and choreography of modules, which can reduce the additional cost of adding sensors and effectors. Thus, the Mobicents platform facilitates the sensing and effecting capabilities of its JSLEE applications.

Sensing Mechanisms

Every SBB of Mobicents is managed and monitored by the MBeans of JMX. Users may develop one MBean, which contains customized monitoring code (i.e., a bunch of "getter" methods), to attach to one SBB. The methods of each MBean are exposed through the JMX interface. The external management software can connect to this interface to access the MBeans, so as to monitor the system.

Besides creating customized MBeans, Mobicents JSLEE offers an easier way to create MBeans – by using *Usage Parameters (UPs)* [122]. During coding, developers can insert UPs into SBBs and use the predefined "setter" methods of UPs to manipulate them. During runtime, the predefined "getter" methods for the UPs will be exposed via MBeans, which are automatically created by Mobicents JSLEE. Briefly, the UP technique can save time in creating MBeans, at the expense of reduced flexibility in writing the customized code.

Effecting Mechanisms

Service boundaries in the Component Model are loosely coupled, and are remarkably flexible connection points for reconfiguration. Mobicents JSLEE benefits from such design, enabling the runtime change of services. As mentioned previously, each service, composed of one or more SBBs, is a well-defined and self-contained component. In Mobicents JSLEE, every service needs to be registered with the system and will be assigned a "ServiceID". At runtime, a predefined MBean, "ServiceManagementMBean", will be created by Mobicents JSLEE to manage these services. The external management software or administrator can invoke the "deactivate" or "activate" methods of ServiceManagementMBean through JMX, in order to enable or disable any service without restarting the system.

Besides services, many other resources (e.g., different server parameters) can be configured by other predefined MBeans at runtime. In addition to the predefined MBeans, developers can also create their own MBeans as the effectors. Just like using customized MBeans for monitoring, using customized MBeans for effecting could offer much more flexibility, so that many more adaptation scenarios can be created.

4.4 Call Controller 2 (CC2)

The previous section presented the architecture of Mobicents JSLEE, as well as its different technologies. Besides the high-performance JSLEE application server, the Mobicents project also contains some JSLEE applications. In this section, we present one service-oriented Mobicents application, and demonstrate how to re-engineer it into an SAS software system with the aid of these technologies. The selected application is a VoIP call controller system, called *Call Controller 2* [128].

4.4.1 The CC2 Architecture

Call Controller 2 (CC2) includes three SBBs and provides three VoIP-related services. Moreover, CC2 depends on another JSLEE application, *SIP Services Application (SSA)*, which comprises three SBBs and provides two more VoIP-related services [128]. The five services provided by CC2 and SSA are listed in Table 4.1, along with the SBBs supporting each service.

Table 4.1: Services Provided by Call Controller 2 and SIP Services Application

Application	**Service**	**SBB**	**Description**
Call Controller 2 (CC2)	Blocking	BlockingSbb	A callee can setup a blacklist indicating unwelcome callers. If a caller is in the list, this call will be blocked.
	Forwarding	Forwarding-Sbb	If a callee is unavailable but has a backup address, CC2 will forward the call to the callee's backup address.
	Voicemail	VoicemailSbb	If a callee is unavailable and has no backup address, but its voicemail is enabled, CC2 will record the caller's voice message.
SIP Services Application (SSA)	Register	LocationSbb Registrar-Sbb	Users can register and expose their IPs to the VoIP platform. After registering, the user is considered to be "online". The registration information is stored and managed by *LocationSbb*.
	Basic Call	LocationSbb ProxySbb	This is the most basic service for any VoIP platform. When a caller contacts a callee, *ProxySbb* will ask *LocationSbb* to locate the callee, and then establish a conversation channel between the two users.

Figure 4.3 is a zoom-in view of Figure 4.2, and is used to better illustrate the way these SBBs are located and connected in Mobicents. In summary, Mobicents is the platform

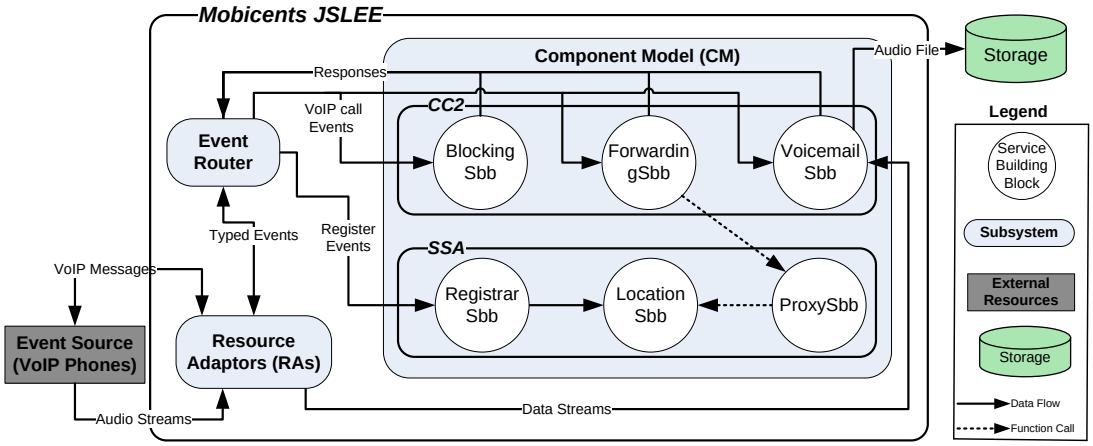

Figure 4.3: The Architecture of CC2 in Mobicents

where CC2 is deployed, and Mobicents plus CC2 as one entity, can be viewed as a typical VoIP platform that provides five services.

Each SBB in Mobicents has a delivery priority [128]. If several SBBs listen to one particular event, only one SBB can receive and process the event at a time. More precisely, these SBBs are linked like a service chain, where SBBs with higher delivery priorities are placed at earlier positions. When an event arrives, it will be delivered to the SBB with the highest delivery priority first. After being processed, the event will be delivered to the SBB with the second highest delivery priority, and so on.

The three SBBs of CC2 are independent and listen to the same incoming event. BlockingSbb has the highest delivery priority among the three, and so the event will be routed to BlockingSbb first, once the event is received. ForwardingSbb has the second highest delivery priority, and the delivery priority of VoicemailSbb is the lowest. Because every SBB of CC2 shares the same interface and each incoming event has to be processed by the SBBs sequentially, CC2 is designed with the Pipe-And-Filter architecture style. Thus, the system should still be functional, even if any of its SBBs are added, removed, or swapped at runtime. Two classes of events are listened to by this VoIP platform: Register-related events and VoIP-related events. Every Register-related event is handled by ReigstrarSbb and LocationSbb. On the other hand, when a VoIP-related event is received, it is handled by BlockingSbb, ForwardingSbb, VoicemailSbb, and ProxySbb.

Figure 4.4 illustrates how the six SBBs collaborate to provide different services after a typed event arrives. As shown in the figure, Register-related event is first sent to ReigstrarSbb, and LocationSbb will store the registration information. If the event is VoIP-related, it will be sent to and handled by BlockingSbb, ForwardingSbb, and VoicemailSbb one after another. Note that when an event arrives at ForwardingSbb, this SBB will first check if

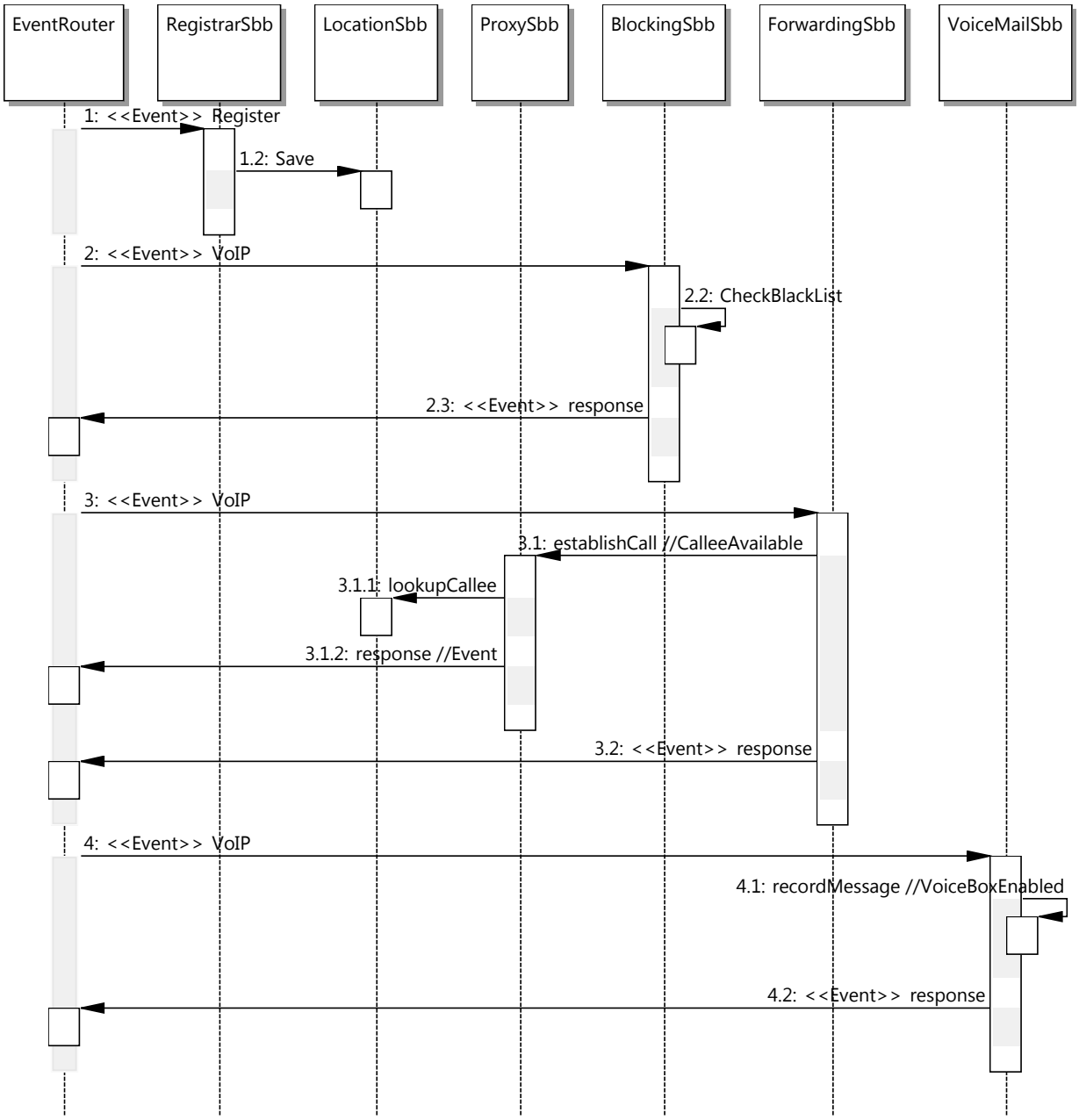

Figure 4.4: The Sequence Diagram of CC2

the callee is available. If it is, ForwardingSbb will invoke ProxySbb to provide the basic call service. Otherwise, ForwardingSbb itself will try to provide the forwarding service to the caller.

4.4.2 Sensors and Effectors for CC2

As mentioned in Section 4.3.2, Mobicents JSLEE facilitates the runtime sensing and effecting capabilities of service-oriented applications via JMX. Developers may use the existing sensors (Usage Parameters) and effectors (the predefined MBeans), or implement their own customized MBeans. Furthermore, Mobicents plus CC2 is a practical VoIP platform that supports quality tradeoffs. Therefore, a variety of sensors and effectors can be used for this VoIP application; some examples are listed in Table 4.2 and Table 4.3.

Table 4.2: Feasible Sensors of CC2 in Mobicents

Sensors	**Observable Attributes**
Customized MBeans	Response Time, Throughput, Arrival Rate, Error Probability, and Number of Time-outs for each service
	CPU and Memory usages of the system
Usage Parameters	Number of served Requests for each service
	Number of registered clients

These sensors and effectors are only samples to show the great adaptivity of the VoIP platform. We will not use all of them to build the SAS VoIP system. Moreover, depending on the different adaptation requirements of users, they can create even more sensors and effectors by JMX technology in addition to these.

Table 4.3: Feasible Effectors of CC2 in Mobicents

Effectors	**Controllable Actions**	**Type**
ServiceManagementMBean	Enable or disable each service.	Coarse-grained
Customized MBeans	Admission control for each service. That is, only certain users can access certain services.	Fine-grained
	Change the length of voicemail recording time.	
Predefined MBeans	Change the value of Jitter for recording voicemail.	
	Change the file format for recording a voicemail audio file (default is .wav).	

The meanings of the sensors in Table 4.2 are straightforward. It is worth pointing out that customized MBeans can implement any sensors provided by Usage Parameters, but not vice versa. For example, Usage Parameters cannot be used to monitor the CPU usage of the system. With respect to Table 4.3, we have discussed how ServiceManagementMBean works in Section 4.3.2. Having this predefined MBean is one of the most unique and powerful features of the Mobicents platform, because it enables runtime component-level (coarse-grained) adaptations for Mobicents applications. For fine-grained adaptations, people can use other predefined MBeans, or develop customized MBeans to tune some parameters of the system. For example, the voicemail audio file may be saved in the WAV format, or in other formats.

4.5 Mobicents for SAS Research

As discussed in Section 4.1, an appropriate case study for SAS research needs to lend itself to adaptation to be accessible and well-supported, and enable sensor and effector instrumentation.

Telephony systems intrinsically include a rich set of features and services. In some situations, adaptation changes help these systems to be available and to deliver services with a specific level of quality. Traffic bursts, which for example happened during hurricane Katrina, can cause a complete crash that leads to the unavailability of even the basic call service. This effect is not limited to huge disasters and was observed even during a small-scale event, such as a bridge collapse in Minnesota or a steam-pipe explosion in New York city [129]. In the literature, some researchers have mentioned communication as a potential domain for building adaptable applications. For instance, Oreizy *et al.* mentions that telephony is the application domain in which runtime adaptation is meaningful [8].

VoIP applications bring more flexibility to telephony systems and can be integrated with other media services for transferring data, voice, and video. A service-oriented architecture for VoIP applications enables us to adapt services at runtime even more, mainly due to the loosely-coupled services.

4.5.1 Mobicents Enabling Characteristics

Mobicents was selected as a candidate platform for a set of service-oriented adaptable software applications. In this section, based on our experience, we discuss why Mobicents seems an apt option for this purpose. Regarding the desired characteristics of adaptable software proposed in Section 4.1, the following items are advantages of using Mobicents as a platform for adaptable applications:

- *Supporting meaningful application domains for adaptability*: Mobicents is widely recognized as a rich open source Java platform for deploying VoIP services [125]. In addition, as a versatile platform with great extensibility, Mobicents can be applied to develop and deploy other applications in other domains. For instance, financial trading and online gaming, in which low-latency and event-driven properties are crucial, can be developed based on Mobicents facilities. Applications in these domains are other potential case studies for our purpose.

- *Providing sensors and effectors*: Mobicnets takes benefit from the management capabilities of the application server, namely the JMX framework. The JSLEE specification [127] emphasizes three reasons for integrating Mobicents with this management framework: i) JMX enables service management for JSLEE without the need for ad hoc, heavy, and hard-to-implement instrumentation; ii) JMX benefits from a component-based design, and consequently, it scales in providing solutions for large-scale projects, particularly in the communication domain; and iii) JMX instrumentation is protocol-independent, and can inter-operate with other management solutions, such as SNMP (Simple Network Management Protocol), TMN (Telecommunications Management Network), and WBEM (Web-Based Enterprise Management). As a component-based container for service-oriented applications, Mobicents integrates with the JMX framework in order to facilitate application runtime sensing and effecting. Both fine-grained (e.g., parameter tuning) and coarse-grained (e.g., service composition) adaptations can be performed.

- *Being an accessible and well-supported platform*: Mobicents is available as an open source project, and its owner, Red Hat, provides support for maintenance tasks, such as: i) tracking and fixing bugs, ii) the continual addition of new features, iii) providing and organizing the design documentation for such features, and iv) answering related questions. Furthermore, in future updates of Mobicents, more and more new applications will be developed and involved in the project, and researchers will have an even wider range of choices for their case studies. Thus, being a long-active open source project, which is officially supported by a company, is one of the most significant advantages of Mobicents.

In summary, Mobicents is a growing rich platform that in combination with complementing Java EE services, provides a strong basis for adaptable applications.

4.5.2 Mobicents Limitations

It is needless to mention that there are some limitations in using Mobicents for developing adaptable software:

- *Complexity*: Mobicents is a quite large and complex platform, and is still evolving to support an even richer set of features. Its latest version has 635k lines of code. Hence, before starting the development process, users need to spend a considerable amount of time to learn the concepts of JSLEE, becoming familiar with the platform, and setting up the environment.
- *Weakness of available applications on Mobicents*: The existing Mobicents applications were developed to assess the capabilities of Mobicents, and have not gone through rigorous testing processes. For example, the performance of CC2 is not as good as some other available VoIP systems, such as Asterisk.

Note that SAS systems are normally large and complex. Therefore, small and simple applications are not suitable candidates for experiments on adaptability. Also, Mobicents is rapidly growing, and more sample applications with higher quality will hopefully be available in the near future.

4.6 Developing Self-Adaptive Software System

After selecting Mobicents plus CC2 as our case study, we now present how to transform it into an SAS system. According to the stated concepts of SAS systems and our experiences, we define a process for developing our self-adaptive VoIP platform. This development process is shown in Figure 4.5. In this section, we will depict how to develop the system by following each step of this process.

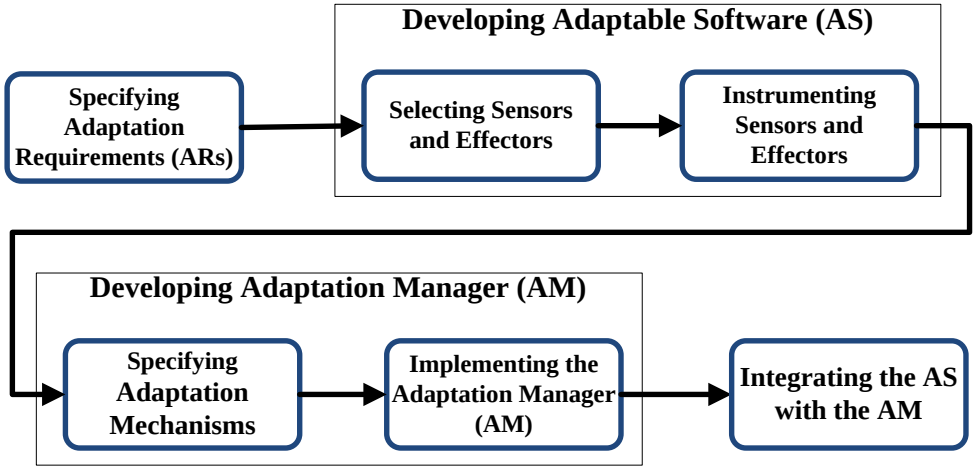

Figure 4.5: Development Process of our Self-adaptive VoIP System

4.6.1 Specifying Adaptation Requirements

An SAS system is developed based on its Adaptation Requirements (*ARs*). Furthermore, as we argued, Service Level Agreements (SLAs) are a main source for defining *ARs*. Thus, we will first define a sample SLA for our self-adaptive VoIP system and then specify the *ARs*. The SLA of the system is presented in Table 4.4. "SLA Violation Penalty" is the penalty that the service provider needs to pay the user after violating its SLA. "Profit" is the money the user needs to pay the service provide after being served.

Table 4.4: The Sample SLA for CC2

Scenario	**User Level**	**SLA Violation Penalty (cents)**	**Profit (cents)**
A user conducts a successful basic call.	Bronze	N/A	20
	Silver	N/A	40
	Gold	N/A	60
A user suffers a basic call response time longer than 6 sec.	Bronze	7	N/A
	Silver	18	N/A
	Gold	35	N/A
A user conducts a successful call forwarding.	Bronze	N/A	5
	Silver	N/A	10
	Gold	N/A	15
A user suffers a call forwarding response time longer than 0.1 sec.	Bronze	2	N/A
	Silver	4	N/A
	Gold	6	N/A
A user conducts a successful voicemail call.	Bronze	N/A	4
	Silver	N/A	8
	Gold	N/A	12
A user suffers a voicemail call response time longer than 1.5 sec.	Bronze	2	N/A
	Silver	4	N/A
	Gold	6	N/A
A user suffers the error probability of any service greater than 7%.	Bronze	20	N/A
	Silver	40	N/A
	Gold	60	N/A
A user receives a timeout from the voicemail service.	Bronze	1	N/A
	Silver	2	N/A
	Gold	3	N/A

As indicated in Table 4.4, this SLA classifies users into three levels: Gold, Silver, and Bronze. The profits (or penalties) of successfully (or unsuccessfully) serving users in different user levels are different. From the perspective of resource management, serving a request consumes some system resources (e.g., memory and CPU). The system may face

resource saturation if the workload is over the system's capacity. Consequently, the clients of all services will suffer degraded performance (e.g., high response time), and the non-functional requirements of all services will be hindered. If the workload is heavy, a better strategy might be re-configuring the system at runtime and disabling some non-essential or low-profit services, in order to give other services more resources to consume.

For example, under normal circumstances, all users of CC2 can access all services. Yet, when the workload is heavy, low level (i.e., Bronze) users can not access some/all services in the interest of preserving resources for serving higher-level users. Serving Gold and Silver users will be more beneficial than serving the same number of Bronze users. Additionally, the (non-essential) voicemail service can be disabled in order to increase the performance of the (essential) basic call service. Therefore, the SAS system should try to satisfy more high-level users by disabling some of the non-essential services of low-level users. Two sample *ARs* for the system could be:

- If the response time of basic call service for Gold users exceeds 6 sec, the system should behave in an adaptive manner by disabling the voicemail service for Bronze users.
- If the response time of basic call service for Gold users becomes less than 6 sec, the system should behave in an adaptive manner by enabling the voicemail service for Bronze users.

4.6.2 Adaptable Software (*AS*) Development

The two key components of an SAS system are the *Adaptable Software (AS)* and the *Adaptation Manager (AM)*. We defined the *AS* as following: "Given the *ARs*, a software system is considered as an *AS* if and only if it exposes the required monitoring data via its sensor interface, and exposes the required effecting operations via its effector interface." Thus, to transform a legacy application into an adaptable one, appropriate sensors and effectors need to be selected and instrumented.

Selecting Required Sensors and Effectors

Although Section 4.4.2 lists a set of feasible sensors and effectors for CC2, not all of them are necessarily related to the *ARs*. Therefore, we should make a selection based on the *ARs*, which are defined based on the defined SLA.

For runtime sensors, customized Managed Beans (MBeans) are created to measure the three main metrics (response time, throughput, and arrival rate) for each service for

each user level in CC2. All of these metrics are calculated based on the Moving Average technique [71]. There are also some metrics that are not used during runtime for adapting the system, but are required for evaluating the system. For example, the number of completed requests and the error probability for each service are important in determining how well the system behaved in the experiments. This information can be logged, so they are not considered as runtime sensed data. These metrics will be discussed later in Section 5.2.2.

For effectors, again, customized MBeans are created to enable/disable each service for each user level. The complete list of actions, which can be performed by the effectors, is presented in Appendix A. One may notice that the blocking service has no role in this action list. This is because disabling the blocking service will not improve the performance of the system. Blocking users from being served will decrease the number of threads in the system, and so the system can serve other users. Thus, there is no harm in allowing the blocking service to always be enabled.

Instrumenting Sensors and Effectors

To instrument MBeans as the sensors and effectors, we need to follow these stages:

- First, **create** a set of MBean classes (Java classes extending and implementing MBean interfaces), and **augment** them with the customized code for monitoring and effecting. For example, the following source code presents some parts of our MBean class for VoicemailSbb for collecting the Response Time (RT):

```
public class VoicemailMBeanImpl extends StandardMBean
                                implements VoicemailMBean{
    ...
  public void updateRT(long rt){
    ...
    RT_MovingAverage.addValue(rt);
    ...
  }

  public long getAverageRT(){
    ...
    return RT_MovingAverage.getValue();
  }
    ...
}
```

- Second, **attach** each MBean to its corresponding SBB (e.g., VoicemailSbb should invoke "updateRT" to update the voicemail call RT after a voicemail request is served).
- Finally, **register** (i.e., expose) the MBean to the JMX MBean server. To this end, we need to **obtain** an instance of the MBean server, and **register** the created MBean to the MBean server, as shown in the following code:

```
myEnv = (Context) new InitialContext().lookup("java:comp/env");
MBeanServer server = (MBeanServer)MBeanServerFactory.
                                    findMBeanServer(null).get(0);
vmMBean = new VoicemailMBeanImpl(myEnv);
ObjectName name = new ObjectName(VociemailServiceID);
server.registerMBean(vmMBean, name);
```

4.6.3 Developing Adaptation Manager (*AM*)

The ultimate goal of a mission-critical SAS system is to better satisfy the SLA. The main strategy of the *AM* is to prevent the system from entering the buckle zone: i.e., to exceed its nominal capacity. In fact, the adaptation actions increase/decrease the system capacity by playing with services settings. For example, when the response time is high and the throughput is low, the system should decrease its service levels in order to increase its capacity. After the capacity is increased, the response time and throughput return to normal. In summary, the objective is to show that the system can properly increase its capacity by sacrificing some non-essential requests from low level users. Under an extreme workload, the system may even block all of the services for a short time in order to prevent a crash.

Specifying Adaptation Mechanisms

The action selection mechanism, also called the *adaptation mechanism*, is the most crucial part of the decision-making component. As stated in Section 2.1.3, most existing *AMs* are implemented based on three types of adaptation mechanisms: rule-based, goal-based, and and utility-based. The first two types are more popular. For the experiment, we will develop one goal-based *AM* and one rule-based *AM*.

For the goal-based *AM*, we chose to implement a typical goal-based *AM* model, GAAM (Goal-Attribute-Action Model) [57]. GAAM is a multi-objective decision model that represents adaptation goals explicitly. In such a goal-driven model, goals can be presented as a hierarchy, from high-level and abstract stakeholder goals to low-level action goals. We

define twenty-eight low-level goals for CC2, listed in Appendix B. These goals are mentioned in our defined SLA. Each low-level goal has: i) its own preference over the actions listed in Appendix A, ii) a weight value (the value of p_i shown at the end of each line in Appendix B), and iii) an activation level (e.g., if response time is less than 7 sec, activate Goal 1). During runtime, the preferences of each activated goal will be aggregated by certain voting mechanisms (e.g., Plurality voting [130]). Finally, actions will be selected and performed based on the result of the voting.

For the rule-based *AM*, thirty-five action policies were defined for adapting the system towards the set of *ARs*. Each of the policies is a condition-action rule that indicates which action (listed in Appendix A) should be performed under certain situations. Because all of the policies are straightforward and very analogous, we will only present two examples here:

- If the voicemail service is enabled for Bronze users AND the basic call response time is greater than 6 sec for Bronze users, then disable the voicemail service for Bronze users.
- If the voicemail service is disabled for Bronze users AND the basic call response time is less than 6 sec for Bronze users, then enable the voicemail service for Bronze users.

Implementing the *AM*

The aforementioned adaptation mechanisms are implemented with the aid of the StarMX framework [113]. StarMX is a generic configurable framework designed for building self-adaptive Java-based applications. It is based on standards and well-established principles, and is a flexible approach for creating the adaptation closed loop. This framework uses a set of entities, called processes, to implement the adaptation logic using various mechanisms such as arbitrary policy languages. The closed loop is formed by the composition of processes, each of which supports one or more parts of the MAPE (Monitoring, Analyzing, Planning, and Executing) loop [5]. In short, our *AMs* are developed by StarMX, which makes use of the IBM ABLE (Agent Building and Learning Environment) [131] policy engine for executing goal-based policies and rule-based policies.

4.6.4 Integrating the *AS* with the *AM*

The last step is to integrate the *AS* with the *AM* via the sensor and effector interfaces. Because we use MBeans as the sensors and effectors, the sensor and effector interfaces are provided by JMX. To integrate StarMX with JMX, the StarMX configuration file should be prepared as shown in Figure 4.6. The *mbeanserver* tag is used to specify the lookup

```
<mbeanserver lookup-type="jndi" id="ms">
  <jndi-param jndi-name="jmx/invoker/RMIAdaptor">
    <property name="java.naming.factory.initial">org.jnp.interfaces.NamingContextFactory
      </property>
    <property name="java.naming.provider.url">jnp://localhost:1099</property>
    <property name="java.naming.factory.url.pkgs">org.jboss.naming:org.jnp.interfaces:org
      .jboss.naming.client</property>
  </jndi-param>
</mbeanserver>

<mbean id="VoicemailMBean"
  object-name="slee:SbbMBean=ServiceID[CallVoicemailService#org.mobicents#0.1]/SbbID[Call
VoicemailSbb#org.mobicents#0.1]" mbeanserver="ms"
  interface="org.mobicents.slee.examples.callcontrol.voicemail.VoicemailMBean">
</mbean>
```

Figure 4.6: Sample StarMX Configuration File

information of the MBean server, whose connection may be obtained via the Java Naming and Directory Interface (JNDI) lookup. Next, we need to declare the registered MBeans (e.g., VoicemailMBean) that we want to expose in the *mbean* tag. After specifying this information, StarMX will be able to locate and control the MBeans at runtime.

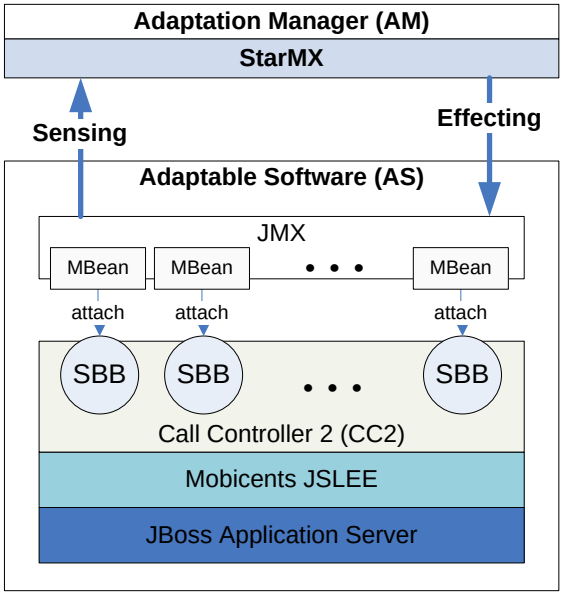

Figure 4.7: Self-Adaptive VoIP System

Figure 4.7 illustrates the high-level architecture of our self-adaptive VoIP system. In this figure, the JBoss application server is the underlying platform. Mobicents JSLEE is deployed on the JBoss application server. CC2, which consists of several SBBs, is deployed on Mobicents. Each SBB is attached to one MBean, which can expose the runtime metrics

of the SBB (i.e., sensors), and can make changes to the SBB at runtime (i.e., effectors). The *AM* is deployed on StarMX, and uses it to control the MBeans via JMX.

4.7 Summary

This chapter introduced our selected case study for this research. We elaborated the architecture of Mobicents and its enabling features regarding SAS research. The selected case study is a JSLEE application deployed on Mobicents, called *Call Controller 2 (CC2)*. The suitability of CC2 as an appropriate case study for SAS research was discussed. Finally, we transformed CC2 into a self-adaptive VoIP platform.

Chapter 5 reports on a set of empirical experiments that examine the proposed evaluation process. The set of experiments uses the proposed process to evaluate three different adaptation mechanisms on the developed self-adaptive VoIP platform.

Chapter 5

Experimental Studies

In the previous chapters, we have presented the proposed evaluation process, overviewed our case study, and demonstrated the development process of transforming the case study into a self-adaptive VoIP platform. In this chapter, a set of experiments will be presented for validating the feasibility of the proposed evaluation process in practical use.

5.1 Experimental Design

The goal of the experiments is to show that our proposed evaluation process can properly evaluate the QoS requirements satisfaction of different SAS systems. Specifically, we aim to answer the following two research questions in the experiments:

- ***Research Question 1 (RQ1)*:** Do the utility values, generated by the proposed evaluation process, reasonably reflect the QoS requirements satisfaction of different systems?
- ***Research Question 2 (RQ2)*:** Does adaptation have a positive impact on the operation of Call Controller 2 (CC2)? In other words, can adaptation improve QoS requirements satisfaction under different workloads?

To address ***RQ1***, three types of systems (one non-adaptive software and two SAS) were evaluated using the proposed evaluation process. Their runtime QoS data were analyzed manually in detail to see if each generated utility value reasonably reflects the QoS requirements satisfaction. To address ***RQ2***, we compared the runtime QoS data of non-adaptive and SAS systems as well as their utility values, to see if adaptation can improve QoS requirements satisfaction.

Therefore, the experiments have a one-factor design with blocking. Moreover, the experiments were conducted for each block three times (three replications). The experiment factor comprises the three following treatments: i)***No-adaptation (NA)***: no adaptation mechanism is enabled, ii)***Goal-based adaptation (GA)***: the goal-based adaptation mechanism is enabled, and iii)***Rule-based adaptation (RA)***: the rule-based adaptation mechanism is enabled.

That is, three types of systems were evaluated during the experiments: one non-adaptive system, one goal-based SAS system, and one rule-based SAS system. There are four blocking levels defined for the system based on different workloads. As mentioned previously, the effect of workload on adaptation effectiveness could be enormous. To evaluate the QoS of a mission-critical SAS system, experiments should be conducted under at least four workload levels: light, medium, heavy, and extreme. These four labels are based on the capacity of the non-adaptive system. Furthermore, the workload is designed based on the randomness of user behavior, in using the VoIP services via different paths. There are three services that users can access – basic call, forwarding, and voicemail.

To minimize the experimental errors due to sporadic events, three replications (experiments) were conducted for each block and each treatment. This also guarantees that experimental results can be subjected to Analysis of Variance (ANOVA).

5.2 Experimental Setting

The proposed evaluation process takes the quantified QoS requirements (i.e., SLA) as inputs, and comprises two main steps: conducting system-level testing, and evaluating QoS requirements satisfaction. To apply the proposed evaluation process, the following artifacts need to be available: i) the SLA of the evaluated system, ii) the user patterns and workloads for conducting load and stress testing, and iii) the utility functions for trade-off analysis.

The SLA of our evaluated system has been defined and shown in the previous chapter, and is used as the quantified QoS requirements. The rest of this section will discuss the user patterns, workloads, and utility functions that were used for the experiments.

5.2.1 User Patterns and Workloads for Experiments

The user patterns for the evaluated system were attained through reasonable predictions, and stored in the Customer Behavior Model Graph (CBMG). The portions of Bronze, Silver, and Gold users in any test case were 50%, 30%, 20%. Details of the workloads are given in Table 5.1. The workloads were generated based on the exponential distribution

with certain mean values (i.e., μ). In Table 5.1, "total calls" is the total amount of requests generated in the experiment.

Table 5.1: Inter-Arrival Time Distribution for each Service

Workload Level	**Basic Call**	**Forwarding**	**Voicemail**
Light Workload	Exponential distribution (μ=0.5 sec), total calls=4000	Exponential distribution (μ=5 sec), total calls=400	Exponential distribution, (μ=8 sec), total calls=240
Medium Workload	Exponential distribution (μ=0.15 sec), total calls=4000	Exponential distribution (μ=1 sec), total calls=700	Exponential distribution, (μ=1.6 sec), total calls=400
Heavy Workload	Exponential distribution (μ=0.075 sec), total calls=8000	Exponential distribution (μ=0.5 sec), total calls=1200	Exponential distribution, (μ=1.6 sec), total calls=400
Extreme Workload	Exponential distribution (μ=0.01 sec), total calls=8000	N/A	N/A

Note that there is no column for the registering service in Table 5.1. It is because a user must register with the platform before accessing any other services. Thus, the registering service is implicitly called by every user.

5.2.2 Utility Functions for Experiments

16 utility functions were involved in the evaluation process. The first 15 functions, called QoS utility functions, were corresponding to the 15 QoS elements guaranteed by the defined SLA. The last one, called global the utility function, was used to aggregate all utility values produced from the QoS utility functions. The 15 QoS elements can be divided into three groups:

1. the Response Time (RT) of each service (totally 3) of each user level (totally 3)
2. the Error Probability (EP) of each service (totally 3)
3. the Time Out (TO) number for the voicemail service of each user level (totally 3)

Runtime QoS Data for CC2

As mentioned in Section 3.4, the utility functions should first be configured based on the system's SLA. Then, the runtime QoS data needed to be passed in as input parameters. The following CC2 QoS data were collected during runtime:

- Basic Call (BC)
  - Basic Call RT: the individual RT for each successful basic call user in each user level - $rt^{BC}_{Gold}, rt^{BC}_{Silver}, rt^{BC}_{Bronze}$
  - Completed Basic Calls: the number of successful basic call requests in each user level - $\Phi^{BC}_{Gold}, \Phi^{BC}_{Silver}, \Phi^{BC}_{Bronze}$
  - Basic Call EP: the ratio of the number of basic call requests incorrectly served to the total number of Basic Call requests - ep^{BC}
- Forwarding (FW)
  - Forwarding RT: the individual RT for each successful forwarding user in each user level - $rt^{FW}_{Gold}, rt^{FW}_{Silver}, rt^{FW}_{Bronze}$
  - Completed Forwarding requests: the number of successful forwarding requests in each user level - $\Phi^{FW}_{Gold}, \Phi^{FW}_{Silver}, \Phi^{FW}_{Bronze}$
  - Forwarding EP: the ratio of the number of forwarding requests incorrectly served to the total number of forwarding requests - ep^{FW}
- Voicemail (VM)
  - Voicemail RT: the individual RT for each successful voicemail user in each user level - $rt^{VM}_{Gold}, rt^{VM}_{Silver}, rt^{VM}_{Bronze}$
  - Completed Voicemail requests: the number of successful voicemail requests in each user level - $\Phi^{VM}_{Gold}, \Phi^{VM}_{Silver}, \Phi^{VM}_{Bronze}$
  - Voicemail EP: the ratio of the number of voicemail requests incorrectly served to the total number of voicemail requests - ep^{VM}.
  - TO: the total number of Timeouts that force users to terminate recording voicemails, due to exceeding the maximum recording time - $to^{VM}_{Gold}, to^{VM}_{Silver}, to^{VM}_{Bronze}$

QoS Utility Functions for CC2

Let $U^s_{i,l}$ be the utility function for QoS element i, service s, and user level l, where $\forall i \in \{RT, EP, TO\}, \forall s \in \{BC, FW, VM\}, \forall l \in \{Gold, Silver, Bronze\}$. Although there are no particular utility functions designed for availability, this QoS element is taken into account implicitly. The total number of generated requests for each service is fixed in one experiment. Therefore, the number of completed requests (i.e., Φ) indicates the availability of the service.

The utility functions for TO (i.e., $U^{VM}_{TO,l}$) are simply linear functions. The utility functions for RT and EP were introduced in Section 3.4.1, and here, we rename them to

$U_{RT-Original}$ and $U_{EP-Original}$. The actual RT and EP utility functions used in this experiment (i.e., $U^s_{RT,l}$ and $U^s_{EP,l}$) are the modified versions of the original ones (i.e., $U_{RT-Original}$ and $U_{EP-Original}$). More precisely, the three types of utility functions are shown as follows:

$U^s_{RT,l}(rt) = \sum_{i=1}^{\Phi^s_l} U^s_{RT-Original,l}(rt^s_{l,i})$, where $rt^s_{l,i}$ is the obtained response time of the i^{th} completed request for service s and user level l.

$U^s_{EP}(ep) = \Phi^s * U^s_{EP-Original}(ep^s)$, where ep^s is the obtained error probability for service s.

$U^{VM}_{TO,l}(to) = to^{VM}_l$, where to^{VM}_l is the number of timeouts for user level l.

Global Utility Function for CC2

Let $u^s_{i,l}$ be the utility value computed by $U^s_{i,l}$, and let $w^s_{i,l}$ be the weight for $u^s_{i,l}$. The magnitude of $w^s_{i,l}$ is proportional to the penalty that needs to be paid if $U^s_{i,l}$'s corresponding QoS element in the SLA is violated. The global utility function is:

$$GlobalUtilityValue = \sum w^s_{RT,l} * u^s_{RT,l} + \sum w^s_{EP} * u^s_{EP} + \sum w^{VM}_{TO,l} * u^{VM}_{TO,l}$$

The global utility function uses several metrics from the system, but it misses at least one important metric: "what is the quality of voice in the communication?" Many studies in the area of VoIP communication have proposed metrics and indices for quantifying the quality of VoIP (e.g., [132, 133]). However, in our case, it was not possible to provide the essential hardware/software support to measure these metrics and indices. Incorporating the quality of VoIP in the global utility function can be a potential direction for extending this work.

5.2.3 Testbed

To conduct load and stress testing, the project used the most famous open source SIP scenario generator, SIPp 3.1 [91].

In VoIP systems, the caller and callee are normally referred to as the User Agent Client (UAC) and User Agent Server (UAS), respectively. Basic call, forwarding, and voicemail are the three different UAC behaviors. One running instance of SIPp can generate only one

UAC behavior. Thus, multiple SIPp instances are executed to mimic all of the behaviors. To simulate the different probabilities of each service being accessed, each of the SIPps will generate traffic based on the exponential distribution with different mean values. Only one UAS, which can receive messages from the different UACs, will be run.

One workstation and one server were used to generate load traffic and to run the Mobicents server respectively. The specification of the workstation is Windows XP professional SP3, Intel Pentium 4 CPU 3.4GHz, 2.00GB of RAM. The specification of the server is Windows Server 2003 Standard x64 Edition SP2, Intel Core 2 Quad CPU Q6700 @ 2.66GHz 7.92 GB of RAM. These two machines are connected via 100.0 Mbps Ethernet LAN.

5.3 Obtained Results

The obtained results (runtime QoS data and utility values) under the four workload levels are presented in this section. The reasons for conducting experiments under the four workload levels were explained in Section 3.3.1. Instead of showing the complete results, each subsection of this section will only present some key results obtained under one workload level. The presented data are:

- **Treatment:** As mentioned previously, No-adaptation (NA), Goal-based adaptation (GA), and Rule-based adaptation (RA) are the three types of adaptation mechanisms evaluated in the experiments.
- **Completed Basic Call, Forwarding, and Voicemail Requests:** The number of successfully completed requests for the basic call, forwarding, and voicemail services. For completed basic call requests, the number of successfully completed requests will be presented for each user level (Bronze, Silver, and Gold).
- **Average RT of Basic Call Requests:** The average response time of basic call requests.
- **Global Utility Value:** The global utility value computed using the utility functions discussed in the previous section.

From the defined SLA, we can learn that: i) serving a basic call request is much more profitable than a forwarding or voicemail request, and ii) serving a request from a Gold or Silver user can produce much more profits than a request from a Bronze user.

The experiments were conducted for each workload and each treatment three times (three replications). The shown values presented in the following subsections are the average among the three replications. In addition, the ANOVA test was applied on the global utility values of each workload.

5.3.1 Light Workload

The data in Table 5.2 show that when the workload is light, the performances of the SAS systems are nearly the same as that of the non-adaptive system. Each system can serve a similar number of completed requests for each service in each user level. Moreover, the response times and other metrics of the SAS and non-adaptive systems are very close. This means that no significant overheads are caused by the SAS systems when the workload is "light". The p-value of ANOVA for this workload is 0.8034. This is a strong indication that different treatments under the light workload have similar utility values.

Table 5.2: Obtained Results under Light Workload

Treatment	**Completed Basic Call Requests - Bronze**	**Completed Basic Call Requests - Silver**	**Completed Basic Call Requests - Gold**	**Average RT of Basic Call Requests**	**Completed Forwarding Requests**	**Completed Voicemail Requests**	**Global Utility Value**
NA	1882	1159	741	4:433 sec	332	207	132074
GA	1871	1161	742	4:337 sec	315	198	134219
RA	1886	1132	746	4:322 sec	335	200	135230

With respect to ***RQ1***, the computed utility values of the three systems are very close under the light workload. This matches our analysis of the runtime QoS data. Therefore, these utility values reasonably reflect the SLA satisfaction for each system. With respect to ***RQ2***, the three systems perform similarly under the light workload.

5.3.2 Medium Workload

As shown in Table 5.3, the performances of the three systems have no major differences for the basic call service. The number of completed basic call requests and the average response times are very similar for each system. Yet, compared with the GA and RA systems, the NA system successfully served much more forwarding and voicemail requests. To help interpret this result more clearly, Table 5.4 shows the number of completed forwarding and voicemail requests for each user level. The results indicate that the NA system completed remarkably more forwarding and voicemail Bronze requests than the GA and RA systems.

The reason is that an SAS system wants to preserve its SLA in advance, and so its threshold values for triggering adaptation actions should be more strict than what the SLA guarantees. In this case, the GA and RA systems might predict that the workload will gradually exceed the original capacity, due to the increasing workload trend. Therefore, the SAS systems blocked some forwarding and voicemail requests from low level users, in

Table 5.3: Obtained Results under Medium Workload

Treatment	Completed Basic Call Requests - Bronze	Completed Basic Call Requests - Silver	Completed Basic Call Requests - Gold	Average RT of Basic Call Requests	Completed Forwarding Requests	Completed Voicemail Requests	Global Utility Value
NA	1847	1140	750	4:469 sec	495	252	132343
GA	1818	1130	786	4:466 sec	295	180	129378
RA	1794	1152	789	4:501 sec	255	211	130097

order to preserve the QoS of the basic call service. These instances can be considered as false positives of the SAS systems. Although the global utility value of the NA system slightly outperforms the ones of the GA and RA systems, the p-value of ANOVA for this workload is 0.3432. This is a strong indication that different treatments under the medium workload have similar utility values. Hence, the impact of false positives is within an acceptable range.

Table 5.4: Completed Forwarding and Voicemail Requests under Medium Workload

Treatment	Completed Forwarding Requests - Bronze	Completed Forwarding Requests - Silver	Completed Forwarding Requests - Gold	Completed Voicemail Requests - Bronze	Completed Voicemail Requests - Silver	Completed Voicemail Requests - Gold
NA	226	150	119	133	79	40
GA	35	133	127	59	82	39
RA	24	108	123	75	91	45

Technically, such false positives are difficult to avoid. The SAS systems do not know if a medium workload test (meaning that workload will not gradually increase until the system crashes) is being conducted. Thus, the GA and RA systems will always try to do some prevention actions to preserve the SLA in advance, even though the workload will not eventually violate the SLA. The threshold values of the activating actions can be raised to decrease the impact of false positives. However, this may also increase the risk of causing more false negatives[1]. With carefully tuning, better threshold values can be found to balance the false positives and negatives of the system.

With respect to ***RQ1***, the runtime QoS data show that the Bronze forwarding and voicemail services of the NA system are served better than the GA and RA systems.

[1]False negative occurs when the adaptation mechanisms fail to prevent the system from SLA violations or crashing.

Because the Bronze forwarding and voicemail services are not very profitable, the utility value of the NA system is only slightly better than those of the GA and RA systems. Thus, we believe that utility values reasonably reflect the SLA satisfaction for each system. However, the QoS differences among these three systems are not significant. Hence, With respect to ***RQ2***, three systems still perform similarly under the medium workload.

5.3.3 Heavy Workload

The results in Table 5.5 show that the SAS systems can successfully serve more basic call requests with a lower average response time. Especially, the number of completed Silver and Gold basic call requests in the SAS systems are considerably higher than in the NA system. In contrast, the forwarding and voicemail services in the NA system have better availability. This is because at runtime, the SAS systems detect that the workload is heavy and that the QoS guarantees might be violated. Rather than allowing all requests to enter the busy system and drag down the overall QoS , the SAS systems performed self-optimizing actions to preserve the QoS of more profitable services and high level users. More specifically, the SAS systems block some non-essential requests (i.e., forwarding and voicemail) or requests from low level users (i.e., Bronze), in order to let other essential services and users consume more resources.

Table 5.5: Obtained Results under Heavy Workload

Treatment	**Completed Basic Call Requests - Bronze**	**Completed Basic Call Requests - Silver**	**Completed Basic Call Requests - Gold**	**Average RT of Basic Call Requests**	**Completed Forwarding Requests**	**Completed Voicemail Requests**	**Global Utility Value**
NA	3496	2110	1407	8:028 sec	557	157	175477
GA	3612	2265	1542	5:833 sec	209	95	238701
RA	3515	2304	1536	6:304 sec	175	88	221165

However, a higher number of completed requests and lower average response time may not necessarily mean that the QoS of the basic call service in the SAS systems is better satisfied than in the NA system. The defined SLA guarantees that the response time of each basic call request should not be over 6 sec (i.e., 6E+09 nanosec). Therefore, we need to examine the response time of each individual request for each user level. Figure 5.1 illustrates the response time of each basic call Gold request in the NA and GA systems.

As we can see from Figure 5.1, the GA system can serve more requests, most of which suffer a response time of around 6 sec. In contrast, in the NA system, the response time frequently changes dramatically, and is over 6 sec in most cases. Therefore, this figure

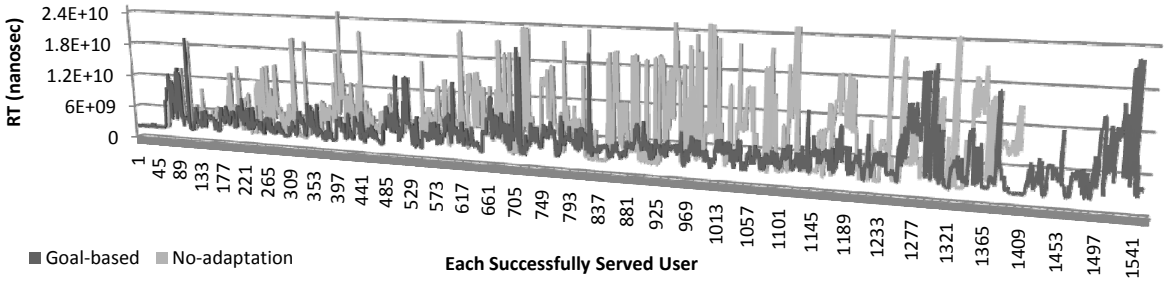

Figure 5.1: Response Time of Each Basic Call Gold Request

verifies that the GA system can better preserve the QoS of the basic call service for Gold users. Figure 5.2 and 5.3 illustrate the response time of each basic call request for Silver and Bronze users.

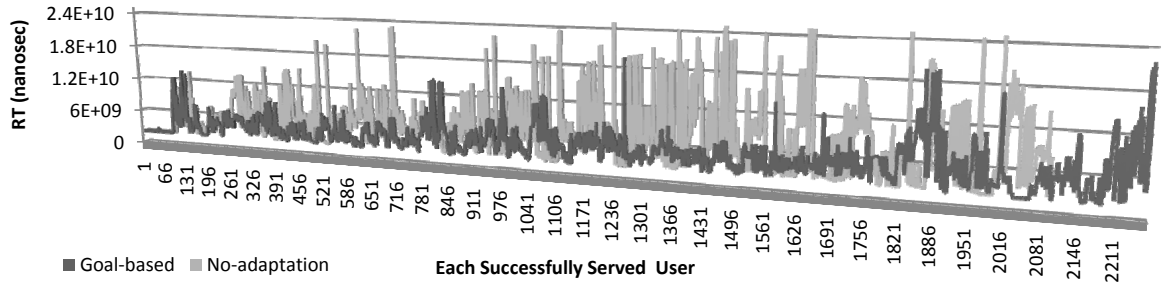

Figure 5.2: Response Time of Each Basic Call Silver Request

Both Figure 5.2 and 5.3 indicate that the GA system also outperforms the NA system. The RA system performed similarly to the GA system, and so the results of the RA system will not be presented here.

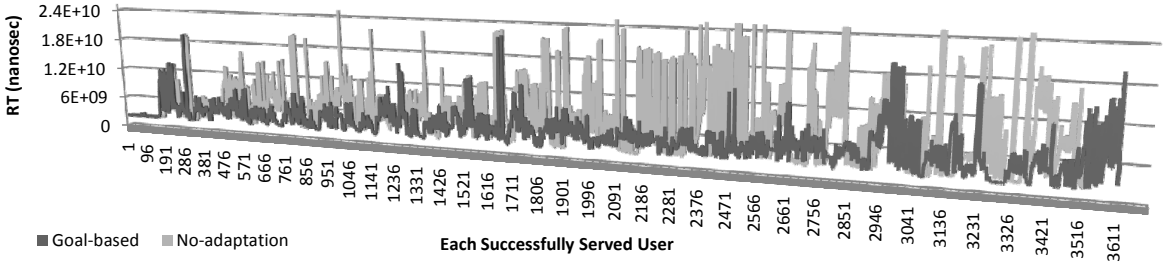

Figure 5.3: Response Time of Each Basic Call Bronze Request

As stated in the SLA, the basic call service is more important and profitable than the forwarding and voicemail services, so improving the QoS of basic call service by sacrificing

the QoS of forwarding and voicemail services should increase the total QoS satisfaction.

With respect to ***RQ1***, the utility values in Table 5.5 support our point of view; that is, the utility values of the GA and RA systems are higher than that of the NA system. The p-value for this workload is 0.0298, which is below 0.05 (95% confidence interval). This is a significant indication that different treatments under the heavy workload have different utility values. Thus, we believe that the utility values reasonably reflect the SLA satisfaction for each system. Note that with respect to the GA system seems to have slightly outperformed the RA system, but the difference is not significant. With respect to ***RQ2***, the GA and RA systems significantly outperform the NA system under the heavy workload. In other words, adaptation has a positive impact on the operation of CC2 under the heavy workload.

5.3.4 Extreme Workload

The purpose of conducting load tests under extreme workload is to prove that the SAS systems have a higher survival possibility than the NA system in abnormal situations. Here, we only want to know whether the adaptation mechanisms can prevent the system from crashing, when the workload is over the capacity of the system. Thus, no utility value needs to be calculated for this workload. The results in Table 5.6 show that adaptivity can successfully protect the system. That is, neither of the GA or RA systems crashed under the extreme workload, but the NA system crashed during the testing.

Table 5.6: Obtained Results under Extreme Workload

Treatment	**Crashed or Not?**
NA	Yes
GA	No
RA	No

Because no utility values were computed for the extreme workload, ***RQ1*** is not discussed for this workload. With respect to ***RQ2***, the GA and RA systems prevented the system from crashing, so they had a positive impact on the operation of CC2.

5.3.5 Lessons Learned

The type of workload plays an important role in evaluating the effectiveness of adaptation treatments. The blocking design and replications aimed at mitigating the impact of workload on answering the research questions ***RQ1*** and ***RQ2***.

- ***RQ1*:** By carefully examining the individual runtime QoS data of the NA, GA, and RA systems, we believe that the generated utility values reasonably reflect the QoS requirements satisfaction of each system. Each global utility value is inversely proportional to the SLA violations. In other words, the fewer SLA violations the system causes, the higher utility value the system receives.

- ***RQ2*:** The experimental results show that adaptation treatments (i.e., GA and RA) outperform or at least perform as effectively as the no-adaptation treatment. The effect of adaptation treatments is particularly significant under the heavy workload. Therefore, statically, adaptation has a positive impact on system behavior with respect to its guaranteed quality goals and SLA. The results under an extreme workload also verify this statement in the conducted experiments.

In the next section, the proposed evaluation process will be evaluated based on the four criteria of validity, reliability, flexibility and cost.

5.4 Discussion on the Evaluation Process

Our experiments were designated to validate the feasibility of the proposed evaluation process in practical use. From these experiments, we may evaluate the proposed evaluation process using the following four criteria:

- **Validity:** An evaluation process is considered valid if it measures what it intends to measure, and if the measurement is accurate [134, 135, 136]. The proposed evaluation process aims to evaluate the degree to which the SAS system meets its QoS requirements. The process makes use of the widely agreed quantified QoS requirements (SLA) and a set of widely used utility functions. The experimental results indicated that the computed utility value reasonably reflects the QoS requirements satisfaction.

- **Reliability:** An evaluation process is considered reliable if the evaluation results are consistent, have no significant errors, and are repeatable over time and across evaluators [134, 137, 138]. As a systematic approach, each step of our process is well-defined and repeatable. From the experiments, we can anticipate that the evaluation results will be highly consistent, because the SLA, workload levels, and utility functions are all predefined and remain the same throughout the experiments.

- **Flexibility:** Each step of the evaluation process is flexible. First, the SLA can be replaced by any quantified QoS requirements. Second, the load testing can be configured based on different user patterns and workload levels. Third, each of the utility functions is tunable, modifiable, and even replaceable.

- **Cost:** The SLA documents (or other quantified QoS requirements) should be available in any mission-critical systems [65]. Moreover, the load testing process consumes much less resources, compared to letting real users access the evaluated system. Finally, the utility functions are provided and are easy to tune.

In summary, we believe that the proposed evaluation process: i) is valid because its computed utility value reasonably reflect the QoS requirements satisfaction, ii) is reliable because its results are consistent, have no significant errors, and are repeatable, iii) is flexible because one can easily customize each step of the process, and iv) is low cost because the process can be easily adopted with low resource consumption.

Chapter 6

Conclusions and Future Directions

In this chapter, we summarize the findings of the thesis by presenting the contributions in Section 6.1, and outline some potential future directions that could be pursued from this research in Section 6.2.

6.1 Research Contributions

This thesis proposed an engineering process for evaluating mission-critical Self-Adaptive Software (SAS) systems. Such a process takes the quantified QoS requirements as inputs, and comprises two main phases: conducting load testing, and evaluating QoS requirements satisfaction. As a proof of concept, a service-oriented VoIP platform, *Call Controller 2 (CC2)*, was selected as the case study. We transformed CC2 into a self-adaptive VoIP platform, and conducted a set of experiments to validate the feasibility of using the proposed evaluation process in real SAS systems. The major contributions of the thesis can be summarized as follows:

- A systematic evaluation process was proposed for mission-critical SAS systems. The process treats Service Level Agreements (SLAs) as the Adaptation Requirements (ARs), which are defined as evaluation requirements for SAS systems. Moreover, it applies utility functions to generate a single value indicating the QoS satisfaction of the evaluated SAS system.
- This research deliberately discussed the uniqueness and necessity of conducting system-level load and stress testing on an SAS system for QoS evaluation.

- This work surveyed the current case studies for SAS research, and discussed what characteristics a good case study should have for SAS research. It selected a service-oriented VoIP platform as the case study, extracted its architecture, and re-engineered it into an SAS system. Compared with most SAS case studies implemented in existing research projects, this SAS system has the following merits: i) it is substantially larger and more practical, ii) it has a considerably large number of meaningful adaptation scenarios with diverse sensors and effectors, and iii) it is flexible and easy to configure, so that new adaptation mechanisms can be added with minimum effort.

- A set of empirical experiments was conducted to validate the feasibility of the proposed evaluation process in practical use. In the experiments, we adopted the proposed process for evaluating three different adaptation mechanisms on the developed self-adaptive VoIP platform.

In conclusion, we argue the validity and reliability of the evaluation process. Furthermore, due to its flexibility, other researchers and practitioners may easily adopt the process in their own work with a relatively low cost.

6.2 Future Work

The proposed evaluation process is a general approach with great extensibility. There are numerous ways to enhance this research. Several potential directions for future work are listed below:

- **To Implement a Generic Evaluation Process:** Currently, the evaluation process only focuses on mission-critical systems. One possible future research direction is developing a more general process, which can also be used to evaluate safety-critical systems. To this end, we may need to re-design or re-define the *ARs*, testing process, and utility functions for safety-critical SAS systems.

- **To Incorporate Other Types of System-Level Testing:** We argued the suitability of conducting load and stress testing for collecting runtime QoS data related to performance, reliability and availability. Yet, load and stress testing is not appropriate for evaluating the system's usability and security. In the future, we may include other types of system-level testing (e.g., usability and security testing) into the current process.

- **To Incorporate Cost Metrics:** This work only takes QoS improvement into account, but does not consider cost. However, in reality, "Benefit/Cost" (or "Cost-Effectiveness") is important in the evaluation of enterprise systems. Including cost

metrics into the process is a promising extension of this work. As mentioned in Section 2.2.4, potential cost metrics are "cost of conversion", "extra overhead", and "cost of maintenance".

- **To Implement an Evaluation Framework:** We may implement an evaluation framework for adaptation mechanisms based on the proposed evaluation process. In addition to the current VoIP platform, some more case study systems can be added to the project. Such a framework should be able to evaluate the effectiveness of most rule-based, goal-based, and utility-based adaptation mechanisms.

Appendix A

Adaptive Actions for CC2

- Blocking
  - ac_1: Completely block any service requests from bronze users.
  - ac_2: Completely block any service requests from silver users.
  - ac_3: Completely block any service requests from gold users.
  - ac_4: Counter-action for ac_1.
  - ac_5: Counter-action for ac_2.
  - ac_6: Counter-action for ac_3.
- Forwarding
  - ac_7: Not allow bronze users' calls to be forwarded.
  - ac_8: Not allow silver users' calls to be forwarded.
  - ac_9: Not allow gold users' calls to be forwarded.
  - ac_{10}: Counter-action for ac_7.
  - ac_{11}: Counter-action for ac_8.
  - ac_{12}: Counter-action for ac_9 .
- VoiceMail
  - ac_{13}: Change the max recording time of bronze users.
  - ac_{14}: Change the max recording time of silver users.
  - ac_{15}: Change the max recording time of gold users.
  - ac_{16}: Not allow bronze users to access voicemail service.

- ac_{17}: Not allow silver users to access voicemail service.
- ac_{18}: Not allow gold users to access voicemail service.
- ac_{19}: Counter-action for ac_{13}.
- ac_{20}: Counter-action for ac_{14}.
- ac_{21}: Counter-action for ac_{15}.
- ac_{22}: Counter-action for ac_{16}.
- ac_{23}: Counter-action for ac_{17}.
- ac_{24}: Counter-action for ac_{18}.

Appendix B

Goals, Weights and Preferences for CC2

- General
  - g_1: Protect from crashing $ac_1 \succ ac_2 \succ ac_3$ ($p_1 = 300$)
- Gold
  - g_2: Max Throughput of Basic calls $ac_{16} \succ ac_7 \succ ac_{13} \succ ac_{17} \succ ac_8 \succ ac_{14} \succ ac_{18} \succ ac_9 \succ ac_{15}$ ($p_2 = 18$)
  - g_3: Max Throughput of Voicemail service $ac_{13} \succ ac_7 \succ ac_{14} \succ ac_8 \succ ac_{15} \succ ac_9$ ($p_3 = 16$)
  - g_4: Min Response time of Basic calls $ac_{16} \succ ac_7 \succ ac_{13} \succ ac_{17} \succ ac_8 \succ ac_{14} \succ ac_{18} \succ ac_9 \succ ac_{15} \succ ac_3$ ($p_4 = 32$)
  - g_5: Min Response time of Forwarding service $ac_{16} \succ ac_{13} \succ ac_7 \succ ac_{17} \succ ac_{14} \succ ac_8 \succ ac_{18} \succ ac_{15} \succ ac_9$ ($p_5 = 14$)
  - g_6: Min Response time of Voicemail service $ac_{13} \succ ac_7 \succ ac_{16} \succ ac_{14} \succ ac_8 \succ ac_{17} \succ ac_{15} \succ ac_9 \succ ac_{18}$ ($p_6 = 18$)
  - g_7: Max Length of Voicemail Recording time ac_{22} ($p_7 = 10$)
  - g_8: Max Basic calls availability ac_6 ($p_8 = 54$)
  - g_9: Max Forwarding service availability ac_{12} ($p_9 = 14$)
  - g_{10}: Max Voicemail service availability ac_{25} ($p_{10} = 20$)
- Silver

- g_{11}: Max Throughput of Basic calls $ac_{16} \succ ac_7 \succ ac_{13} \succ ac_{17} \succ ac_8 \succ ac_{14}$ ($p_{11} = 14$)
- g_{12}: Max Throughput of Voicemail service $ac_{13} \succ ac_7 \succ ac_{14} \succ ac_8$ ($p_{12} = 12$)
- g_{13}: g_{10}: Min Response time of Basic calls $ac_{16} \succ ac_7 \succ ac_{13} \succ ac_{17} \succ ac_8 \succ ac_{14}$ ($p_{13} = 24$)
- g_{14}: Min Response time of Forwarding service $ac_{16} \succ ac_{13} \succ ac_7 \succ ac_{17} \succ ac_{14} \succ ac_8$ ($p_{14} = 11$)
- g_{15}: Min Response time of Voicemail service $ac_{13} \succ ac_7 \succ ac_{16} \succ ac_{14} \succ ac_8 \succ ac_{17}$ ($p_{15} = 14$)
- g_{16}: Max Length of Voicemail Recording time ac_{21} ($p_{16} = 8$)
- g_{17}: Max Basic calls availability ac_5 ($p_{17} = 41$)
- g_{18}: Max Forwarding service availability ac_{11} ($p_{18} = 11$)
- g_{19}: Max Voicemail service availability ac_{24} ($p_{19} = 15$)

- Bronze
  - g_{20}: Max Throughput of Basic calls $ac_{16} \succ ac_7 \succ ac_{13}$ ($p_{20} = 9$)
  - g_{21}: Max Throughput of Voicemail service $ac_{13} \succ ac_7$ ($p_{21} = 8$)
  - g_{22}: Min Response time of Basic calls $ac_{16} \succ ac_7 \succ ac_{13}$ ($p_{22} = 16$)
  - g_{23}: Min Response time of Forwarding service $ac_{16} \succ ac_{13} \succ ac_7$ ($p_{23} = 7$)
  - g_{24}: Min Response time of Voicemail service $ac_{13} \succ ac_7 \succ ac_{16}$ ($p_{24} = 9$)
  - g_{25}: Max Length of Voicemail Recording time ac_{20} ($p_{25} = 5$)
  - g_{26}: Max Basic calls availability ac_4 ($p_{26} = 27$)
  - g_{27}: Max Forwarding service availability ac_{10} ($p_{27} = 7$)
  - g_{28}: Max Voicemail service availability ac_{23} ($p_{28} = 10$)

References

[1] M. Salehie, S. Li, R. Asadollahi, and L. Tahvildari. Change support in adaptive software: A case study for fine-grained adaptation. In *Proceedings of the IEEE Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, pages 35–44, 2009. 1, 21, 29, 37

[2] A. Vincenzi, J. Maldonado, M. Delamaro, E. Spoto, and W. Wong. Component-based software: an overview of testing. In *Proceedings of Component-Based Software Quality-Methods and Techniques*, pages 99–127. Springer, 2003. 1

[3] D. Pescovitz. Helping computers help themselves. *IEEE Spectrum*, 39(9):49–53, 2002. 1

[4] V. Markl, G.M. Lohman, and V. Raman. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal*, 42(1):98–106, 2003. 1

[5] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, pages 41–50, 2003. 1, 6, 10, 54

[6] Z.M. Jiang, A.E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 125–134, 2009. 2, 18, 27

[7] K. Fowler. Mission-critical and safety-critical development. *IEEE Instrumentation & Measurement Magazine*, 7(4):52–59, 2004. 2

[8] P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the International Conference on Software Engineering*, pages 177–186, 1998. 2, 37, 47

[9] J. Zhang and B.H.C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the International Conference on Software Engineering*, pages 371–380, 2006. 2

[10] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. Engineering self-adaptive systems through feedback loops. *Lecture Notes In Computer Science*, 5525:48–70, 2009. 2

[11] D. Garlan, S.W. Cheng, A.C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004. 2

[12] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):1–42, 2009. 2, 3, 6, 7, 8, 9, 10, 11, 12

[13] J.A. McCann and M.C. Huebscher. Evaluation issues in autonomic computing. *Lecture Notes in Computer Science*, 3252:597–608, 2004. 2, 20

[14] B.H.C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G.M. Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H.M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. Software engineering for self-adaptive systems: A research roadmap. *Lecture Notes In Computer Science*, 5525:1–26, 2009. 2, 22

[15] J.O. Kephart. Research challenges of autonomic computing. In *Proceedings of the International Conference on Software Engineering*, pages 15–22, 2005. 3, 9, 10, 37

[16] S. Hariri, B. Khargharia, H. Chen, J. Yang, Y. Zhang, M. Parashar, and H. Liu. The autonomic computing paradigm. *Cluster Computing*, 9(1):5–17, 2006. 3

[17] R Laddaga. Self-adaptive software. Technical Report 98-12, DARPA BAA, 1997. 6

[18] M. Jackson. *Software requirements & specifications: a lexicon of practice, principles and prejudices*. Addison-Wesley, 1995. 7

[19] N. Subramanian and L. Chung. Software architecture adaptability: an NFR approach. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 52–61, 2001. 8

[20] H. Müller. Bits of history, challenges for the future and autonomic computing technology. In *Proceedings of the Working Conference on Reverse Engineering*, pages 9–18, 2006. 8

[21] IBM. Autonomic computing toolkit: developers guide, 2005. Technical Report SC30-4083-03. 8, 9

[22] Common information model standard., last visited: October 2010. http://www.dmtf.org/standards/cim/. 9

[23] Simple network management protocol, last visited: October 2010. http://www.ietf.org/html.charters/OLD/snmpcharter.html. 9

[24] Web-based enterprise management standard, last visited: October 2010. http://www.dmtf.org/standards/wbem/. 9

[25] Application response measurement, last visited: October 2010. http://www.opengroup.org/tech/management/arm/. 9

[26] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transaction*, 19(3):332–383, 2003. 9

[27] Sun JVM tool interface, last visited: October 2010. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/. 9, 10

[28] Sun Microsystems. Java management extensions instrumentation and agent specification, 2002, last visited: October 2010. http://java.sun.com/jmx. 9, 10, 41

[29] Eclipse BtM (Build to Manage), last visited: October 2010. http://www.ibm.com/developerworks/eclipse/btm. 9, 10

[30] JRat (Java Runtime Analysis Toolkit), last visited: October 2010. http://jrat.sourceforge.net/. 9, 10

[31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995. 9, 10

[32] D. Alur, D. Malks, and J. Crupi. *Core J2EE Patterns: best practices and design strategies*. Prentice Hall, 2001. 9, 10

[33] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A system of patterns: pattern-oriented software architecture*. Wiley, 1996. 9, 10

[34] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 141–147, 2002. 9

[35] D.C. Schmidt and C. Cleeland. Applying patterns to develop extensible ORB middleware. *IEEE Communications Magazine*, 37:54–63, 1999. 9

[36] R. Pawlak, L. Duchien, G. Florin, and L. Seinturier. JAC: A flexible solution for aspect-oriented programming in Java. In *Proceedings of MetaLevel Architectures and Separation of Crosscutting Concerns*, pages 1–24, 2001. 9

[37] S.M. Sadjadi, P.K. McKinley, B.H.C. Cheng, and R.E.K. Stirewalt. TRAP/J: transparent generation of adaptable Java programs. *Lecture Notes in Computer Science*, 3291:1243–1261, 2004. 9, 11

[38] A. Mukhija and M. Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *Proceedings of the International Conference on Architecture of Computing Systems*, pages 124–138, 2005. 9, 11

[39] H. Chan and T.C. Chieu. An approach to monitor application states for self-managing (autonomic) systems. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 312–313, 2003. 10

[40] P. Greenwood and L. Blair. A framework for policy driven auto-adaptive systems using dynamic framed aspects. *Lecture Notes In Computer Science*, 4242:30–65, 2006. 10

[41] C. Landauer and K.L. Bellman. New architectures for constructed complex systems. *Applied Mathematics and Computation*, 120(1-3):149–163, 2001. 10

[42] A. Dantas and P. Borba. Adaptability aspects: An architectural pattern for structuring adaptive applications with aspects. In *Proceedings of the Latin American Conference on Pattern Languages of Programming*, pages 1–15, 2003. 11

[43] Y. Liu. Enabling adaptation of J2EE applications using components, web services and aspects. In *Proceedings of the Workshop on Adaptive and Reflective Middleware*, pages 9–10, 2006. 11

[44] S. Fleissner and E.L.A. Baniassad. Epi-aspects: aspect-oriented conscientious software. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 659–674, 2007. 11

[45] B. Verheecke, M.A. Cibran, and V. Jonckers. Aspect-oriented programming for dynamic web service monitoring and selection. *Lecture Notes In Computer Science*, 3250:15–29, 2004. 11

[46] M. Engel and B. Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 51–62, 2005. 11

[47] IBM. An architectural blueprint for autonomic computing, 2006. *White paper*, last visited: October 2010. `http://www-01.ibm.com/software/tivoli/autonomic/`. 11

[48] J. Adamczyk, R. Chojnacki, M. Jarzqb, and K. Zielinski. Rule engine based lightweight framework for adaptive and autonomic computing. *Lecture Notes in Computer Science*, 5101:355–364, 2008. 12

[49] A. Friday, N. Davies, and K. Cheverst. Utilising the event calculus for policy driven adaptation on mobile systems. In *Proceedings of the IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 13–24, 2002. 12

[50] J. Keeney and V. Cahill. Chisel: A policy-driven, context-aware, dynamic adaptation framework. In *Proceedings of the IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 3–20, 2003. 12

[51] H. Lutfiyya, G. Molenkamp, M. Katchabaw, and M. Bauer. Issues in managing soft QoS requirements in distributed systems using a policy-based framework. In *Proceedings of the IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 185–201. Springer. 12

[52] L. Lymberopoulos, E. Lupu, and M. Sloman. An adaptive policy-based framework for network services management. *Journal of Network and Systems Management*, 11(3):277–303, 2003. 12

[53] W.E. Walsh, G. Tesauro, J.O. Kephart, and R. Das. Utility functions in autonomic systems. In *Proceedings of the International Conference on Autonomic Computing*, pages 70–77, 2004. 12, 15, 23, 31

[54] R.P. Doyle, J.S. Chase, O.M. Asad, W. Jin, and A.M. Vahdat. Model-based resource provisioning in a web service utility. In *Proceedings of the Conference on Internet Technologies and Systems*, pages 5–18, 2003. 12

[55] S. Rampal, D. Reeves, I. Viniotis, and D. Argrawal. Dynamic resource allocation based on measured QoS. In *Proceedings of the International Conference On Computer Communications and Networks*, pages 24–27, 1996. 12

[56] J. Rolia, X. Zhu, M. Arlitt, and A. Andrzejak. Statistical service assurances for applications in utility grid environments. *Performance Evaluation*, 58(2-3):319–339, 2004. 12

[57] M. Salehie and L. Tahvildari. A weighted voting mechanism for action selection problem in self-adaptive software. In *Proceedings of the International Conference on Self-Adaptive and Self-Organizing Systems*, pages 328–331, 2007. 12, 21, 53

[58] M.N. Bennani and D.A. Menasce. Resource allocation for autonomic data centers using analytic performance models. In *Proceedings of the International Conference on Autonomic Computing*, pages 229–240, 2005. 12, 21

[59] D.A. Menasce, D. Barbara, and R. Dodge. Preserving QoS of e-commerce sites through self-tuning: A performance model approach. In *Proceedings of the ACM Conference on Electronic Commerce*, pages 224–234, 2001. 12, 31

[60] D.A. Menasce and V. Dubey. Utility-based QoS brokering in service oriented architectures. In *Proceedings of the International Conference on Web Services*, 2007. 12, 15, 23, 31, 32, 33, 34

[61] J. Radatz, A. Geraci, and F. Katki. IEEE standard glossary of software engineering terminology. *The Institute of Electrical and Electronics Engineers (IEEE)*, 1990. 12, 14

[62] J. Tian. *Software quality engineering: testing, quality assurance, and quantifiable improvement*. Wiley, 2005. 13

[63] ISO 9126 Software quality characteristics, last visited: October 2010. `http://www.sqa.net/iso9126.html`. 13

[64] S.H. Kan. *Metrics and models in software quality engineering*. Addison-Wesley, 2002. 13

[65] S. Haines. *Pro Java EE 5 performance management and optimization*. Apress, 2006. 13, 14, 16, 17, 18, 19, 26, 28, 69

[66] J. Gozdecki, A. Jajszczyk, and R. Stankiewicz. Quality of service terminology in IP networks. *IEEE Communications Magazine*, 41(3):153–159, 2003. 13

[67] CISCO. Internetworking technology handbook, quality of service (QoS), 2010, last visited: October 2010. `http://www.cisco.com/en/US/docs/internetworking/technology/handbook/QoS.html`. 13

[68] D. Hutchison, G. Coulson, A. Campbell, and G.S. Blair. Quality of service management in distributed systems. *Network and Distributed Systems Management*, 1:273–303, 1994. 13

[69] J. Sauro and E. Kindlund. A method to standardize usability metrics into a single score. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 401–409, 2005. 14

[70] A. Seffah, M. Donyaee, R.B. Kline, and H.K. Padda. Usability measurement and metrics: A consolidated model. *Software Quality Journal*, 14(2):159–178, 2006. 14

[71] R. Jain. *The art of computer systems performance analysis*. Wiley, 1991. 14, 20, 21, 52

[72] M.R. Lyu. *Handbook of software reliability engineering*. McGraw-Hill Inc., 1996. 14

[73] J.B.D. Joshi, W.G. Aref, A. Ghafoor, and E.H. Spafford. Security models for web-based applications. *Communications of the ACM*, 44(2):38–44, 2001. 14

[74] D.A. Menasce. Load testing, benchmarking, and application performance management for the web. In *Proceedings of the Conference on Computer Measurement Group*, volume 1, pages 271–282, 2003. 15, 18, 19

[75] Z. Liu, M.S. Squillante, and J.L. Wolf. On maximizing service-level-agreement profits. In *Proceedings of the ACM Conference on Electronic Commerce*, pages 213–223, 2001. 16

[76] A. Keller and H. Ludwig. Defining and monitoring service level agreements for dynamic e-business. In *Proceedings of the Conference on System Administration*, pages 189–204, 2002. 16

[77] D.D. Lamanna, J. Skene, and W. Emmerich. Slang: A language for defining service level agreements. In *Proceedings of the IEEE Workshop on Future Trends in Distributed Computing Systems*, pages 100–106, 2003. 16, 26

[78] E. Marilly, O. Martinot, S. Betge-Brezetz, G. Delegue, A. CIT, and F. Marcoussis. Requirements for service level agreement management. In *Proceedings of the IEEE Workshop on IP Operations and Management*, pages 57–62, 2002. 16

[79] L. Jin, V. Machiraju, and A. Sahai. Analysis on service level agreement of web services. Research Report HPL-2002-180, last visited: October 2010. `http://www.hpl.hp.com/techreports/2002/HPL-2002-180.pdf`. 16, 17

[80] A. Gambi, M. Pezzé, and M. Young. SLA protection models for virtualized data centers. In *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 10–19, 2009. 16

[81] A.D.H. Farrell, M.J. Sergot, M. Salle, and C. Bartolini. Using the event calculus for the performance monitoring of service-level agreements for utility computing. In *Proceedings of the IEEE International Workshop on Electronic Contracting*, pages 6–13, 2004. 16, 23

[82] D.A. Menasce, L.W. Dowdy, and V.A.F. Almeida. *Performance by Design: Computer Capacity Planning by Example*. Prentice Hall, 2004. 16

[83] D.A. Menasce and V. Almeida. *Capacity Planning for Web Services: metrics, models, and methods*. Prentice Hall, 2001. 17, 19, 20, 25

[84] J. Kosinski, D. Radziszowski, K. Zielinski, S. Zielinski, G. Przybylski, and P. Niedziela. Definition and evaluation of penalty functions in sla management framework. In *Proceedings of the International Conference on Networking and Services*, pages 176–181, 2008. 17

[85] Z.M. Jiang, A.E. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 307–316, 2008. 18

[86] M. Bayan and J.W. Cangussu. Automatic feedback, control-based, stress and load testing. In *Proceedings of the ACM Symposium on Applied Computing*, pages 661–666, 2008. 18

[87] D.A. Menasce. Load testing of web sites. *IEEE Internet Computing*, 6(4):70–74, 2002. 19

[88] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. *ACM SIGMETRICS Performance Evaluation Review*, 26(1):151–160, 1998. 20

[89] JMeter, last visited: October 2010. `http://jakarta.apache.org/jmeter/`. 20

[90] Grinder, last visited: October 2010. `http://grinder.sourceforge.net/`. 20

[91] SIPp, last visited: October 2010. `http://sipp.sourceforge.net/`. 20, 61

[92] M. Litoiu. A performance analysis method for autonomic computing systems. *ACM Transactions on Autonomous and Adaptive Systems*, 2(1):3–29, 2007. 21

[93] M. Amoui, M. Salehie, S. Mirarab, and L. Tahvildari. Adaptive action selection in autonomic software using reinforcement learning. In *Proceedings of the International Conference on Autonomic and Autonomous Systems*, pages 175–181, 2008. 21

[94] M. Salehie, S. Li, and L. Tahvildari. Employing aspect composition in adaptive software systems: a case study. In *Proceedings of the Workshop on Linking Aspect Technology and Evolution*, pages 17–21, 2009. 21, 37

[95] S.W. Cheng, D. Garlan, and B. Schmerl. Evaluating the Effectiveness of the Rainbow Self-Adaptive System. In *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 132–141, 2009. 21, 23, 36, 38

[96] L. Zhang and D. Ardagna. SLA based profit optimization in autonomic computing systems. In *Proceedings of the International Conference on Service Oriented Computing*, pages 173–182, 2004. 22

[97] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing SLA violations. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*, pages 119–128, 2007. 22

[98] D.A. Menasce, M.N. Bennani, and H. Ruan. On the use of online analytic performance models in self-managing and self-organizing computer systems. *Self-* Properties in Complex Information Systems*, 3460:128–142. 23

[99] M.N. Bennani. *Autonomic computing through analytic performance models*. George Mason University, 2006. 23, 31

[100] S.J. Russell and P. Norvig. *Artificial Intelligence: a modern approach*. Prentice Hall, 2nd edition, 2003. 23

[101] A. Mas-Colell, M.D. Whinston, and J.R. Green. *Microeconomic theory*. Oxford University Press, 1995. 23

[102] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano-SLA based management of a computing utility. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*, volume 5, pages 855–868, 2001. 23

[103] M.J. Buco, R.N. Chang, L.Z. Luan, C. Ward, J.L. Wolf, and P.S. Yu. Utility computing SLA management based upon business objectives. *IBM Systems Journal*, 43(1):159–178, 2004. 23

[104] C.S. Yeo and R. Buyya. Service level agreement based allocation of cluster resources: handling penalty to enhance utility. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 1–10, 2005. 23

[105] J.O. Kephart and R. Das. Achieving self-management via utility functions. *IEEE Internet Computing*, 11(1):40–48, 2007. 23

[106] D. Kim and S. Park. Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 76–85, 2009. 23

[107] J. Skene, D.D. Lamanna, and W. Emmerich. Precise service level agreements. In *Proceedings of the International Conference on Software Engineering*, pages 179–188, 2004. 26

[108] H. Ludwig, A. Keller, A. Dan, R. King, and R. Franck. A service level agreement language for dynamic electronic services. *Electronic Commerce Research*, 3(1):43–59, 2003. 26

[109] Commercetest. Introduction to non-functional testing material, last visited: October 2010. `http://www.commercetest.com/what/introtoNFT.htm`. 27

[110] TestHouse. Non functional testing, 2010, last visited: October 2010. `http://www.testhouse.net/non-functional-testing/non-functional-testing.html`. 27

[111] Software Testing Stuff. Functional testing vs non-functional testing, last visited: October 2010. `http://www.softwaretestingstuff.com/2007/10/functional-testing-vs-non-functional.html`. 27

[112] D.A. Menasce. TPC-W: A benchmark for e-commerce. *IEEE Internet Computing*, 6(3):83–87, 2002. 37

[113] R. Asadollahi, M. Salehie, and L. Tahvildari. Starmx: A framework for developing self-managing java-based systems. In *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 58–67, 2009. 37, 54

[114] A. Brown and C. Redlin. Measuring the effectiveness of self-healing autonomic systems. In *Proceedings of the International Conference on Autonomic Computing*, pages 328–329, 2005. 38

[115] Y. Li, K. Sun, J. Qiu, and Y. Chen. Self-reconfiguration of service-based systems: A case study for service level agreements and resource optimization. In *Proceedings of the IEEE International Conference on Web Services*, pages 266–273, 2005. 38

[116] D. Lorenzoli, D. Tosi, S. Venticinque, and R.A. Micillo. Designing multi-layers self-adaptive complex applications. In *Proceedings of the International Workshop on Software Quality Assurance*, pages 70–77, 2007. 38

[117] D. Lorenzoli, S. Mussino, M. Pezze, D. Schilling, A. Sichel, and D. Tosi. A SOA based Self-Adaptive Personal Mobility Manager. In *Proceedings of the IEEE International Conference on Services Computing*, pages 479–486, 2006. 38

[118] B. Goode. Voice over internet protocol (VoIP). *Proceedings of the IEEE*, 90(9):1495–1517, 2002. 39

[119] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: session initiation protocol, last visited: October 2010. http://www.ietf.org/rfc/rfc3261.txt/. 39

[120] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RFC1889: RTP: A transport protocol for real-time applications. *Internet RFCs*, 1996. 39

[121] I. Ivanov. Mobicents: JSLEE for the people, by the people, last visited: October 2010. http://today.java.net/pub/a/today/2006/03/09/mobicents-jslee.html. 39, 40

[122] D. Silas. Mobicents user guide, last visited: October 2010. http://hudson.jboss.org/hudson/job/MobicentsDocumentation/lastSuccessfulBuild/artifact/html/index.html. 39, 40, 42

[123] JBoss, last visited: October 2010. http://www.jboss.org. 39

[124] Red Hat. JBoss communications platform, last visited: October 2010. http://www.redhat.com/solutions/telco. 39

[125] Mobicents, last visited: October 2010. http://www.Mobicents.org. 39, 40, 48

[126] SWiK. Jain-JSLEE, last visited: October 2010. http://swik.net/jain-slee. 40

[127] S. Lim and D. Ferry. JSLEE 1.0 specification, 2008. http://jcp.org/aboutJava/communityprocess/final/jsr022/index.html. 40, 48

[128] Call blocking, forwarding and voice mail service, last visited: October 2010. http://groups.google.com/group/mobicents-public/web/jain-slee-example-call-controller-2. 40, 43, 44

[129] T. Conlon. Bridge collapse: Why did cell phones fail?, last visited: October 2010. http://www.switched.com/2007/08/03/bridge-collapse-why-did-cell-phones-fail/. 47

[130] H. Nurmi. *Comparing voting systems*. D Reidel Pub Co, 1987. 54

[131] J.P. Bigus, D.A. Schlosnagle, J.R. Pilgrim, and Y. Diao. ABLE: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3):350–371, 2002. 54

[132] L. Atzori and M.L. Lobina. Playout buffering in IP telephony: a survey discussing problems and approaches. *IEEE Communications Surveys & Tutorials*, 8(3):36–46, 2006. 61

[133] A. Bacioccola, C. Cicconetti, and G. Stea. User-level performance evaluation of voip using ns-2. In *Proceedings of the International Conference on Performance Evaluation Methodologies and Tools*, pages 20–29, 2007. 61

[134] E.G. Carmines and R.A. Zeller. *Reliability and validity assessment*. Sage Publications, Inc, 1979. 68

[135] E. Innes and L. Straker. Validity of work-related assessments. *Work: A Journal of Prevention, Assessment and Rehabilitation*, 13(2):125–152, 1999. 68

[136] K. Schultz-Johnson. Functional capacity evaluation following flexor tendon injury. *Hand Surgery*, 7(1):109–138, 2002. 68

[137] E. Innes and L. Straker. Reliability of work-related assessments. *Work: A Journal of Prevention, Assessment and Rehabilitation*, 13(2):107–124, 1999. 68

[138] D.L. Streiner and G.R. Norman. *Health measurement scales*. Oxford University Press, 1989. 68