Automatic Performance Optimisation of Component-Based Enterprise Systems via Redundancy

by

Ada Diaconescu

A Thesis submitted to
Dublin City University
for the degree of Ph. D.
in the
Faculty of Engineering and Computing

April 2006

School of Electronic Engineering
Jim Dowling, Mr. (Head of School)
Under the supervision of
John Murphy, Ph. D.

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Ph.D. is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: _____

(Candidate) ID No.: 50161113

Date: the 24th April 2006

CONTENTS

LIST OF FIGURES

LISTINGS

List of Acronyms

AOP: Aspect Oriented Programming
API: Application Programming Interface
BMP: Bean Managed Persistence
CBSD: Component Based Software Development
CCM: CORBA Component Model
CLR: Common Language Runtime
CMP: Container Managed Persistence
COP: Component Oriented Programming
COTS: Commercial Off The Shelf
DRE: Distributed Real Time Embedded
EJB: Enterprise JavaBeans
EPP: Encapsulation Performance Problem
J2EE: Java 2 Platform Enterprise Editions
JAAS: Java Authentication and Authorization Service
JCA: Java Connection Architecture
JDBC: Java Database Connectivity
JDO: Java Data Objects
JMS: Java Message Service
JMX: Java Management Extensions
JNDI: Java Naming and Directory Interface
JSP: Java Server Pages
JTA/JTS: Java Transaction API and the Java Transaction Service
JVM: Java Virtual Machine
MTS: Microsoft Transaction Service
NVP: N-Version Programming
OMG: Object Management Group
OOP: Object Oriented Programming
RAIC: Redundant Arrays of Interchangeable Components
RB: Recovery Blocks
RG: Redundancy Group
RSL: Resource Specification Language
SAN: Storage Area Networks
SDK: Software Development Kit
SOA: Service Oriented Architecture
SQL: Structured Query Language

ABSTRACT

Component technologies, such as J2EE and .NET have been extensively adopted for building complex enterprise applications. These technologies help address complex functionality and flexibility problems and reduce development and maintenance costs. Nonetheless, current component technologies provide little support for predicting and controlling the emerging performance of software systems that are assembled from distinct components.

Static component testing and tuning procedures provide insufficient performance guarantees for components deployed and run in diverse assemblies, under unpredictable workloads and on different platforms. Often, there is no single component implementation or deployment configuration that can yield optimal performance in all possible conditions under which a component may run. Manually optimising and adapting complex applications to changes in their running environment is a costly and error-prone management task.

The thesis presents a solution for automatically optimising the performance of component-based enterprise systems. The proposed approach is based on the alternate usage of multiple component variants with equivalent functional characteristics, each one optimized for a different execution environment. A management framework automatically administers the available redundant variants and adapts the system to external changes. The framework uses runtime monitoring data to detect performance anomalies and significant variations in the application's execution environment. It automatically adapts the application so as to use the optimal component configuration under the current running conditions. An automatic clustering mechanism analyses monitoring data and infers information on the components' performance characteristics. System administrators use decision policies to state high-level performance goals and configure system management processes.

A framework prototype has been implemented and tested for automatically managing a J2EE application. Obtained results prove the framework's capability to successfully manage a software system without human intervention. The management overhead induced during normal system execution and through management operations indicate the framework's feasibility.

ACKNOWLEDGEMENTS

To my supervisor Dr. John Murphy for his continuous, professional support and guidance, for ensuring an open, flexible and productive research environment. Also for persuading me there is no reason for worry as "everything will be fine".

To Peter Hughes, James Noble, Michael Stal, Peter Tuma, Liam Murphy, Jean-Bernard Stefani and everyone questioning my research objectively and competently, for helping improve, shape and more clearly define my research and subsequently making my PhD Viva experience tolerable.

To Adrian Mos and Mircea Trofin for their creative ideas, suggestions and feedback; for the inspiring discussions on our research field and adjacent domains, even at times with contradictory perspectives and points of view. To Adrian Mos for his technical and moral support throughout the PhD; for always finding the right motivation and reassuring I would have no difficulty in successfully finalising my thesis.

To all my colleagues in the Performance Engineering Laboratory, IONA Technologies and INRIA Rhone-Alpes for sharing their thoughts and research opinions and for creating an enjoyable and stimulating working environment; In particular, to Trevor Parsons and Dave McGuiness for improving my understanding of spoken English in a cheering manner; To Doru Todinca for being a fun colleague and flat-mate; To Lucian-Mircea Patcas for the tranquil discussions over the final PhD year.

Special thanks to Jeremy Philippe for his selfless and unconditional help and support through some of the most critical phases of the PhD.

To my family for always being there for me and believing in my work and my capacity to go through with the thesis.

LIST OF PUBLICATIONS

[1] A. Diaconescu, J. Murphy, "Automating the Performance Management of Component-Based Enterprise Systems through the use of Redundancy", in Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), Long Beach, California, USA, November 7-11, 2005

[2] A. Diaconescu, "A Framework for Automatic Performance Management of Component-Based Software Systems", Doctoral Symposium, 19th European Conference on Object-Oriented Programming (ECOOP 2005), Glasgow, Scotland, UK, 25-29 July 2005

[3] A. Diaconescu, J. Murphy, "A Framework for Automatic Performance Monitoring, Analysis and Optimization of Component Based Software Systems", Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS), held in conjunction with the International Conference on Software Engineering (ICSE 2004), Edinburgh, Scotland, UK, May 23-28, 2004

[4] A. Diaconescu, A. Mos, J. Murphy, "Automatic Performance Management in Component Based Software Systems", in Proceedings of the first International Conference on Autonomic Computing (ICAC2004), full paper, New York, USA, May 16-18, 2004

[5] A. Diaconescu, "A Framework for Using Component Redundancy for Self-Adapting and Self-Optimising Component-Based Enterprise Systems", Doctoral Symposium, Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'03), Anaheim, USA, Oct 2003

[6] A. Diaconescu, "A Framework for Using Component Redundancy for Self-Adapting and Self-Optimising Component-Based Enterprise Systems", ACM Student Research Competition - 3rd place, Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'03), Oct 2003

[7] A. Diaconescu, John Murphy, "Using Redundancy for Adaptive, self-Optimising and self-Healing Component-Based Systems", International Conference on Software Engineering (ICSE2003), Workshop on Software Architecture for Dependable Systems (WADS), Portland, OR, May 3, 2003

[8] A. Diaconescu, John Murphy, "Component Redundancy for Adaptive Software Applications", Poster in the 16th European Conference on Object-Oriented Programming (ECOOP2002), University of Málaga, Spain, June 10-14, 2002

CHAPTER

ONE

Introduction

Goals of this chapter:

- The performance of large-scale, distributed systems is complex to understand and manage
- The execution contexts and internal behaviour of Internet-enabled enterprise systems repeatedly change over the system lifetime, rendering initial performance optimisations obsolete and requiring repeated optimisation processes
- Often there is no single design, implementation or configuration solution that provides optimal performance under all possible execution contexts
- There is an astringent need for automating the performance management process of enterprise systems
- Thesis contributions:
  - A component-redundancy based solution for optimising the performance of complex software systems
  - An automatic management framework for implementing the proposed solution and dynamically optimising system performance
  - A fully-automated framework prototype for the J2EE component technology
  - Documented case studies showing how the optimal component design and configuration solutions vary with the component's execution contexts

1.1 Background and Motivation

1.1.1 Enterprise Software System Complexity

Businesses are increasingly relying on software systems to manage their data, control their processes and provide access to their services. At the same time, the growing complexity of computer applications renders software development and management processes ever more difficult and costly to implement. For the same reasons, it becomes increasingly complicated to predict and reliably argue about the enterprise systems' runtime architectures, associated functional behaviour and quality-related characteristics. Global system complexity stems from the complexity of the implemented business logic and the complexity of the supporting middleware technologies and underlying platforms.

As computer applications are gradually being employed to support everyday life processes they are required to provide ever more complex functionalities and quality guarantees. Middleware technologies are commonly being adopted to provide reusable services across enterprise systems, including security, persistence and distributed connectivity. This approach clearly separates the business logic specific to each application from the common middleware services required across multiple systems. Developers can concentrate on building the application's business logic, without having to consider the required non-functional system services. In addition, component technologies have been introduced to further increase the modularity and reusability of the application business logic and its separation from common system services. During system execution, the required middleware services are automatically intertwined with the components' business logic, so as to provide the complete required system behaviour. The modularity and consequent flexibility and reusability provided by this solution increases productivity and lowers development and maintenance costs. Nonetheless, the emergent system runtime behaviour commonly becomes considerably complex and difficult to comprehend. The runtime composition of the software system can significantly vary from its design-time architecture, in terms of both the number of instantiated components and their interconnections. In certain component technologies the actual number of runtime component instances is generally controlled by the middleware lifecycle services rather than by the components' code. In addition, the dynamic characteristics of component-based enterprise systems mean that components can be repeatedly updated or that they can interconnect during runtime in unpredictable ways. Additionally, available system resources and incoming workloads can significantly vary during a system's lifetime, potentially having considerable impacts on its quality characteristics. This consequently multiplies the efforts required to understand, predict, and control the systems' functional and quality-related behaviours.

1.1.2 Enterprise System Performance - Importance and Challenges

The progressive assimilation of computing systems into everyday life places important requirements on the quality characteristics of such systems, including correctness, availability, reliability, performance, or security. For most enterprise systems, meeting performance requirements can be as important as providing the correct functionality of their advertised ser-

vices. According to a recent study at IBM[1]:

> "A quick look at almost any current IT survey reveals that optimization ranks high. More than any other issue, except perhaps for the ubiquitous and unrelenting concern for security, optimization leads the list of issues that the IT community cares about.
> From a business perspective, the appeal of optimization is obvious and consistent with the goals of any organization – delivering customer value in a timely and cost efficient manner. It's a value proposition that's not lost on the decision makers across the IT organization. In the day-to-day world of IT, optimization translates into more efficient use of resources, as well as lowered management costs."

Component technologies [91], such as J2EE[2] and .NET[3] are increasingly being used for building complex enterprise system. In component-based applications, the individual behaviour of every component and the collective behaviour of interacting components, in the specific execution environment, determine the overall application performance. Nonetheless, system complexity and lack of information on components and their execution platform make performance of enterprise applications hard to analyse and predict. In these conditions, understanding, predicting and controlling the resulting system performance or reliability characteristics becomes a complicated task. The existing component technologies provide little or no support for facilitating such tasks.
While successfully addressing complex system functionality issues and flexibility requirements, current component technologies provide little support for managing the emerging performance of systems assembled from distinct components. Static component testing and tuning procedures are typically run in isolation or simulated environments. Although important, such procedures provide insufficient performance guarantees for components that are to be run in diverse component assemblies, under unpredictable workloads and on different platforms. The environmental conditions in which a component runs as part of a software application can periodically change during the component's lifetime. Such environmental conditions include the incoming workload and the software and hardware resources available to a component. Changes in these running conditions can significantly impact on a component's availability and performance characteristics, including the component's throughput and response times. As such, repeated variations in a component's execution context can render initial performance optimisations obsolete.
Often, no single design, implementation and configuration solution exists that provides optimal performance under all possible execution conditions in which an application component may run [101, 93, 3]. Attempting to predict all possible execution scenarios and accordingly build optimal behaviours corresponding to each scenario into a single monolithic component is an inflexible solution. The lack of modularity and separation of concerns implied by such monolithic designs would make this a costly and risky approach, if at all accomplishable. A more attainable solution would be to constantly adapt applications at runtime, so as to contin-

[1] "Self-managing systems. The optimization challenge", Rob Cutlip, 23 Aug 2005: www-128.ibm.com/developerworks/library/ac-selfo/?ca=dnt-634

[2] The Java 2 Platform, Enterprise Edition technology (J2EE), from Sun Microsystems: java.sun.com/j2ee

[3] .NET technology, from Microsoft: www.microsoft.com/net

ually optimise them to variations in their execution environments and variations in their functional and quality requirements. Nonetheless, this consequently implies that complex system management, configuration, and tuning tasks must be performed repetitively during runtime, rather than solely before system deployment. Manually optimising and adapting complex applications to changes in their running environment is a costly and error-prone task. Performing such processes during system execution in a timely and reliable manner becomes a difficult task, at best. In consequence, there is a stringent need to provide autonomic system adaptation and management solutions that decrease the burden on human system managers. Automatic management processes should assist human administrators in performing common system adaptation and verification tasks, in a dependable and timely manner. This would allow administrators to concentrate on high-level system management goals, rather than worry about how these goals should be attained via low-level system management processes.

1.2 Thesis Contributions and Scope

The main contribution of this thesis is to propose the use of component redundancy to automate the performance optimisation of Internet-enabled enterprise systems, at the application component level. Conforming to this approach, multiple component variants are acquired to provide equivalent functionalities, but via different design, implementation and configuration strategies. Each component variant is optimised for a different execution context, such as various incoming workloads, or available underlying resources. The redundancy-based optimisation solution is based on knowledgeably alternating the use of the available component variants, so as to execute the optimal component variant in each execution context. Based on this, application components are able to adapt their behaviour so as to always be optimal in their varying execution environments. Performance optimisations are considered at different granularities, from the local component level to the overall system level. In effect, the entire software system is able to adapt to variations in its running environment and yield optimal performance characteristics, at all times.

As a related contribution, several example scenarios were described, implemented and tested in order to support the proposed redundancy-based optimisation solution. The examples clearly show the impact that design and configuration choices have on runtime performance characteristics, such as response times, throughput or resource consumption. Test results prove that a component's optimal implementation strategy can decisively depend on the execution context in which the component runs. The tested scenarios indicate the way the knowledgeable alternation of several distinctive strategies, optimised for different execution contexts, can yield better performance in varying execution conditions than any single implementation strategy could.

A further contribution of this thesis is an automatic management framework for supporting the redundancy-based optimisation approach. The framework is referred to as AQuA (Automatic Quality Assurance). AQuA enables component-based systems to fluidly mould to variations in their execution environments so as to provide optimal performance at all times. Similar automatic management frameworks exist in the related research areas. AQuA generally complies with the overall architectures of the management framework specifications proposed in the literature [54, 72]. Nonetheless, it notably differs from other existing frame-

works in the way of addressing the particular challenges of the targeted system types.
AQuA's automatic management process involves monitoring the system and its execution environment, using decision logic to detect optimal system adaptation solutions and enforcing optimisation solutions into the running system. In short, AQuA collects system monitoring data, detects execution environment variations and identifies application performance anomalies. It subsequently identifies possible solutions to the detected problems and accordingly adapts the application so as to optimise it for its current running environment. This process is executed in a feedback-loop manner. More precisely, the outcome of an adaptation operation is detected by the monitoring function and compared with the predicted result, or adaptation goal. A learning mechanism was specified for enabling AQuA to use the collected monitoring data for enhancing its knowledge on the managed components and improving its management behaviour over time.
The AQuA framework was designed taking into consideration the particular characteristics of the targeted managed applications and their underlying implementation technologies. Specifically, AQuA was specifically devised to manage enterprise applications built using contextual composition framework technologies, such as Enterprise JavaBeans (EJB) [80]. The focus was on managing enterprise systems at the software application component level. Nonetheless, AQuA can be extended to manage other enterprise system component types, such as middleware services, or application servers, with minimum required modifications.
In order to meet the specific management requirements of the targeted enterprise systems, AQuA was designed so as to be flexible with respect to the management control topology used. In other words, AQuA can conceptually implement either a centralised, decentralised, or mixed (e.g. hierarchical) management control solutions. The reason is that AQuA's management capabilities do not rely on a centralised system model. The rationale behind this choice was that obtaining, analysing and maintaining an accurate model of the runtime application would have been exceedingly costly and might have not scaled well in the context of the targeted managed systems.
Related to the AQuA framework contribution, the thesis also provides a sample framework prototype for managing J2EE enterprise applications. This prototype is referred to as AQuA_J2EE and was built to work with the JBoss application server for J2EE. AQuA_J2EE was built around a modular architecture, providing increased flexibility and manageability to the prototype's implementation. As a result, AQuA_J2EE can be extended so as to work with diverse J2EE application servers, other than JBoss, without requiring major design or implementation modifications. In addition, any of the prototype's management functions can be independently extended or replaced, without affecting the other function implementations. The provided framework prototype features sample implementations for AQuA's main management functions, including runtime monitoring, data analysis for performance anomaly detection and learning, redundant component evaluation, adaptation decision and dynamic component activation.
The scope of this thesis includes a redundancy-based approach for optimising the performance of complex software systems. The thesis aims to clearly describe the proposed solution and supporting it via documented examples, implementations, test results and discussions. Nonetheless, the thesis does not attempt to address all the challenges associated with fully implementing the redundancy-based optimisation solution. As such, providing an optimised, reliable and fully-functional framework implementation, ready to be applied for managing any complex system, is out of the thesis scope.

1.3 Thesis Overview

Chapter 2 provides an introduction to component-based software (CBS) and the J2EE component technology. It also presents a general overview of the area of performance management for complex software systems. The main research directions and associated approaches are described and analysed. The existing research efforts towards automating system performance optimisation are discussed and their applicability and limitations identified.

Chapter 3 describes the proposed solution for automatic performance optimisation and adaptation in component-based enterprise systems. The focus is on component technologies based on contextual composition frameworks [91], such as the Enterprise JavaBeans (EJB) technology [80]. The presented approach is based on the use of redundant components for dynamically optimising system performance. The chapter discusses possible scenarios that would benefit from the proposed optimisation approach, addresses the associated complexity and cost concerns and indicates the solution's applicability in the context of component-based enterprise systems. The solution also proposes an automatic management framework, capable of administering redundant components, of optimising applications and of meeting system performance requirements. The framework achieves its goals by automatically and repetitively monitoring, evaluating and swapping the available redundant components during runtime. The main roles and functionalities of the framework's main logical modules are being described.

Chapter 4 describes the design and implementation details of proposed optimisation solution and framework. It explains the implementation logic and technical choices made for building each framework function, including the system monitoring, anomaly detection, learning, component evaluation, adaptation decision and component activation.

Chapter 5 is allocated to presenting the experimental work of the thesis. The goal is to validate the proposed optimisation solution, the management framework and the provided prototype implementation. The chapter describes the investigated example scenarios, testing procedures and the obtained results. Two implemented example scenarios are described in order to indicate the potential benefits of the component redundancy based solution. Results from testing the framework prototype's automatic management capabilities on one of these examples are shown and analysed. Preliminary test results from the framework's data analysis and learning capabilities are presented. The chapter discusses the significance of the obtained experimental results for supporting the thesis proposed solution, validation and claims.

Chapter 6 concludes the thesis by reviewing the contributions, summarizing the addressed performance management problems and the provided optimisation solution. General related work presented in Chapter 2 is being reviewed and relevant existing projects are being compared with the proposed optimisation solution. Finally, the chapter summarizes the validation procedures and results supporting the provided solution and indicates the current limitations and further investigation opportunities for the presented thesis research.

CHAPTER

TWO

Background and Related Work

Chapter Summary
This chapter presents background information and related work relevant to the thesis topic. It describes quality attributes, enterprise systems, component-based software, contextual composition frameworks and the J2EE component technology. Relevant related work is presented from the areas of autonomic computing, self-adaptive systems, automatic management frameworks and the use of redundancy in software applications. The chapter discusses the goals, problems addressed and remaining challenges of existing work in the targeted research domain.

Goals of this chapter:

- Besides functional requirements, enterprise systems have stringent quality constraints which are critical to the supported business's success
- Component technologies are increasingly being adopted to provide flexible, manageable and reusable solutions for complex software systems. However, existing component technologies do not address performance management issues
- EJB is one of the most highly adopted component technology for implementing enterprise applications
- Certain EJB system characteristics make the existing self-management approaches developed for different system types difficult to apply for administering enterprise applications
- The goals and problems solved by previous redundancy-based solutions do not fully address the challenges of performance optimisation in EJB-based enterprise systems

2.1 Introduction to Relevant Research Areas and Technologies

This thesis is concerned with the automatic management of complex software applications. The focus is on the performance management of Internet-based enterprise systems, built using component technologies. At present, component technologies based on contextual composition frameworks [91] are commonly selected for building enterprise applications. This thesis consequently targets the management of enterprise systems built using such technologies. The Java 2 Platform, Enterprise Edition (J2EE)[1] technology was chosen for implementation and experimentation purposes in the presented research work. The choice was based on the fact that J2EE is currently the most popular component technology adopted by the industry for building complex enterprise systems.

This chapter provides background information relevant to the targeted research domain. This includes a description of system quality attributes, relevant software engineering disciplines, such as component software and contextual composition frameworks and the J2EE Enterprise JavaBeans (EJB)[2] component technology. Based on this, the chapter subsequently discusses some of the main concerns and the most significant related research in the area. Several remaining problems concerning performance management of Internet-based enterprise systems are indicated and the reasons these problems are not currently addressed by existing research in area examined.

Various approaches have been proposed for managing software system complexity. Some of the most significant initiatives involve dynamic system evolution (e.g. [35], [79], [38], [8]), autonomic computing (e.g. [54], [3], [99] and [73]), self-managing systems[3], adaptive systems (e.g. [72], [61], [37], [16] and [66]), fault-tolerance (e.g. [62], [17], [5] and [78]), self-optimisation (e.g. [34]) and self-healing (e.g. [47], [23], [20] and [19]) approaches. These initiatives have somewhat overlapping goals. Taken together, they can be viewed as addressing different parts of the more general goal of autonomic system management. The Autonomic Computing (AC) initiative[4] provides a complete vision and specification on this overall goal.

Several approaches use redundancy as a means of achieving fault-tolerance and/or performance optimisation in software applications (e.g., [6], [78], [47], [64] and [101]). Research on the topic has been directed towards all layers that are typically involved in a complex software system, including software applications (e.g. web applications, business logic and databases), distributed platforms or middleware, operating systems and hardware platforms.

This chapter presents some of the most relevant initiatives and challenges in the aforementioned research directions. Background information relevant to the presented research precedes discussions on related work. This includes information on system performance and other quality attributes, component-based software, contextual composition frameworks and the J2EE/EJB technology. Numerous challenges remain to be addressed for achieving auto-

[1]Java 2 Platform, Enterprise Edition (J2EE) technology from Sun Microsystems: java.sun.com/j2ee

[2]J2EE / Enterprise JavaBeans (EJB) Technology from Sun Microsystems: java.sun.com/products/ejb

[3]Self-managing systems. The optimization challenge, by Rob Cutlip, IBM developerWorks, Autonomic computing, online article series: www-128.ibm.com/developerworks/library/ac-selfo/?ca=dnt-634

[4]Autonomic Computing initiative from IBM: www.research.ibm.com/autonomic

nomic performance management in the area of Internet-based enterprise systems. The particular characteristics of such complex software systems cause difficulties in applying existing approaches from related research domains.

2.2 Software Quality and Quality Attributes - general taxonomy, definitions and tradeoffs

For most software systems, merely satisfying functional requirements is not enough. Quality characteristics, such as performance, dependability or security are just as important as the functional capabilities of the software system. Failure to satisfy such non-functional requirements can render the system as unable to accomplish its designated mission, jeopardizing its success and possibly resulting in loss of revenue, data, or business.
Software system *functionality* refers to the system's capabilities or behaviour, or more informally, to the 'things it can do'. System functionality is provided to system clients as *services*, *methods*, or *functions*, accessible via system *interfaces*. An *interface* is defined as a functionality abstraction that only defines the operations supported by that functionality but not their implementation [91].
With respect to software *quality*, a general definition is provided in the IEEE Standard for a Software Quality Metrics Methodology (1061-1992) [49]:

> "Software quality is the degree to which software possesses a desired combination of attributes (e.g. reliability, interoperability)."

Such attributes represent non-functional characteristics, or properties, of delivered system services. They are commonly referred to as *quality attributes*. A further discussion on the concept of quality in large-scale, distributed software systems is available from [32].
Software quality attributes can be divided into two main categories, based on whether they are observable at runtime, or during the software application development cycle. Quality attributes observable at runtime include performance, dependability and usability. Development time attributes include maintainability, reusability and portability. Runtime quality attributes, also referred to as *external attributes* [45], are perceived by system users, whereas development-time quality attributes, also referred to as *internal attributes* [45] mostly affect system developers and maintainers. The thesis is mainly concerned with the former category of quality attributes, namely, external attributes observable during runtime. The focus is on performance-related attributes, such as response times, throughput and resource usage.
Performance and dependability quality attributes are further discussed in subsections 2.2.1 and 2.2.2 next. Some of the quality attribute terminology defined for the scope of this thesis may have slightly different meanings depending on the targeted system's type (e.g. latency and throughput in telephony, data processing, or computer networking systems). Subsection 2.2.3 indicates the dependencies that exist between various quality attributes and the way they influence global system optimizations.

2.2.1 Performance

Performance is a broad concept, with many connotations, depending on the context in which it is used. The IEEE Standard Glossary of Software Engineering Terminology (610.12) [48] provides a very general definition of performance:

> "The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage."

In [11], performance is regarded as a system quality attribute that characterises the timeliness of system provided services. A more accurate definition of performance is provided in [12], or [11]:

> "Performance refers to responsiveness of the system - the time required to respond to stimuli (events) or the number of events processed in some interval of time".

In some cases, performance is also equated with efficiency [45]. Efficiency generally refers to the amount of resources, such as time, energy and effort, consumed by an approach, or solution, for obtaining certain result or effect [91]. A general taxonomy of system quality attributes is specified in the 'Quality Attributes' initiative from the Software Engineering Institute [11].

Performance Metrics

Metrics represent parameters by which a system quality attribute is specified, measured and evaluated [11]. Some of the most important performance metrics include:

- *Response time*: the delay for returning a response to a client request. It is a time measurement between the moment a request is received and the request is fulfilled.
- *Latency*: the time interval after which a system reacts to an incoming event or request. It is a measurement of temporal delay. In communication applications it refers to the delay in time between the sending of a unit of data at one end of a connection until the receipt of that unit at the destination. In data processing systems, latency refers to the delay between the receipt of an input event and the system starting to react to that event. It is the time it takes a system to react to a service request, or incoming event.
- *Throughput*: the number of requests that can be handled in a certain, clearly specified time interval. It is a measure of the amount of work a system can perform over a given time period. This is different from the system processing rate, since it does not mean that a constant amount of work is performed over the specified interval. For example, if a system's throughput equals 100 requests per minute, it cannot be concluded that the system is processing 50 requests per 30 seconds. It is possible for the system to be idle for 30 seconds and then handle 100 requests in the subsequent 30 seconds. In communication systems, throughput is defined by the amount of data transferred per time unit.
- *Capacity*: the maximum load a system can support while meeting response time, latency and throughput requirements. It is a measure of the maximum amount of work a sys-

tem can perform. Capacity is commonly defined in terms of throughput, meaning the maximum number of events processed per time interval, in the ideal execution conditions. The useful capacity is the maximum throughput a system can achieve while not violating other performance specifications such as response times or latency.

A criticality factor can be associated with most performance metrics to indicate the importance of that metric to the system. Another important performance concern (sometimes referred to as system *modes* [11]) describes the way a system's performance behaviour changes over time, with the variations in the system's demand and resources.

Performance-Specific Factors

Attribute-specific factors represent properties of the system and its environment that have an impact on the attribute metrics [11]. The main factors influencing performance include the incoming workload, the types and amounts of available system resources and the manner in which resources are allocated to request handling processes. System workload generally maps to various demands on the required resource types. Workload can be characterised in terms of the incoming request frequency (or load), request types (or event types), arrival patterns and rates for the existing request types (or work-mix) and resource usage requirements for handling each request. Each request can be mapped to the underlying software and hardware resources required to handle the request. Software resources include threads, processes, or connections at the JVM and OS levels. Common hardware resource types include CPU, memory, device I/O, storage, or network resources. Software services, typically implemented at the OS level, manage the access to and allocation of each available resource type.
The arrival patterns and execution times required for handling the possible request types can be used to predict system performance, in terms of response times and throughput values. Queuing and scheduling theories are sued for this purpose to model system behaviour and predict their performance behaviours in different execution conditions. Arrival patterns can be periodic, meaning that they regularly occur at fixed intervals, or aperiodic, meaning that they occur repeatedly at irregular time intervals. Execution times can be defined using probability distributions functions. Worst-case and best-case values can be specified to help define boundary conditions behaviour.

Performance Management Methods

Management methods are means of addressing and optimising an attribute's metric values - methods encompass activities such as prediction and evaluation of targeted system metrics, as well as system architecture and implementation modification for improving these concerns. Management methods can be devised for different stages in a system's lifecycle. Development methods are applied during the system's architecture specification, design, and implementation. After system deployment, methods are employed during system execution, for runtime system management purposes.
Methodologies for ensuring software performance have been devised, targeting both system development and runtime stages. Approaches concerned with software system development, advocate the integration of performance analysis and prediction with the various phases of system development. Some of these approaches include:

- The Software Performance Engineering (SPE) initiative[5], [84], or [100]. This initiative advocates building system performance models and simulating system behaviour to predict its runtime performance. Models represent the main system entities and their interconnections. Each system entity is characterised by its predicted behaviour, performance characteristics and interactions with other entities. Simulating system model executions helps predict the system's performance characteristics. The method is to be applied from the early stages in a system's development process, and applied iteratively at increasing levels of detail until the system completion.
- Initiatives from the Software Engineering Institute (SEI)[6], (e.g. Architecture Trade-off Analysis (ATA) [53], Attribute-Based Architecture Styles (ABAS) [58]); Performance-Critical Systems (PCM), Component Based Development (CBSD), or COTS Based Systems initiatives[7]. Component-based software development (CBSD) and COTS integration focuses on building large software systems by integrating previously-existing software components. This approach improves system flexibility and maintainability, potentially reducing software development and maintenance costs. The Performance-Critical Systems (PCS) initiative focuses on technical practices for analyzing and predicting the performance and dependability of software systems. It includes model-based engineering practices and tools and methodologies for documenting and predicting system performance and dependability.
- Approaches based on design patterns (e.g. [97]) (e.g. patterns for J2EE [2], or general patterns (GoF) [41]), or anti-patterns. These approaches focus on determining and defining best design practices, providing optimal solutions to common implementation problems in certain contexts. Applying the recommended design patterns may insure certain system quality properties, such as flexibility and manageability (internal attributes), as well as performance (external attributes). Related practices can be performed during system development or runtime, in order to detect and correct application anti-patterns, or ensure pattern consistency during system evolution.
- Approaches based on prototypes and trace analysis (e.g. [46])
- Efforts towards performance modelling and analysis using UML (UML documentation from OMG: [70]) (e.g. OMG "UML Profile for Schedulability, Performance, and Time" [71]; or [75], [56], [26]).

These methods assess and predict system performance at various stages in the software development process - specification, architecture, design and implementation. Performance management methods are also required during system runtime, to detect and correct design and configuration problems affecting system performance, optimise systems to various execution environments and maintain performance-critical design patterns and architectural styles during system evolution.

Development Time Performance Management Methods

In the early stages of system creation, there is no executable system or prototype implementa-

[5]Software Performance Engineering (SPE) initiative - founded by Dr. Connie Smith: www.perfeng.com

[6]Software Engineering Institute, at Carnegie Mellon University: www.sei.cmu.edu

[7]www.sei.cmu.edu/str/descriptions/cbsd.html

tion available for performance evaluation. System artefacts are used instead for performance prediction purposes. System artefacts, or models, are system abstractions that capture the system's structural and behavioural aspects. The accuracy or level of detail of these artefacts depends on the system's development stage when these artefacts are being built (e.g. system general architecture stage, or detailed design stage). Architecture specification is considered the first stage in a system's creation when system quality requirements, such as performance, can begin to be addressed [13]. System architecture is a first system artefact, representing an abstraction of the system's structural and behavioural characteristics. A system's architectural structure generally consists of the system modules (or components) and their interconnections. Various formal notations have been devised for representing system structure and runtime behaviour [9]. Formal notations for system structure include directed graphs, architecture design languages (e.g. Fractal ADL[8]) or Unified Modelling Language (UML) class diagrams. Example notations for a system's runtime behaviour include UML interaction diagrams, Message Sequence Charts (MSC), Execution Graphs, Use Case Maps (UCM), Automata, Process Algebras and Petri Nets.

System performance models are created based on system artefacts and various relevant estimates. Estimates are made on workloads, software execution paths, resource characteristics such as processing delays, resource requirements for the executable units involved and execution environment characteristics [100], [84]. When executable prototypes or simulation models are available, these estimates can be obtained by means of monitoring. Some of the most common performance model classes are Queuing Networks (QN) (or extensions, such as Extended QN and Layered QN), Stochastic Petri Nets (SPN), and Stochastic Process Algebras (SPA). In addition, recent approaches propose creating performance models by augmenting notations already used for representing system models or designs, such as UML [70], RM-ODP, or SDL, with performance related information (e.g. performance annotations for UML [71], [75], [56], or [26]). System models augmented with performance information are methodologically or automatically transformed into one of the traditional performance model representations - QN, SPN, or SPA.

Performance models are evaluated by using analytical methods or simulation techniques, in order to predict performance indices, such as throughput, response times, or resource utilization. Simulation techniques are considered harder and more costly to build, but provide more accurate results than analytical methods [45], [9]. Therefore, a combination of the two techniques is sometimes recommended [45]. More precisely, during the incipient system creation phases a general analytical model is built in order to detect performance-critical system parts that require a more detailed examination. Consequently, simulation models are (only) built for the detected performance-critical system parts.

If performance prediction results are in line with performance objectives, developers can proceed to reify the existing system architecture, design, or implementation. Otherwise, alternatives must be considered, taking into account performance indicators such as bottlenecks, or resource contention. Iterative processes have been recommended for system development [84], [4], [18], or [100], in order to facilitate performance assessment at various development stages rather than only for the fully developed system.

[8]Fractal Architecture Description Language (ADL): fractal.objectweb.org/tutorials/adl

Runtime performance methods

System performance management does not end with the system development phase. It continues with system deployment and into the system execution phases, as a continuous or repetitive process. This is not surprising, since performance is an external quality attribute that is only observable at system runtime.

Extensive research has been carried out towards modelling, evaluating and improving software performance during runtime. Some significant research directions in the area of runtime performance management include the following:

- Runtime performance assessment, by means of monitoring: involves the use of monitors that collect data from the executing system and detect bottlenecks. Runtime monitoring mechanisms have been proposed and implemented for various system types and at various system layers (e.g. EJB software applications: COMPAS [67], [66]; EJB application servers, CORBA middleware; JVM; OS; network layer). For certain software application types, system execution often provides the first opportunity for an accurate assessment of system performance. This is especially the case for complex, large-scale software systems, built using contextual composition frameworks [91] such as Enterprise JavaBeans (EJB), or CORBA Component Model (CCM).
- Runtime performance anomaly detection, bottleneck localisation and performance prediction, based on monitoring information (e.g. [74], or [67])
- Runtime performance improvement by means of dynamic system modification. Extensive initiatives in this area focus on developing systems with dynamic, adaptive, self-optimizing, self-repairing, reflective, or evolving characteristics. It also includes research on load balancing, component migration, replication, caching - targeted at different system layers and system types. In several cases, performance-related methodologies initially devised for system development time were adopted and modified to be applied during system execution.

2.2.2 Dependability

Most researches in the area of computer system dependability, promote a general definition of this concept. Conforming to this definition, *dependability* is commonly thought of as a quality of a system such that reliance can justifiably be placed on the services this system delivers [62], [11], or [17].

Dependability Metrics

Dependability is a composed concept, encompassing several attributes, such as:

- *Availability*, or readiness for usage, which is usually expressed as the time percentage during which the system is not out of service [11].
- *Reliability*, or continuity of usage in terms of statistical behaviour, it is the probability that the system will work as expected over a certain time interval [45], [11]. Reliability encompasses sub-attributes such as:

- *Correctness* is the capability of a system to perform its job, according to its (functional requirements) specification [45]; this definition assumes that the system specification is available and that the equivalence between the software and its specification can be unambiguously established. Correctness can be assessed using experimental methods, such as testing, or analytical methods, including formal correctness verifications.
- *Robustness* is the ability of a system to handle abnormal situations, which were not anticipated in the specification, such as incorrect inputs, or hardware resource failure. Hence, correctness and robustness are strongly related concepts, being delimited by the specification of the system. If a system requirement is in the specification, its achievement becomes a problem of correctness. Otherwise, it is an issue of robustness [45].

Reliability can consequently be thought of as a measure of observable faults, or failures.

- *Security*, or the ability of the system to withstand unauthorized access, alteration, or destruction of data or processes [11]. Security concerns that can consequently be inferred include confidentiality, integrity and availability.
- *Safety*, or the non-occurrence of catastrophic consequences on the environment, such as accidents, or mishaps [11].

Even though the general definition of dependability is consistent across different research domains, dissimilar views exist on the exact attributes that constitute dependability. Performance for instance, is considered a dependability attribute by some works, for example [17], but not in others [11], or [62]. Nevertheless, certain attributes, such as reliability, availability, or safety, commonly appear as dependability-related concerns. Therefore, the adopted view in the presented thesis regards dependability and performance as separate quality attributes. In addition, the two attributes are considered as generally conflicting, as improved performance may negatively impact dependability and vice versa [53].

Dependability-Specific Factors

The main factors that affect dependability are *faults*, *errors* and *failures* [11], [62]. A *fault* represents impairment in the system, or system usage. It can consist of a design defect, an illegal input, or a resource failure. The activation of a fault, by executing faulty code for example, leads to an *error*. The occurrence of an error means that the system state deviated from the designers intent. The propagation of an error to the externally observable system behaviour leads to *failure* [62]. Failure occurs when system behaviour deviates from its specification. However, some argue that deviation of system behaviour from its specification can be a result of a specification fault. Hence, in their view, failure is defined as system behaviour that differs from intent [11]. System failure can occur both in terms of the system not meeting its functional specifications, as well not complying with its performance requirements.

Dependability Management Methods

The main means or methods typically used for achieving dependability include fault-avoidance, fault-tolerance, error-removal and error forecasting [62]. Fault-avoidance methodologies provide means of preventing the introduction of faults, at system development time.

Fault-tolerance mechanisms enable systems to provide their intended or specified services, even in the presence of system faults. Most fault-tolerance mechanisms are redundancy-based. Subection 2.7.1 presents several commonly used fault-tolerance schemas. Error-removal methodologies are intended to minimise the presence of latent errors, by means of system verification, possibly followed by fault detection and removal. Error forecasting is related to the process of system evaluation, in order to predict the presence of system errors and estimate their consequences. In short, fault-avoidance and tolerance are means of providing system dependability. Error removal and forecasting are ways of validating system dependability, thereby increasing confidence in the system's capability of delivering the specified service(s).

2.2.3 Quality Attribute Tradeoffs

Most system quality attributes are in conflict. For example, improving system performance would typically impact system dependability and vice-versa. Another example, improving system efficiency generally implies diminished portability and maintainability. In addition, internal quality attributes observable at system development time, can conflict with external quality attributes observable during runtime. The Component-Based Software Development (CBSD) approach (subsection 2.3.4) provides a relevant example in this sense. CBSD mainly benefits internal quality attributes, observable during software development and management stages. In other words, CBSD facilitates certain software application features, such as modularity and reuse. For complex, large-scale software systems, such features are essential for providing (internal) quality attributes, such as manageability and development efficiency. However, even though these features prove beneficial at system development time, they have a rather negative impact on external quality attributes, such as performance, during runtime [21], [97].
The thesis focuses on the automatic performance optimization of complex, large-scale systems, built with component technologies. Tradeoffs between the various quality attributes have to be considered when performing runtime optimizations. Namely, rather than focusing on optimizing individual quality attributes, quantified combinations of such attributes must be evaluated and system quality optimized overall.

2.3 Component Technologies for Enterprise Systems

2.3.1 Enterprise Software Systems

This thesis focuses on the performance management of Internet-enabled enterprise systems. Representative examples of this type of software systems include online banking, e-commerce, or online stock brokerage applications, as well as corporate and government intranets and information systems. This subsection summarises the main features of enterprise systems, in order to support subsequent discussions on viable management solutions.

Internet-enabled enterprise systems are generally large-scale, distributed, with complex business logic and complex middleware infrastructures. They are also highly dynamic, as they must undergo frequent modifications, to adapt to constant changes in business requirements, or execution environments. For these reasons, system properties such as flexibility, reusability and manageability are critical for an enterprise system's long-term success. These properties ensure seamless support for repeated modifications and reconfiguration operations on a system's complicated behaviour. In terms of quality characteristics, it is crucial for business success to make enterprise systems highly available, reliable and capable of functioning at competitive performance levels. However, in enterprise systems short service interruptions (of the order of seconds) or seldom degraded performance events are not entirely unacceptable.

2.3.2 Component Software

There are different views on what software components really are. Consequently, different definitions exist for software components, stating dissimilar component characteristics or criteria[9]. Ambiguity is increased as terms such as 'component', 'module' and 'object' are sometimes used interchangeably. To avoid confusion, the thesis adopts one of the most widely accepted views on software components[10] [91], [92]. This view on software components is shortly discussed in the following sections. The presentation includes component definitions and describes the main component characteristics. It also discusses component related concepts, such as component frameworks, architectures, models and platforms. Throughout the thesis, the term component will always be used to refer to a software component, defined in conformance with the presented view.

2.3.3 Component Concepts and Definitions

This section describes and defines the main concepts and terms related to component software. The thesis uses the software component concept as defined in Clemens Szyperski's book on Component Software [91]:

> A (software) component is a "unit of composition with contractually specified interfaces and explicit context dependencies only. Context dependencies are specified by stating the required interfaces and the acceptable execution platform(s). A component can be deployed independently and is subject of composition by third parties. For the purpose of independent deployment, a component needs to be an executable unit. To distinguish between the deployable unit and the instances it supports, a component is defined to have no observable state. Technically, a component is a set of atomic components, each of which is a module plus resources. A component targets a particular component platform. The

[9] 'Beyond Objects', Software development magazine (online): http://www.sdmagazine.com/beyond/ - debate on software component definitions, between Bertrand Meyer and Clemens Szyperski

[10] Software development magazine (online), Beyond Objects track, by Clemens Szyperski

composition of components follows one or more composition schemes that are mandated by that component platform."

This definition indicates three main component characteristics:

- Unit of deployment
- Unit of third-party composition
- No (externally) observable state

Some of the important implications of these characteristics are subsequently discussed. In order for a component to be a unit of deployment, it needs to be separated from other components and from its environment. Consequently, a component encapsulates all its constituent features. This means that internal component features, or construction details, should not be visible or accessible by any another component, or by any third party, including application integrators and deployers. In addition, as a unit of deployment, a component can never be only partially deployed. For a component to be composable with other components by a third party, the component has to clearly specify what it provides and requires. In other words, a component has to encapsulate its implementation and only interact with its environment via well-specified interfaces. The fact that a component has no visible state implies that it cannot be distinguished from copies of its own. Therefore, only one copy of each particular component is to be available in any given process. The reason for this is that replicas of the same component would be indistinguishable in the same process. As implied by their name, "components are for composition" [91]. This means that components can be reused and reorganized in various ways, resulting in different, new composites. Ultimately, components may or may not be built using Object-Oriented (OO) technologies. This means that a component may be built using classes, but also traditional procedures, functional programming approaches, assembly code, combinations of the above, or any other approach. The only constraint is that component characteristics must be respected.

Component-based System Architecture

System architecture is of utmost importance for the success of any large-scale software system. This includes systems built using component technologies. System architecture largely influences a system's quality attributes, including system performance and reliability [86], [85]. In the context of component-based systems, architecture specifies which are the permitted component types and their roles, the way they interact with their environment and the allowed interactions between them. Consistent component evolution and maintenance operations are highly dependent on the overall system architecture design. Devising and using the wrong architecture, or not being able to maintain the right architecture during system evolution is almost certain to cause project failure.

As presented in [91], a component system's architecture 'consists of a set of platform decisions, a set of component frameworks, and an interoperation design for the component frameworks.' System architecture can also be viewed as an abstract reusable model that can be transferred from one software system to the next. The main concepts related to system architecture are subsequently described: component platform, component framework and framework interoperation design.

Component Platforms

A component *platform* is an underlying layer that supports the installation of components and component frameworks, so that they can subsequently be instantiated and activated. Examples of component platforms include application servers (e.g. J2EE application servers: BEA WebLogic, IBM Websphere, JBoss and JOnAS), or operating systems.

Component Frameworks

Frameworks have emerged as means of developing complex middleware and software applications (e.g. [82] and [81]). The success of this type of application depends on the presence of certain application qualities, such as affordability, extensibility, flexibility, portability, reliability, or scalability. Three main framework characteristics are considered to facilitate the achievement of such qualities in software applications:

- A framework provides reusable software, decoupled from specific application software. This facilitates specific application customisations, while not allowing the framework's imposed interaction protocols and Quality of Service (QoS) properties to be violated.
- A framework provides an integrated set of domain-specific structures and functionalities. A well constructed framework models the commonalities of all applications in a certain domain, including common business processes, or graphical user interfaces). This allows all applications based on such a framework to take advantage of the previously acquired domain knowledge and previous experience. Frameworks leverage common solutions to reoccurring application requirements and design challenges. It is unnecessary to redevelop and revalidate such common solutions for each new specific application.
- A framework is a 'semi-complete' application that developers can use and customize for creating new specific applications, by extending the reusable framework components.

A framework definition is provided in [91]. According to this definition:

> A component framework is "a dedicated and focused architecture, usually around a few key mechanisms, and fixed set of policies for mechanisms at the component level."

Component Framework Interoperation Design

Similarly to component composition, component frameworks can also be composed. This is done in accordance to a framework interoperation design, resulting in a higher-level component framework [91]:

> "An interoperation design for component frameworks comprises the rules of interoperation among all the frameworks joined by the system architecture."

2.3.4 Component-Based Software Development (CBSD)

CBSD Definition

Component-Based Software Development (CBSD) is an emerging discipline in software engineering. It is largely considered as the next evolutionary step to Object Oriented Programming (OOP), for building large-scale software applications by integrating already existing software components [87]. The CBSD approach can potentially decrease development and maintenance costs, by increasing software reuse, flexibility and maintainability. Systems can be built rapidly by assembling previously existing software parts, or components. This approach is based on the central assumption that certain system parts (re)appear in different large systems with sufficient regularity. Consequently:

- Common parts should be written once and reused, rather than rewritten every time
- Common systems should be assembled from reusable parts, rather than repeatedly built from scratch.

Component-based systems encompass both commercial-off-the-shelf (COTS) components [88], as well as components acquired by other means (e.g. built in-house, or non-developmental items (NDI) [88]). CBSD is also referred to as Component Based Software Engineering (CBSE).

CBSD Motivation

Several economic and technological aspects provide the main motivation for the CBSD approach:

- Economic pressure to reduce system development and maintenance costs and time
- The increase in the number and the quality of COTS components
- The increasing amount of software in organisations, which can be reused in new software
- The emergence of component integration technologies, such as J2EE/EJB, CCM/ORB, COM/CLR, SOA and Web services

CBSD Process

In the CBSD approach, the focus has been shifted from the *programming* of software, to the assembling and *integration* of existing software components. Thus, integration, rather than implementation, has become the central activity in system development. As such, the CBSD process encompasses four major activities:

- Component qualification - the process of selecting components and determining their 'fitness for use'
- Component adaptation - the process of adjusting existing components to the particular running context in which they are integrated. This step is needed because existing components might be built to meet slightly different requirements and with somewhat different assumptions about their running context.

- Assembly of components into systems - the process of integrating components based on a well-defined infrastructure, or architectural style
- Component evolution - the process of adding, removing, or replacing components for error-removal, upgrading, optimisation, or addition of new functionalities.

Component Frameworks for Component-Based Software

The main objective of component technologies is the 'independent deployment and assembly of components' [91]. However, in order for individually-developed components to cooperate in a useful manner they have to comply with some common standard. Otherwise, component interconnections, or wiring, would be built ad-hoc, separately for each component pair. Such an approach would duplicate efforts and would be prone to errors, making the goal of component-oriented programming (COP) difficult to achieve.

Component frameworks have been devised to address this challenge. The purpose of specifying and building component frameworks is to impose certain requirements or standards on components. If an application is built based on a certain framework, then all components used to assemble that application must conform to that framework. Consequently, components that comply with a certain component framework are simply 'plugged' into the framework and can then seamlessly interoperate. Thus, a component framework is a software construct that establishes the environmental conditions for component instances and controls component instance interactions. In addition, component frameworks generally enforce certain policies on the instances of conforming components. Policies impose that instances of plugged-in components can only perform certain tasks via specific mechanisms, which are provided and controlled by the component framework (e.g. ordering of event multicasts). This approach helps prevent a number of classes of subtle errors that can otherwise occur.

The main component frameworks currently available on the market include the Enterprise JavaBeans (EJB) framework from Sun Microsystems, the CORBA Component Model (CCM) framework from OMG and the COM+/CLR framework from Microsoft. As indicated in [91] these are *contextual composition frameworks*. The contextual composition concept is discussed in the following section. Subsequently, section 2.4 presents the J2EE and EJB framework and technology for contextual composition. J2EE is the main technological choice of the presented dissertation research.

Component Frameworks for Contextual Composition

A *context* is defined by set of properties that characterise a number of constraints; all elements in a context must abide to the constraints that are defined by that context [91].

Composition refers to the process of assembling elements into an aggregate, without having to modify the elements involved. The formed ensemble is also referred to as a *composite*.

As stated in [91], it is impossible to predict the resulting properties of a non-trivial composite, unless the contexts of the composed parts are at least partially known. To achieve this, certain aspects of a context should be specified:

- The conditions to be met by a context in order to be considered well-formed
- The context composition rules. Each composition rule specifies the resulting properties of a composite formed by applying the defined composition operator, over a given set of elements, with given properties, inside a given well-formed context.

Component frameworks are an interesting application of contextual composition in the area of component software. In this area, the elements of composition are component instances. Composition operations combine component instances sets. Contexts contain a number of instances with analogous, execution-wide properties. From this perspective, a mechanism that supports contextual composition of attributed components can be seen as a composition framework [91], also referred to as *contextual composition framework*.
In such frameworks, instance composition involves creating contexts and placing instances in the right contexts. A component instance placed in a context is accessible from outside its context. However, the context gets an opportunity to intercept all messages, or requests, crossing the context boundaries. The context objects that intercept such requests remain invisible to component instances both outside and inside the context. Contextual composition frameworks can be created for properties such as transactional computation, security, or load balancing. Such properties are sometimes also referred to as aspects as they are likely to crosscut the system, resulting in system qualities that are not provided by any individual component.
Current technological support for contextual composition in component software includes COM apartment model, Microsoft Transaction Service (MTS) contexts, EJB containers, COM+ contexts, CCM containers, and Common Language Runtime (CLR) contexts. For the presented research, the EJB technology and its associated contextual composition model was adopted. The main motivation for this choice is the extensive adoption of the EJB technology in the industry, compared to other available technologies and consequently the high availability of application server implementations and related tools. The EJB technology is shortly introduced in subsection 2.4.2.

2.4 J2EE - Component Technology for Enterprise Systems

2.4.1 Introduction to J2EE

J2EE is the component technology standard specified by Sun Microsystems for building multi-tiered enterprise applications. The multi-tier distributed model that J2EE uses generally includes a presentation tier, an application logic tier, and a data persistence tier. Clients of a J2EE system can be Java applications, or web browsers. J2EE specifies different component types for implementing the various enterprise application tiers, such as servlets for the presentation (or web) tier and Enterprise JavaBeans (EJBs) for the application tier. The data storage (or persistence) tier contains the existing applications, files, and databases. For the storage of business data, the J2EE platform requires a persistence mechanism, such as a database that is accessible through the JDBC, SQLJ, or JDO API. The database may be accessible from web components, enterprise beans, and application client components. Figure 2.1 illustrates the tiers in a typical J2EE scenario.

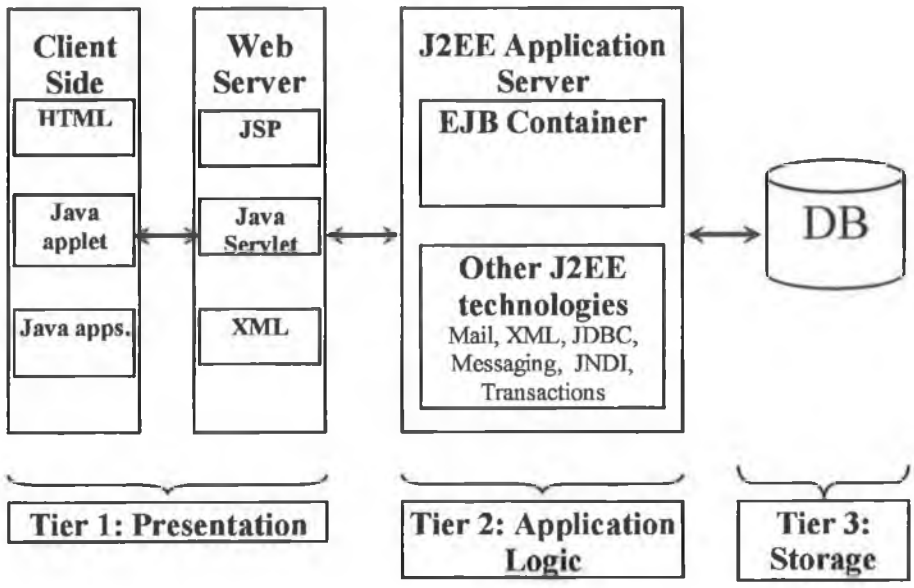

Figure 2.1: application tiers involved in a typical J2EE system: presentation tier, application tier and data persistence tier

2.4.2 Enterprise JavaBeans (EJB)

Part of the J2EE specification, *Enterprise JavaBeans (EJBs)* represent server-side reusable software components that encapsulate application business logic. The programming language of choice for EJB is Java. As EJB is a component-based technology, its main purpose is to simplify and reduce the costs of the development and management processes of large-scale, distributed applications, such as enterprise applications. Therefore, developers can use EJB to build scalable, reliable and secure applications without having to devise complex distributed frameworks and middleware services. EJB provides the distributed platform support and common services such as transactions, security, persistence and lifecycle management. In short, developers implement the application business logic, which is stored into EJB components (Figure 2.2). Subsequently, EJBs are deployed and managed by EJB containers, as part of a J2EE application server. EJB containers provide middleware services and manage the EJB lifecycle during runtime. These processes can be configured via xml documents, referred to as EJB deployment descriptors.

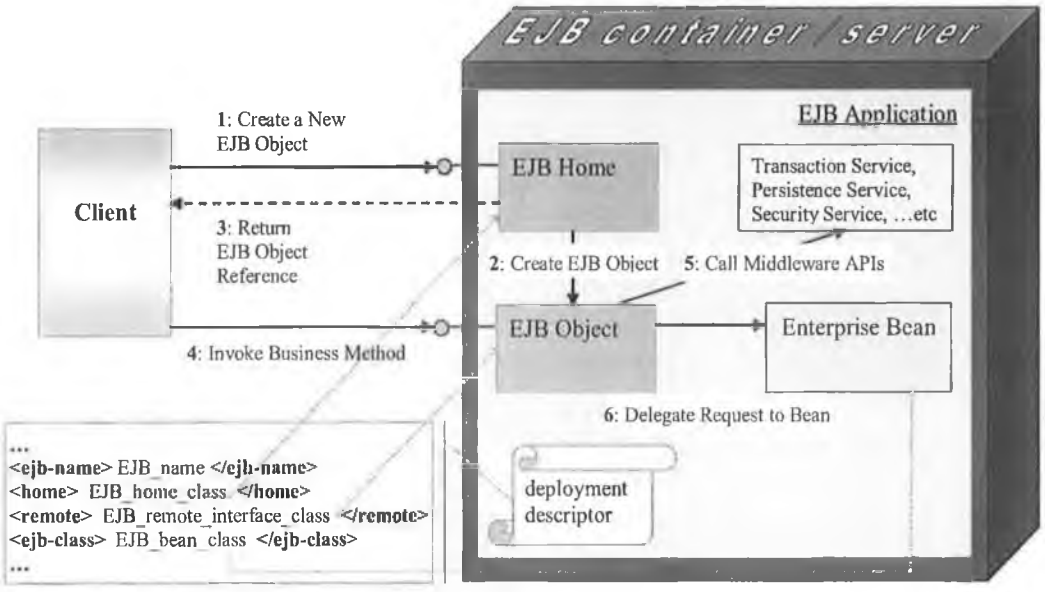

Figure 2.2: EJB infrastructure:
accessing EJB components via the EJB container and application server

EJB provides a specification for EJB containers and the services they offer, for EJB components hosted by such containers and for the way containers and EJB components interact [80]. In addition, EJB provides a set of Java interfaces. Both the EJB components and the EJB application servers must conform to these interfaces. Consequently, any EJB application server can manage any EJB component(s) and any EJB component can interoperate with other EJB components.
The main terms and concepts of the EJB component technology are described over the following sections. The main benefits of using EJB for building enterprise applications are also discussed.

EJB Fundamentals
An *enterprise bean*[11] is a server-side software component that encapsulates application business logic and that can be deployed in a multi-tiered distributed environment [80]. In the context of component technologies, an enterprise bean is a J2EE component that implements the Enterprise JavaBeans (EJB) technology.
At runtime, EJBs are managed by EJB *containers*, which represent runtime environments within the J2EE server (Figure 2.2). EJB containers provide system- or middleware-level services to deployed EJBs, including distribution, transaction support, security, or lifecycle management. Such services are not directly implemented as part of the system's business logic specification. However, they are a critical part of any enterprise system's implementation. Using the J2EE technology to build distributed enterprise systems provides a clear separation between the business logic (implemented in the EJBs) and the middleware services (provided by the EJB application server). As a result, application reusability and manageability are

[11]J2EE Tutorial from Sun Microsystems: java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBConcepts.html

improved and the implementation and management costs significantly reduced. Application programmers can focus on implementing the business logic functionalities and not worry about the system infrastructure-related aspects. The required middleware services are provided and managed by the J2EE servers instead. The exact manner in which a server container is to invoke middleware services and manage individual EJBs at runtime can be configured via XML *deployment descriptor* files. EJB deployment descriptors are bundled together with the EJB business logic implementation classes into deployable component packages, or archives.

An important consequence of the way the EJB technology is designed is that in EJB applications all client calls to methods of EJB instances must go through the EJB container [Figure 2.2]. As such, clients can never access EJB instances directly. This provides a good opportunity for intercepting, analysing and processing all accesses to EJB instances, at the server container layer.

Enterprise beans are built using an Object technology, in Java. Hence, one enterprise bean can be composed of one or more Java classes. In addition, it may contain various component resources, such as a number of immutable objects capturing default initial state [91]. At runtime, Objects instantiated from classes of an enterprise bean can become visible to clients of this enterprise bean. For example, clients can consist of other enterprise beans, servlets, applets, or plain Java applications. Clients can acquire references to EJB instances and use them by means of method invocations.

When clients access and use an enterprise bean instance, they deal with a single exposed component interface. This is regardless of the composition of the targeted enterprise bean. This interface, as well as the enterprise bean itself, must conform to the EJB specification. Conforming to the EJB specification, the required interface must provide a number of compulsory methods, which allow the EJB container to consistently manage all enterprise beans, during runtime.

Enterprise JavaBean Types

Starting with EJB 2.0 specification, three types of enterprise beans are defined. The first type, *Session beans*, is used for modelling business processes. *Entity beans* are used for modelling business data. Finally, *Message-driven beans* are used for modelling asynchronous business processes. The three EJB types are briefly described below.

Session beans are action-oriented, in that their main purpose is to execute tasks for clients. A session bean instance has only one client at a time. Session beans are not *persistent*, which means that session bean data is not saved to a persistent storage. When the client of a session bean terminates the session bean is no longer associated with that client and it can be removed, or it can be stored in an instance pool and latter associated with a different client.

There are two types of session beans. One type is *Statefull Session beans*. The state of a statefull session bean object is retained while the session between the bean and its client lasts. The state of an object is considered as composed of the object variables values. Because a client is considered to 'talk' to a session bean during a session, the session bean state is also referred to as *conversational state*. When the session between a session bean and its associated client is over, the conversational state vanishes. The second type of session beans are *Stateless Session beans*. Stateless session beans do not maintain conversational state for a particular client. A stateless bean may contain state, but only during the execution of one client method

invocation. When the method execution is over, the state vanishes. This implies that all stateless bean instances are identical, except during method execution. This allows the EJB container to assign any stateless bean instance to any client, potentially improving application scalability. Generally, less stateless bean instances (than statefull bean instances) are necessary for supporting the same number of clients.

Entity beans represent data objects whose states are stored and maintained in a persistent storage. In enterprise applications, data objects generally represent business entities, such as orders, products, accounts, or customers. Various persistent storage types can be used for maintaining the state of entity beans. For example, the J2EE Software Development Kit (SDK) provided by Sun Microsystems as a (proof-of-concept) reference implementation of the J2EE platform, uses relational databases as the persistent storage mechanism. An entity bean is generally associated with a table in a relational database and each entity bean instance represents a record in this table. The state of an entity bean is *persistent*, since it is saved in a storage mechanism, such as a relational database. As such, *persistence* means that the state of an entity bean instance exists beyond a client session, a J2EE process, or the lifetime of the software application.

Two persistence types are defined for entity beans. The first persistence type is *Bean-Managed Persistence (BMP)*. When BMP is used, the entity bean class contains the code for accessing the persistence storage. In case relational databases are used, the entity bean will contain the Standard Query Language (SQL) calls needed to access and manipulate persistent data. The second persistence type is *Container-Managed Persistence (CMP)*. When CMP is used, the EJB container automatically generates the code for accessing the persistence storage (e.g. database).

Multiple clients can share (or have the impression of sharing) the same entity bean instance. For example, in an e-commerce application multiple clients can concurrently access a certain product entity bean, in order to get product information or buy a product item. Multiple users may be buying the same product at the same time, each user separately modifying the number of available product items in stock. Business managers may want to change the product price, description, or number of available items, in parallel. Therefore, as different clients may want to change the same data, it is important that entity beans work within *transactions*. The EJB container typically provides transaction management. The enterprise application's deployer is responsible for setting the transaction attributes in the entity bean's deployment descriptor. Each entity bean instance has a unique object key, also referred to as a *primary key*. This enables a client to locate and access a particular entity bean instance.

An entity bean may be related to other entity beans, similarly to the way relational database tables can be related to other tables in the same database. For example, in the e-commerce application example, a customer entity bean may be related to an account entity bean. Or, a shopping-cart entity bean can be related to a product entity bean.

Message-driven beans allow a J2EE application to process messages asynchronously. Message-driven beans rely on the Java Message Service[12] (JMS) technology. A message-driven bean acts as a message listener, similar to the way event listeners listen for events. Messages can be sent by any J2EE component, such as another enterprise bean, an application client, or a web component. In addition, messages can also be sent by other JMS applications that do not

[12]Java Message Service (JMS), from Sun Microsystems:
http://java.sun.com/products/jms/

use the J2EE technology. Message-driven beans can currently process only JMS messages, but this might change in the future, allowing message-driven beans to process other message types. Entity beans and session beans can also send and receive JMS messages, but only in a synchronous manner.

Accessing Session and Entity Enterprise JavaBeans

A client may only access an enterprise bean through the bean's interfaces. The client is unaware of other features of the bean, such as bean method implementations, database access calls and abstract persistence schemas, or deployment descriptors. Consequently, as long as a bean's interfaces do not change, the bean can change internally without affecting its clients.

The EJB specification defines two types of client accesses to enterprise beans: *remote* and *local*. Clients that access an enterprise bean using the *remote access* type may run on the same machine as the enterprise bean they access, on a different machine, or on a different Java Virtual Machine (JVM). They do not have to be aware of the location of the enterprise bean they want to access. Remote clients can be Web components, J2EE application clients, or other enterprise beans.

In order to create an enterprise bean that supports remote access, a *remote interface* and a *home interface* have to be provided for that bean, by the bean developer (Figure 2.2). The *remote interface* contains the business methods of the enterprise bean. The *home interface* defines the life-cycle methods of the enterprise bean, including 'create', or 'remove' method(s). For entity beans, the home interface may also contain finder and home methods. When clients use an entity bean, they never invoke bean methods directly on an actual instance of the bean class. Rather, the EJB container *intercepts* client method invocations and *delegates* them to the bean instance. Intercepting method requests enables the EJB container to perform (middleware-specific) services, such as transaction management, security, resource and component life-cycle management, persistence, remote accessibility, or monitoring. Hence, the EJB container acts as an intermediate layer between clients and enterprise beans. This layer of indirection is referred to as the *EJB Object*. The EJB Object is thus part of the EJB container. It performs the intermediate logic that the container requires before a method call can be serviced by a bean instance.

An EJB Object replicates and exposes every business method that a bean class itself exposes. Clients of an entity bean call methods on the EJB Object rather than on the actual bean instance. EJB Objects subsequently delegate all incoming business calls to the corresponding enterprise bean instances (Figure 2.2). One EJB Object is automatically generated for each enterprise bean. The EJB Object creation process is different depending on the application server implementation, from different vendors. The EJB Object implements the bean's remote interface, as specified by the enterprise bean provider.

A client cannot instantiate an EJB Object directly, as the client and the enterprise bean to be instantiated may be on different machines, and/or in different JVMs. In addition, the client does not need to know the location of the enterprise bean to be instantiated. Therefore, clients can only obtain EJB Objects from an EJB Object *factory*, also referred to as *Home Object*, or *EJB Home*. The main responsibilities of a Home Object include creating, removing and finding existing EJB Objects. Like EJB Objects, Home Objects are generated automatically for each enterprise bean in a vendor-specific manner. The Home Object implements the Home Interface, specified by the enterprise bean provider.

Clients that access an enterprise bean using the *local access* type can only run in the same Java Virtual Machine (JVM) as the enterprise bean they access. They have to be aware of the location of the targeted enterprise beans. Local clients can be Web components or another enterprise bean. In order to create an enterprise bean that supports local access, a *local interface* and a *local home interface* have to be provided, by the enterprise bean developer. The *local interface* contains the business methods of the enterprise bean and the *local home interface* contains its life-cycle methods.
Local access to entity beans is faster and more efficient than remote access. As both the client and the entity bean are required to reside in the same JVM, actions needed for inter-JVM communication in the remote EJB access type are no longer needed in local call procedures. More precisely, local access is performed directly between application-level objects, via object references, in the same memory address space. Thus, local access does not require mapping of requests between the application layer and the lower-level network layers, as it is the case with remote communication. Additional operations needed when remote access is used typically include request marshalling and de-marshalling, as well as network connection management operations.
A client can call an enterprise bean locally, by using its *local Object* instead of its EJB Object. A local Object implements a bean's *local interface* rather than its remote interface. Creating entity bean instances for local access is also faster than creating instances for remote access. Local EJB instances are created by calling the EJB's *local home interface* rather than its remote home interface. A *local Home Object* implements the local home interface.

2.4.3 Enterprise JavaBeans - Important Characteristics for Performance Management

Performance Management Challenges in EJB Systems
Numerous challenges remain to be addressed for achieving autonomic performance management in the area of Internet-based enterprise systems. The particular characteristics of such complex software systems cause difficulties in applying existing approaches from related research domains. One such characteristic is the increased dynamicity of runtime component instances and their interconnections, which may cause the runtime application architecture to significantly differ from the development time design. While this may happen in other systems types, where components are represented by entire applications or servers, the frequency and extend of such dynamic modifications are typically reduced when compared to EJB applications. In addition, the complexity of the implemented business logic and of the underlying platform layers (e.g. application server, JVM, OS), as well as the possible lack of access to their implementation, make it difficult to understand and predict the emerging system performance behaviour during runtime. It is highly expensive to build accurate performance models that indicate an EJB application's performance characteristics and behaviour, at the EJB component level. Some of the most important EJB characteristics that influence the design of performance management solutions are presented below.

EJB - Separation of Concerns
A significant consequence of the EJB specification is that in EJB applications, the life

cycle management of deployed EJB components is entirely managed by application server containers. This is different from other technologies, for example plain Java, CORBA, or Fractal[13], where the instantiation, caching and removal of runtime entities such as objects or component instances are the exclusive responsibility of application programmers. In such cases, the code for managing the lifecycle of runtime entities is mixed with the business logic code of those entities. For instance, in order for a client application to be able to use a component's provided functionalities, as advertised by the component's interface, the client must first instantiate the component as part of its own code. Then the component's business methods can be called on the created component instance. In contrast, in the EJB technology, the business logic and lifecycle management functionalities are clearly separated. Business logic is implemented in the EJB classes, whereas lifecycle management functionalities are provided by the application server and managed by EJB containers. Clients must also obtain an EJB instance before calling methods provided by that EJB. However, the EJB instantiation, pooling and caching operations are no longer the client's responsibility. Instead, the client merely asks the application server for an EJB instance and then uses the retrieved instance as before. However, in this case, the actual manner in which retrieved EJB instances are being acquired is completely transparent to clients. Such tasks are the exclusive responsibility of the application server container. Upon receiving a client request for returning an EJB instance, the server container may create a new instance, or retrieve an existing instance from the appropriate EJB instance pool, or instance cache. Similarly, when an EJB instance is no longer needed by a client, the server container may decide to destroy the instance, or return it to the EJB instance pool or cache for future use.

Certain lifecycle management aspects can be configured via EJB deployment descriptors, including EJB instance pool sizes, or EJB instance caching policies. However, the exact EJB instance management behaviour is generally not known. For example, in case a number of EJB instances are being required by clients, the EJB container may decide to perform a bulk instantiation of 100 EJB instances and store them in the EJB instance pool. As EJB instantiations are generally costly operations, storing EJB instances in a pool and having them ready to use upon request can considerably increase system performance. In a similar manner, if no client requests are received for EJB instances during a certain period, the server container may decide to downsize the EJB instance cache and pool, in order to free system resources (i.e. memory). Nonetheless, the application server and container behaviour can significantly differ depending on the application server implementation and configuration (e.g. different behaviours in the IBM WebSphere, BEA WebLogic, Sun One, JBoss and JOnAS J2EE application server implementations). These considerations also hold for other system-level functionalities, such as transaction support, security, or persistence. While in certain technologies these services need to be provided by programmers along with the business logic code, in the EJB technology middleware-services are provided by the application server, separately from the application business logic.

EJB - Connectivity Specification

Regarding the component interconnection specification in the EJB technology, it is advisable that all the *dependencies* of an EJB component on other EJB components be documented in the EJB deployment descriptor document. This is in order to promote reusability of compo-

[13]The FRACTAL Project, from Object Web: fractal.objectweb.org

nent interconnections between deployments in different environments. However, such EJB dependency specifications are neither compulsory to provide, nor do they necessarily mean that instances of the indicated EJBs will actually interconnect during runtime. A dependency specification means that instances of an EJB may bind to instances of the EJBs declared in its descriptor. Nonetheless, an EJB instance may also bind to instances of other EJBs, whose names (i.e. JNDI identifiers) can be received dynamically at execution time. This situation may particularly occur in workflow engine applications, in which core business logic delegates operations dynamically to other business logic, based on input parameters. Such a situation contrasts with component technologies such as C2 [94], or Fractal, in which runtime component interconnections are clearly specified via special-purpose configuration files (i.e. ADL scripts). In these technologies, when the component interconnections change during runtime, the corresponding ADL description must accordingly change to reflect the new application state. Providing such information is not required by the EJB specification and is therefore unavailable in the current EJB implementations. In addition, frequently and extensively updating the ADL specification of an EJB application at runtime, so as to accurately reflect its dynamically changing runtime architecture, may raise serious scalability concerns.

2.5 J2EE Application Servers

For the scope of the thesis, an *application server* is considered to be a software product for component-based, server-centric architectures. In this context, application servers reside between server-side software applications and the underlying software (e.g. JVM, OS) and hardware platforms. Their role is to provide middleware services to deployed applications, such as transactions, security, distribution and persistence. Java application servers are based on the Java 2 Platform, Enterprise Edition (J2EE). More precisely, J2EE application servers are those that have been certified by Sun Microsystems as being fully compliant with all the J2EE capabilities.
The JBoss application server was used as the J2EE platform for the implementation and experimentation work of the thesis. This choice was based on the fact that JBoss is currently the most popular open-source J2EE application server available. The JBoss server characteristics relevant to the thesis experimental work are presented in section 4.1.

2.6 Automatic System Management

This section presents some of the main concerns and the most relevant approaches in the area of automatic management of complex software systems. Existing research in the area can be categorised based on its goals and adopted strategies. An important research direction targets the design and implementation of automatic frameworks for dynamically managing, adapting and optimising software applications. Research projects in this category generally comply with the adaptive and autonomic system initiatives.
Complexity is currently becoming a prime concern in managing software systems. Software complexity has emerged as a consequence of ever more elaborated development processes, increasing system scale and growing requirements for system transformation support. Com-

ponent technologies have been increasingly adopted for building large-scale software systems, as they successfully address complex functionality and flexibility issues and reduce development and maintenance costs. Nonetheless, current component technologies (e.g., J2EE, CCM, or .NET) provide little support for predicting and controlling the emerging performance of software systems that are assembled from distinct components. Static component testing and tuning procedures are typically run in isolation or simulated environments. Although important, such procedures provide insufficient performance guarantees for components that are to be run in diverse component assemblies, under unpredictable workloads and on different platforms. In addition, due to the highly dynamic nature of today's software systems, the environmental conditions in which a component runs are likely to periodically change during the component's lifetime. Such environmental conditions include the incoming workload and the software and hardware resources available to a component. Changes in these running conditions can significantly impact a component's availability and performance characteristics, which include component's throughput and response times. For these reasons, it is essential that software systems and their constituting components are able to seamlessly adapt during runtime to the constant changes in their business requirements and execution environments. However, due to system complexity, manually performing such tasks becomes an exceedingly costly and error prone process. An essential need for automating the system management processes has been consequently identified. The Autonomic Computing (AC) initiative has emerged to address stringent requirements for self-management capabilities in complex software systems.

This section provides a general overview of the main research directions in the area of performance management in large-scale component-based systems. Performance must be considered during the entire system lifecycle, starting with system design time and continuing through system runtime as a constant, repetitive process. Several approaches related to design- and run-time performance planning were presented in subsection 2.2.1. This section focuses on analysing possible approaches for implementing the main functionalities required for automatic management support during system execution.

Some perspectives on viable alternatives for implementing self-management capabilities in complex component-based applications are presented. The intent is to highlight the important aspects to consider when designing, or porting management solutions used for administering software systems from different domains. Different solutions are best suited for building self-management capabilities into different software system types, with different component technologies and at different component granularities. In certain cases, a particular management approach may not be viably ported across certain system types. The reason is that the particular characteristics of managed entities in various systems can raise distinct problems, which frequently require custom solutions.

The rest of this section presents some of the main research directions in the area of automatic performance management for software systems. It reviews some of the main approaches towards implementing the key functionalities of automatic performance management, including monitoring, analysis, optimisation decisions and adaptation execution. It also discusses the suitability of the existing approaches towards managing enterprise systems built using component technologies based on contextual composition frameworks (e.g. EJB). The focus is on solutions for managing component-based enterprise systems and on the J2EE component technology in particular. Discussions on automatic self-management procedures are preceded by a short presentation of the autonomic computing initiative.

2.6.1 Autonomic Computing

Autonomic computing is IBM's research initiative towards solving the *software complexity problem*[14]. The main goal is to reduce the increasing complexity of managing large, distributed computer systems. The idea behind autonomic computing is to enable software systems manage themselves, according to high-level administrative objectives [54], [42]. Autonomic systems would thus be able to maintain and adjust their operation in response to changes in various system modules, in the incoming workloads and resource availability and in the system functional and quality requirements. Ideal autonomic systems "just work, configuring and tuning themselves as needed" [42].
Autonomic computing is regarded as an emblematic term, encompassing a collection of technologies, for different computing system elements - from small devices to large-scale networked systems, from different disciplines, including both software and hardware domains, and the integration of these technologies in order to make self-managing systems possible. The main aspects advocated by the autonomic computing initiative as part of a self-management solution include self-configuration, self-optimisation, self-healing and self-protection. Research in each of these areas contributes to the overall autonomic computing vision. The presented research work is consistent with the autonomic computing initiative, focusing on the self-optimisation functionality of component-based enterprise systems.

2.6.2 Separation of Concerns

An important part in the design of any system management solution is the degree of separation between the management adaptation logic and the system application code. This aspect mainly concerns situations in which adaptation actions involve changes in the software application behaviour, for optimising performance, or for fault-tolerance. In such cases, several behaviours are typically available and the adaptation logic is used to decide which behaviour to use at each time. Various solutions for implementing this conduct are available, differing in their flexibility and manageability characteristics.
The most tightly-coupled management solutions imply the use of monolithic components. In these solutions, the adaptation logic is mingled with the business logic code in a single component implementation [78]. All possible behaviours are implemented as part of the same component. "If - then - else" policy constructs are used to implement the adaptation logic that selects between the available behaviours. Such solutions are limited by their lack of flexibility, manageability and reuse. The reason is that the entire monolithic component needs to be understood, modified and recompiled whenever a behaviour or the adaptation logic needs to be altered in anyway (i.e. addition, update or deletion operations).
Improved modularity is obtained when design patterns [40] are used to separate different behaviours from each other and also from the adaptation logic that selects between them (e.g. the Strategy pattern [40]). Such solutions allow both the behaviours and the adaptation logic to be independently modified. Nonetheless, these need to be designed into the software application when the application is built, since they cannot be added or removed at application deployment or runtime. In addition, in order to modify the application's behaviour and adaptation

[14] Autonomic Computing: IBM's Perspective on the State of Information Technology: www-1.ibm.com/industries/government/doc/content/resource/thought/278606109.html

logic, the entire application needs to be stopped, the targeted parts modified and recompiled and the application restarted. The actual pattern implementation itself cannot be reused and needs to be re-implemented for each new application.

Management frameworks, completely separated from the managed applications have been proposed to provide full separation between the managed application and the management adaptation logic. Such frameworks typically provide uniform, automated management support to all applications that comply with a certain standard or specification (e.g. applications implemented using a certain technology). General management frameworks, as well as several specific framework implementations are presented in [72], [54], [43], or [73]. Automated frameworks are able to dynamically monitor applications for extracting runtime data on the system metrics of interest, analyse current system state and possible remedial or optimisation actions and finally decide and enforce adaptation strategies into the running application (Figure 2.3).

The implementation or configuration of a management framework can be modified completely independently from the underlying managed applications. Management frameworks are also used for maximum flexibility and manageability of cases in which multiple possible behaviours can be selected as means of application adaptation. Namely, separating different behaviours in individual components and employing a suitable management framework for selecting amongst the available components significantly increases flexibility and reusability for the entire managed system. In addition, clearly separating a management framework's functionalities into different modules, such as monitoring, adaptation logic and adaptation execution functions, further improves system modularity. The various modules can consequently be updated independently from the other framework modules, without affecting the managed applications' implementation.

Subsequent sections discuss a number of important aspects to consider when building automated management frameworks. Several existing or possible approaches with their main characteristics and implications are discussed.

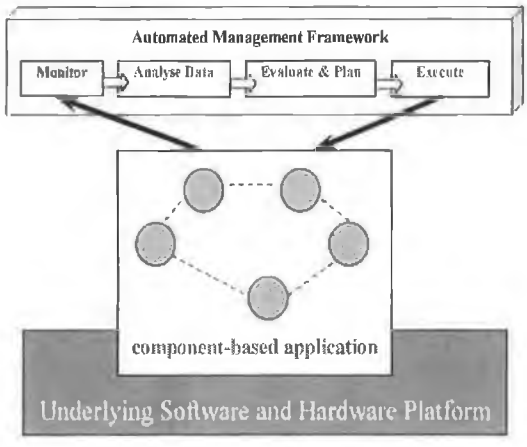

Figure 2.3: automated management frameworks -general logical architecture

2.6.3 Granularity of Managed Entities

Complex software systems can be managed at various component granularities. In the context of Internet-enabled enterprise applications, managed entities can represent single components (or component groups) implementing application business logic (e.g. EJBs in COMPAS [66], or JAGR [20] frameworks), middleware services (e.g. transactions or security services in jonasALaCarte [1]), or entire web or application servers (e.g. Rainbow [43], or JADE [73] frameworks). While managing a system at lower granularity levels provides the opportunity for more accuracy and control, it can also induce higher management overheads; more complex adaptation logic may be required to prevent management operations from inducing excessive performance degradation. For this reason, certain centralised approaches successfully applied to manage coarse-grained system components (e.g. servers) may not scale well when applied for managing fine-grained components (e.g. EJB instances).

2.6.4 Management Control Logic - Topology and Organisation

Several options are available for designing the architecture and control logic of management frameworks. The most relevant approaches are based on centralised and respectively decentralised management control. Hierarchical topologies, combining the characteristics of the centralised and decentralised approaches are also possible and in fact recommended for managing certain system types [39]. The three types of topologies are illustrated in Figures 2.4, 2.5 and 2.6 and discussed in more detail below.

When *centralised control logic* is used (Figure 2.4), a single management control entity collects and analyses all monitoring data and makes all adaptation decisions for all managed entities in the system. An *application model* is typically used to provide an abstract representation of the managed application. Models are normally represented as directed graphs, where graph nodes represent application runtime entities (i.e. component instances) and links indicate runtime communications between these entities (i.e. method calls between component instances). In some cases, models can initially be built based on available source code information of the managed application. At runtime, monitoring data is used as necessary to update the centralised model, so as to maintain an accurate system representation while changes occur in the base running system. Reflective middleware and component technologies have been proposed to support this type of model-based approaches (e.g. JADE [73], Rainbow [43], K-Components [37], Arctic Beans[15] or IguanaJ [79]). Centralised solutions impose that an abstract meta-model be built and maintained as an accurate reflection for the managed application. Any change in the base layer, or running application, must be reflected in the meta-layer, or the application meta-model. Similarly, application adaptation operations are initially performed at the reflective system meta-level, upon the application model, and then automatically reflected at the base level, in the actual running application.

The viability of a centralised, model-based approach critically depends on the targeted system scale and on the frequency and extent of dynamic system transformations that occur during runtime. Application scale refers, in this case, to the average number of component instances

[15] Arctic Beans project: abean.cs.uit.no

available at runtime. Dynamic system transformations refer to the changes that occur in the running managed application and should therefore be reflected in the application meta-model. Such changes can represent the creation, reconfiguration or removal of component instances, or component instance interconnections. Scalability issues may occur when adopting centralised solutions for managing large applications, with hundreds of component instances and a high frequency of multiple instance creation, deletion and interconnection events [39].

On the other hand, centralised solutions have successfully been used to manage component-based applications with a limited number of runtime component instances and relatively rare dynamic changes. For example, in the JADE [19] and Rainbow [43] projects, managed components represent software servers in distributed enterprise applications. In such systems, the number and type of component instances (e.g. servers) as well as the initial interconnections among these instances are generally known at deployment time. Also, the frequency of server addition, removal, or interconnection operations is limited. In pervasive computing research projects (e.g., [79]) component instances represent embedded devices. In this type of systems, the number of devices that enter or leave the pervasive system is also limited and the occurrence of such events manageable. Nonetheless, scalability problems are likely to occur when using a centralised approach for managing large-scale, dynamic systems at finer granularities. A good example of this situation constitutes the management of J2EE systems at the EJB component level (e.g., [66] and [99]).

As previously discussed in section 2.4.3, in EJB applications, the application business logic implementation is completely separated from the middleware services. The combined complexity of both the business logic and middleware services makes it hard to determine or predict the exact system runtime behaviour. In consequence, EJB models at the EJB instance granularity level can only be built and updated based on runtime monitoring information. Nonetheless, the fluctuating EJB instance numbers, caused by complex lifecycle behaviour, as well as the fine granularity of instance interconnections would cause centralised models at the EJB instance level to not scale well for most enterprise applications.

Scalability could be improved if centralised models were built at the EJB component level rather than the EJB component instance level. This means that a node in the directed graph model would represent one EJB component, rather than one EJB runtime instance. Thus, a single node would be used to represent all instances of a certain EJB component. The number of actual EJB instances available for each component can be provided as an attribute of the corresponding EJB node in the model. This approach was taken for example by adaptive monitoring and management frameworks such as COMPAS [66]. Such solutions decrease the number of model update operations performed during runtime, because events related to component instance creation and removal do not cause structural changes; rather, they only cause node attribute value changes. In such cases, only changes in the actual set of deployed EJB component classes are considered. The set of EJB classes deployed and used at runtime, as well the runtime interconnections between EJB instances can be automatically detected in any EJB-compliant application server, via EJB container instrumentations.

In centralised management approaches, adaptation decisions are generally based on an overall evaluation and optimisation of the application model. Adaptation decisions are then enforced into the running system, so that all modifications performed at the meta-model level are reflected in the actual base application. Nonetheless, globally evaluating and optimising the entire application each time a performance problem or optimisation opportunity is detected at the component level, may consume unnecessary resources and not scale well.

A *decentralised approach*, where each managed component is evaluated and optimised independently from other components provides a viable solution to the scalability problem. However, exclusively focusing on local component optimisations may lead to non-optimal global configurations for the overall application. A combined solution - *hierarchical control* topology - seems to be the answer to this problem, using local optimisations when possible and employing global optimisations when necessary [39, 27]. The tradeoffs are between optimal solutions provided by global, centralised adaptation processes versus the better scalability featured by decentralised approaches.

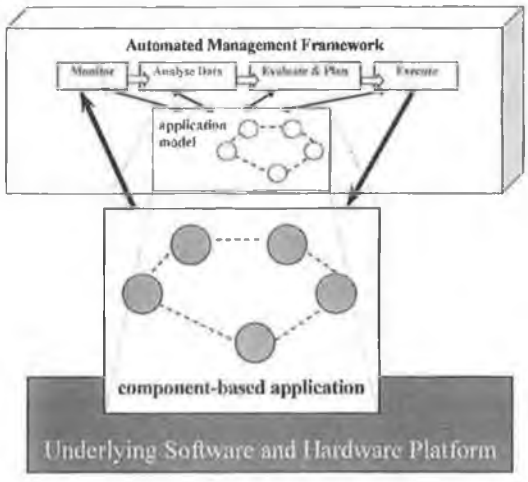

Figure 2.4: centralised topology (model based)

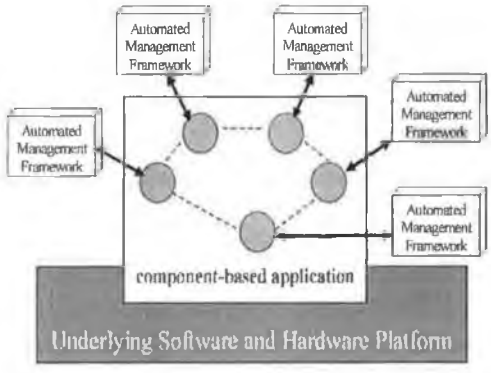

Figure 2.5: decentralised topology

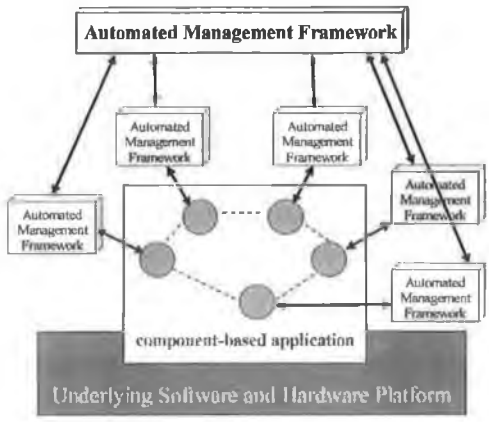

Figure 2.6: hierarchical topology

2.6.5 System Instrumentation and Monitoring Considerations

System instrumentation is directly related to runtime monitoring. It refers to the process of modifying a system so as to be able to extract runtime monitoring information on the metrics of interest. This is typically performed by inserting some sort of hooks into the system, at the particular points of interest. Instrumentation hooks are used to intercept and monitor system activity at the targeted points. Events of interest may include the occurrence of incoming client requests, outgoing responses (including exceptions thrown), as well as component instance creation and removal operations. Based on these events, performance and availability metrics such as response time, throughput, or the occurrence of faults can be computed. In addition, instrumentation hooks can also be used to control the system, by redirecting or delaying system requests, or by replacing or reconfiguring system parts.
System instrumentation also refers to that part of a system that allows monitoring information to be obtained, and control operations to be performed on the system during runtime. This section discusses some of the important design and implementation aspects of system instrumentation for automatic system management. The discussion is targeted at systems built using contextual composition frameworks, specifically considering the EJB component technology.
In order to be able to control applications, management frameworks need to be able to extract runtime data from the executing system, analyse this monitoring data, evaluate possible remedial actions and eventually be able to enforce adaptation decisions into the running system. Control hooks for extracting monitoring data and modifying running applications need to be available in the system in order to enable management frameworks to control these systems.
Several approaches are possible for providing management hooks, depending on the architectural layer where the hooks are being inserted. A first approach is to instrument the system at the software application level. This is achieved by inserting proxies, or wrappers, in front of all component entities to be managed. This way, all incoming and outgoing calls, to and from the managed components, can be intercepted at the proxy level. Hence, data can be extracted from intercepted calls and sent to the management framework. Monitoring data so obtained

can include performance data (e.g. response times, throughput), fault-related data (e.g. occurrence of exceptions), call path data (e.g. the sequence of component method calls in a business transaction), or component life cycle related data (e.g. number of instances available and used for a certain component). As previously indicated, the instrumentation proxies can also be employed for performing control operations on the managed components. Such operations may involve redirecting incoming client request for load-balancing purposes, or delaying requests during component versioning operations.
When application-level instrumentation is used, proxies need to be created and inserted into the managed components for each separate application to be administered. For this reason, it is more than desirable for the application *proxification* process to be performed in an automatic manner, seamlessly for all targeted applications. A viable solution that provides such capabilities for J2EE applications, at the EJB component level, in a portable and non-intrusive manner, is provided by the COMPAS project [66, 68].
A second approach for instrumenting managed systems is to insert instrumentation hooks into the application server, or middleware layer, or into any of the underlying layers on which the managed application is deployed and executed. In J2EE for example, all incoming and outgoing calls to software components (e.g. servlets or EJBs) are intercepted and managed by the corresponding server containers (e.g. servlet containers in web servers, or EJB containers in application servers). Thus, server containers can be modified as needed to intercept calls and provide monitoring information and adaptation control on the managed components [20]. In this approach, the server containers provide the same functionality that the component proxies do in the application-level instrumentation approach presented before. Any middleware platform can be instrumented using this general technique. This is because the main role of middleware platforms is to provide non-functional services to application entities, including distributed communication, security, or transaction support. Service provisioning is achieved by intercepting and managing client calls to the managed components, at the middleware level, as necessary. Thus, adding management capabilities as a novel middleware service fits naturally with this design. Middleware-level instrumentation solutions only need to be implemented once for the targeted middleware platform and can then be used for managing all software applications deployed and run on that platform. However, this solution is not portable across middleware products, or application servers. A separate implementation must be provided for each different middleware platform used.

2.6.6 Adaptation Logic - Strategies, Design and Implementation

Adaptation logic dictates the way automated frameworks take their management decisions. It directs the decision processes for analysing input data, detecting performance anomalies, planning and evaluating possible remedial actions and triggering application adaptation operations. In short, adaptation logic determines when to adapt applications and which operations to initiate as part of the adaptation process. In more complex management frameworks, adaptation logic can also be used to adapt the management framework itself. This includes for example decisions on when to start and stop application monitoring [66] and adaptation mechanisms. Such framework adaptation actions can be provided for minimising

management overheads, while not leaving performance problems undetected. Additionally, adaptation logic can be used to conduct learning procedures and optimise itself over time via auto-reconfiguration operations [22].

Several strategies exist for designing and implementing adaptation logic in automated management frameworks. A discussion of some of the most commonly used solutions follows. The mostly adopted approach for implementing adaptation logic is through *decision policies*, which provide a formal way of specifying management behaviour. More thorough definitions of policies are available [55, 98, 90, 15].

Various types of policies can be used for implementing adaptation logic, including declarative (or procedural) policies, goal-oriented policies, or utility-based policies. An important decision when building a management framework is selecting the strategy to use for designing and implementing the framework's adaptation logic. Available options differ in the type of policies used, the manner in which decision policies are to be processed, as well as in the actual mechanisms used to specify or implement policies. The decision regarding which policy type or type combination to use when building a management framework depends on the system management requirements, system scale and available resources for building the framework. Some of the main policy types and the possible strategies to implement them are presented next.

Declarative (Procedural) Policies

Currently, the most widely adopted policy type in the AC domain is the declarative or procedural policy. It specifies the actions to be executed when certain conditions are met. These policies are typically characterised by: i) an event that triggers the policy evaluation, ii) conditions to be evaluated and iii) actions to be taken when the policy conditions are met. For this reason, this type of policy is also sometimes referred to as event-condition-action (ECA) policy (or simply action policy [55]), or if-then policies. Several options are available for implementing adaptation logic based on if-then decision policies [52, 7].

Besides the adopted policy types, an important aspect to consider when building policy-based adaptation logic is the technology used to implement the actual policy sets and the mechanism for processing them. Various policy languages are available for this purpose. Ad-hoc solutions are also sometimes implemented, usually for building simple adaptation logic with few if-then policies. Possible approaches differ in their suitability for the problem addressed, in their implementation costs, and in the flexibility and manageability that they offer. Most available policy languages provide a *rule specification standard* and an *inference engine* to process policies that comply with that standard. The principal rule processing methods, or inference engines, include sequential processing, backward and forward chaining, or various fuzzy solutions (in case fuzzy variables are used). Some of the existing rule languages that provide these types of inference engines include the ABLE Rule Engine[16] (ARL), Jess [17], or Mandarax[18]. Certain rule languages (e.g. ARL) allow system administrators to specify management rules as a set of statements stored in special-purpose files, which are separated from the managed application code. This implies that managers do not need to understand the intricacies of the underlying application implementation in

[16]ABLE Rule Language (ARL): www.research.ibm.com/able

[17]Jess, the Rule Engine for the Java platform: herzberg.ca.sandia.gov/jess

[18]Mandarax, Java Rule Engine: sourceforge.net/projects/mandarax

order to specify sound management policies. Additionally, management policies can be modified independently from the application implementation logic and without requiring the application code to be recompiled. These features significantly increase the flexibility and manageability of the policy-based adaptation logic. In ad-hoc implementations, ECA rules with sequential processing can simply be specified by using the if-then structures supported by the programming language used for coding the management framework (e.g. Java, C++). This option may reduce the implementation costs of implementing simple, small-scale policy sets; it also provides reduced flexibility and manageability for the adaptation framework, as the policy code is mingled with the application business logic code.

Goal-Oriented Policies

Another approach for implementing adaptation logic is based on *goal-oriented policies* and *plans* [10, 52, 55]. Even though this approach was less implemented in the autonomic computing domain, it has recently started to receive increasing attention from the community. In goal-oriented approaches, high-level system *goals* are specified, rather than actual instructions for how to achieve those goals. Goals indicate desirable system states, or desirable characteristics that the targeted states must provide. A specialised *strategy* is generally used for identifying the possible means for attaining the specified goals [10]. Thus, strategies represent the connection between the actual system (or system description) and the specified system goals. In order to implement such strategies, the high-level goals are mapped to lower-level policies that the system can execute. A policy engine is used for this purpose, automatically mapping high-level goal policies to low-level executable policies [52]. As the inferred low-level policies must be executable policies, they must be supported by the underlying system mechanisms, or devices. When this is the case, running a selected low-level policy sequence triggers the execution of corresponding operations in the underlying system mechanisms. This in turn places the system into the desired state, thus attaining the high-level specified goals.

Policy engines for goal-oriented policies are also referred to as *planning* [52]. A plan consists of a sequence of actions to be taken for achieving a certain goal. A planning algorithm searches the spaces of all possible or available plans and extracts a single feasible solution. This is done by efficiently searching the space of possible system states, selecting the ones that are favourable for achieving the goal and determining the sequences of actions needed to reach those states. Plans can be specified as workflows, with sequential, conditional, parallel and loop constructs. They can be created automatically, or provided by clients or domain-specific methods. Over time, a repository of possible plans is created, providing the opportunity to select an existing plan or create a new one for reaching a certain goal. More specifically, various available plans, consisting of a set of actions, are being evaluated to see how their execution would help meet specified goals (e.g. performance goals). A plan is selected and executed based on its predicted capability of leading to an application state closest to the optimal goals. If none of the existing plans suffice, a new plan can be created, from existing sub-plans, rules, or possible actions. Some of the most relevant domain-independent technologies available for implementing planning-based adaptation logic include ABLE (the Agent Building and Learning Environment) from IBM [89], LPG[19] (Local search for Planning

[19]LPG Project: zeus.ing.unibs.it/lpg/

Graphs) and SAPA[20] and [36].
In comparison to procedural (if-then) policies, goal-oriented policies allow for more flexibility in the adaptation process, and they free policy designers from having to know the low-level details of system functionalities. Nonetheless, this comes at the cost of having to implement fairly complex planning and/or modelling algorithms [55].
Procedural policies and goal-oriented policies based on planning are not necessarily conflicting approaches. In [52], it is argued that planning-based approaches are the next evolutionary step from the procedural policy-based approaches today. Additionally, planning can be used to manage procedural policies. For example, a typical planning problem would be finding a combination of existing procedural policies that would achieve a certain high-level system goal. Reciprocally, procedural policies could be used to select information and operations used in planning processes.

Utility-Based Policies

As a further evolutionary step from goal-oriented management solutions, *utility functions* can be used to more accurately specify the desirability of various system states. In goal-oriented approaches, binary functions are used to determine whether a system state does or does not meet the specified system goals. Utility-based approaches are different in that the desirability of a state is a real value, rather than a Boolean value. This allows for the desirability of a system state to be more accurately evaluated and ranked based on multiple considerations, rather than merely classifying a state as desirable or non-desirable. Thus, the aim in utility-based approaches is to place the system into a feasible state that has the highest utility value. This has the benefit of rendering *utility policies* conflict-free. However, such policies require the extra cost of having to precisely specify numerical values over the entire system state space [55].

Trusting Decision Policies

An important matter to address when providing policy-based management solutions is the 'trust' that can realistically be placed on the efficiency and correctness of such automated solutions. The main concern raised is on how system administrators can trust automatic frameworks to manage their systems. A possible solution for evaluating automatic management frameworks and gaining the trust of system administrators is proposed in [22]. Conforming to this solution, automated frameworks are assigned different levels of trust, based on an evaluation of their runtime decisions. The evaluation is made by human system managers, who compare their own, correct decisions, with the management actions proposed by the automated framework. A positive evaluation is given when there is a good match between the two sets of management decisions. Based on this evaluation, an automated framework is placed in one of the existing trust categories: minimum trust, partial trust, or maximum trust. Minimum trust indicates that the automatic system cannot take any action unless first approved by a human administrator. At the other extreme, maximum trust means that an automated system can freely implement its decisions for the managed system, without needing confirmation from human administrators. The partial trust is an intermediate trust level where the automatic system may take some decisions on its own but needs the approval of a human administrator for others. The described evaluation process is repeated multiple times,

[20]SAPA Project: rakaposhi.eas.asu.edu/sapa

in several scenarios. Each time, the automated framework is tuned and updated so that its decisions better match the ones recommended by human managers. Following this process, the automatic framework can move over time from the minimum trust level initially assigned to it, towards the maximum trust level - the goal of automatic system management.
In regard to the actual decision behaviour implemented in the adaptation logic, two main approaches exist. In the first approach, all decision behaviours are specified and the best one is selected for each circumstance. In the second approach, statistical data and learning procedures are used to iteratively construct and adapt the decision behaviour in dynamic environments [51].

2.6.7 Obtaining Component Performance Information

Overview of Performance Information Retrieval

Automatic management solutions adapt applications and optimise their quality characteristics during runtime. The adaptation and optimisation decisions taken are based on the information available about the managed components, the current system state and the current execution environment. More sophisticated decision policies can also consider historical data on previously taken decisions and corresponding outcomes.
In performance management frameworks, component performance information is used to take adaptation decisions and optimise system performance. Therefore, an important matter when devising performance management solutions concerns the actual *source* of the information needed on the performance characteristics of managed components. Component performance information is required to support adaptation decisions for optimising application performance. Many research solutions require this information to be supplied by component providers and be made available at component deployment time [101]. Component-level performance information is stored whether in a performance description document, or accessible from the components themselves via special-purpose API functions [93]. Depending on the targeted component technology, several limitations may arise when performance information must be provided at deployment time and used as such throughout the system's lifetime. Specifically, two major concerns influence the applicability of this approach. The first is related to the granularity of the targeted managed entities. The other is related to the complexity of the underlying software platform on which managed entities are being deployed and run. More precisely, the finer-grained the managed components and the more complex the layers between components and the supporting hardware platform, the harder it is to apply this approach. Fine grained components and complex platforms make it increasingly difficult to obtain useful mappings between managed component instances and the hardware resources they need in order to properly function. Obtaining such information statically, or offline, for fine-grained components running in complex environments, would be in most cases an extremely costly and error-prone task.
Obtaining and using runtime performance information on managed components may prove a more viable solution for complex systems. Performance monitoring tools are available for most component-based middleware platforms. They can extract runtime information on the resource consumption at component level. Clearly, the overheads induced by runtime monitoring and diagnosis must be considered, such as with any management activity or non-functional service provided at system runtime. Component information obtained

during runtime can successfully be used to predict the performance of the component on the same platform, under similar execution conditions. Nonetheless, component information obtained in a certain execution context will be less relevant when the same component is integrated in different applications, and run on different platforms. For this reason, static performance testing does not provide performance guarantees for components that are to be integrated in different systems, deployed on different platforms and run under different workloads. Therefore, if static performance information, such as absolute response times or resource usage values, were supplied by component providers, this information would almost certainly prove inaccurate under most execution contexts. It might indeed be possible in theory for performance information to be provided with sufficient accuracy for a particular component and a certain, specific execution context; the targeted execution context would consist of a particular release version of a certain application server, JVM, Operating System and hardware platform. In this case, separate, extensive tests would have to be run to determine the way the component's performance changes on this particular platform, with changes in the incoming workloads and diverse server and JVM configurations. The process of obtaining such information for a single, very well specified platform would be extremely costly. Having to repeat the process each time one of the platform layers changes, or indeed a new version appears for one of these layers, might prove prohibitively expensive.

Certain approaches propose that components provide special-purpose API functions that return the component's performance characteristics upon request. Responses would be based on given input values describing the current execution environment [93]. More precisely, the performance functions return a component's response time, throughput and resource usage characteristics, based on input data on the current deployment platform and incoming workloads. This is a valid approach for certain system domains, such as for scientific applications where components implement data processing algorithms and do not necessitate any significant middleware support [93]. Applying these approaches to component-based enterprise applications would necessitate carrying out extensive testing and/or modelling processes so as to be able to determine correct component performance behaviour and formally represent it as API functions. In addition, the performance characteristics of those components that are being used by the targeted component would also have to be considered, since components used may have a critical impact on the overall transaction performance.

For the presented reasons, approaches based on static performance information may only be suited when managing coarse-grained components, running on relatively simple platforms, such as in the case of pervasive computing systems, Storage Area Networks (SAN), or enterprise systems managed at the server granularity level. For example, an application server distribution may provide recommendations on the approximate amounts of CPU, or memory that should be available to the server at runtime in order to ensure reasonable performance. The subject of obtaining component performance information in the context of the EJB component technology is discussed below.

EJB Component Performance Information

EJB components are fine-grained components, with a fairly complex underlying platform. The software platform consists of multiple layers with complicated behaviours and provided functionalities that include application server, JVM and Operating System. These platform layers provide services that are not directly related to the application business logic, such

as distribution, component instance creation and destruction, caching operations, or garbage collection. Thus, the execution behaviour of such services is not easily correlated with the business logic-related activities performed by running EJB instances. However, these services have a profound impact on performance and hardware resource usage [21]. In consequence, the exact mapping between the activity of EJB instances and the hardware resource usage is not straight forward to observe.

When commercial application servers are used, the server source code is not available. Thus, it is not possible in these cases to model the server's behaviour based on knowledge of the way the server is implemented. It could be argued that this problem can be solved by predicting the server's behaviour to a certain degree, based on extensive testing procedures. It may be indeed possible to obtain satisfactory results by running a sufficient number of separate tests and correctly analysing and merging collected monitoring data. However, this approach may prove prohibitively costly due to the high complexity of application server behaviour. Overall, the cost of the process needed for obtaining such results would probably not justify the benefits. In addition, the process would have to be repeated for various container configurations and for each different server version release.

When open-source EJB servers are used (e.g. JBoss, JOnAS[21]), their source code is available and could be analysed for understanding and predicting the server's behaviour. Nonetheless, the complexity of the application server logic, combined with the complexity of the underlying layers remains a critical factor within the scope of creating accurate models, rendering the process highly expensive and error-prone. Such models would also have to be configurable to consider different incoming workloads and underlying platform layers (e.g. JVM, OS, hardware). As before, this costly process has to be repeated for each different server release.

Alternatively, a viable approach could be to automate the described testing process, so as to automatically obtain performance information on a specific component, when running in a certain execution context [25]. In such cases, inferred performance information can accurately be used to predict EJB components' performance while they run under the same execution environment in which the information was obtained. The same information can also be used as general guidelines on the EJBs' performance behaviour (rather than as absolute values) when the same EJB components execute as part of different applications or on slightly different execution platforms. In these cases, such general, initial performance information would have to be validated and correspondingly updated at runtime with accurate monitoring information from the current system. Runtime performance diagnostic tools are available for assessing the performance of EJB instances. Commercial tools, as well as research instrumentation solutions (e.g., [66, 43, 73, 20] can be used at runtime to collect accurate monitoring data from the targeted system and execution environment. Data mining and machine learning strategies can then be used to process collected monitoring data and infer reliable performance information.

2.6.8 Adaptation Actions - Strategies and Operations

Several types of remedial actions can be used to adapt applications, depending on the type of managed entities and the targeted management goals. Some of the most frequently employed adaptation actions include:

[21]JOnAS J2EE application server: jonas.objectweb.org

- replacing application components with new component versions, in order to correct detected functional faults (e.g. [77])
- swapping components with functionally-equivalent components, for performance optimisation purposes (e.g., [101, 3, 93])
- redeploying components on different hardware nodes, in order to recover from node failure, optimise resource usage on a servers cluster, or for load-balancing purposes (e.g., [73], or [95])
- restarting components in order to deal with certain types of faults (e.g., [20]); in J2EE systems for example, restarting software components, such as EJBs, can be achieved by redeploying components (e.g., [73], or [20]).
- reconfiguring components, in order to optimise system performance. The set of reconfigurable parameters strictly depend on the type of targeted managed component (e.g. maximum size and instance lifetime for a caching component, or number of deployed applications for a clustered server component)

In J2EE systems, adaptation operations can be performed at different component levels, such as:

- At server level: software servers can be started, stopped, restarted, redeployed on a different hardware node, reconfigured, or updated with a newer version (e.g., [73], [19], or [43]).
- Middleware service level: server-provided services can be reconfigured, or replaced with different implementations (e.g. in JBoss, developers can provide custom service implementations to be used by JBoss during runtime) (e.g., [1])
- Application component level: EJB components can be redeployed, hot-swapped, or reconfigured (e.g., [20], [99], [28], [33])

2.7 Software Redundancy for System Management

Previous sections discussed relevant related work on automatic management frameworks for complex, component-based systems. Other research directions, significant for the presented dissertation, are based on the use of software redundancy for managing and meeting the quality goals of software systems. Specifically, various approaches use redundancy in order to achieve fault-tolerance, self-healing, or self-optimisation. Various system types are being targeted, including procedural, object-oriented, component-, or services-based software applications. This subchapter discusses the main research directions in which software redundancy is used for automatic management purposes.

2.7.1 Redundancy for Fault-Tolerance

Redundancy for increased robustness or reliability has been successfully used in various domains, including computer hardware, mechanics, or constructions. The same concept was later introduced in computer software, in order to achieve fault-tolerance in software systems [47]. 'Design diversity' is another term used in some research projects to denote the redundancy concept [76], [63].
Significant examples of system fault-tolerance schemes implementing the redundancy concept include the Recovery Blocks (RB) [78] and the N-version programming (NVP) [5], or [6]. Other, intermediate schemes exist, combining and/or optimising the RB and NVP approaches. These include the N self-checking programming (NSCP), t/(n-1)-Variant Programming, the certification trail scheme, the Self-Configuring Optimistic Programming (SCOP) [62], or (even) the exception handling approach. The Recovery Blocks (RB) and the N-version programming (NVP) techniques are described in the following sections. The main similarities and differences between these approaches and the proposed research are subsequently discussed.

Recovery Blocks
The Recovery Block (RB) was the first scheme to be developed for achieving fault tolerance in software applications. It was initiated by Brian Randell, at the University of Newcastle upon Tyne in 1970 [78]. Two main considerations led to the development of the RB scheme:

- Structuring software systems in order to control their complexity
- Enabling fault-tolerance capabilities in software systems

These considerations led to the RB design style, based on the concept of *idealized fault-tolerant components*. The main purpose of idealized fault-tolerance components is to prevent residual software faults from propagating and impacting the system environment. Software faults are generated by a component's code and can be propagated through neighbouring components. The Recovery Block (RB) approach uses idealized fault-tolerance components for increasing software system reliability. It is important to note at this point that in the RB context, the term 'component' is used to generally refer to software modules, or pieces of software code.
An idealized fault-tolerance component, or recovery block, consists of a number of code variants, or *alternates* and an acceptance test, or *adjudicator*. When a client request reaches an idealized fault-tolerance component it is first executed by the first variant, also referred to as the *primary alternate*. The outcome of the primary alternate execution is evaluated by the adjudicator, which runs an acceptance test. If the acceptance test is passed, the execution outcome of the primary alternative is considered successful. The recovery block is consequently exited and the request results returned. Otherwise, in case the acceptance test fails, the state of the system is restored and the second alternative is invoked to execute the same client request, with the same input data. The acceptance test is run again, and so on, sequentially, until one of the alternates passes the acceptance test, or all alternates are exhausted. In the latter case, an exception is signalled to the recovery block's environment. In addition, recovery blocks can be nested, meaning that each alternate variant can itself be an idealised fault-tolerant component. As such, an exception raised by an inner-block can be handled by a recovery alternate of the enclosing block. Also, it is possible for different

alternates to produce different results, as long as the results are acceptable, in the sense that they pass the acceptance test. This would allow a primary alternate to aim at providing the desired service and an alternate to only attempt to supply degraded service.
The Recovery Block (RB) scheme involves little processing overhead, and induces small delays, unless faults occur. Nonetheless, its success critically depends on the effectiveness of the employed fault-detection mechanism - the acceptance tests.

N-Version Programming
The concept of N-version programming (NVP) was first introduced in 1977 [6], as a method for achieving fault-tolerance in software systems. The main element in NVP is the N-version software (NVS) unit [6, 5]. An NVS unit is a fault-tolerant software unit encompassing two or more member versions. Member versions provide equivalent functionalities, but are implemented by different parties. All member versions in an NVS run in parallel, processing client requests. The separate results delivered by each NVS member version are collected and compared. Individual results constitute the input of a decision algorithm, also referred to as a voter, which establishes the consensus result. If the individual results are not identical, the voter assumes that the majority (should there be one) is correct. The process by which NVS member versions are produced is referred to as N-version programming (NVP). As specified in [6], the success of NVP as a fault-tolerance method critically depends on whether the residual software faults in each version of the program are distinguishable. This is because in order to increase software reliability the NVP method highly depends on the assumption that different member versions produced by independent parties will fail independently. This would mean that failures in the different versions occurred randomly and were unrelated. The probability of all versions failing for the same input would be consequently very small. More specifically, for N versions, failure probability would be proportional with the Nth power of the probability of failure in the independent versions. If this assumption were true, the reliability of the software system could be considerably higher than the reliability of the individual versions. Nonetheless, certain research initiatives question the validity of this assumption [59]. Raised concerns stem from the fact that when working on difficult problems programmers tend to make the same mistakes, even when working independently. This is explicable by the fact that some problem parts are inherently more difficult than others. Based on NVP evaluation experiments presented in [59] it was concluded that "the assumption of independence of errors that is fundamental to the analysis of N-version programming does not hold". This does not mean that NVP should never be used, but that the NVP reliability might not be as high as predicted under the assumption of independence. Hence, a major objective in NVP is to maximize the independence of the member versions involved, by employing for example the design diversity concept. This would minimize the probability of two or more NVP versions to produce erroneous results for the same decision action [5].

2.7.2 Redundancy for Performance Optimisation

The previous section discussed how redundancy can be used for providing fault-tolerance capabilities in software applications, in order to increase the reliability of such applications. This section describes how redundancy can also be used for improving the performance characteristics (of software applications). This idea is based on the fact that different

implementation strategies are optimal in different environmental conditions [101]. One of the oldest initiatives towards using redundancy for improving software performance is the Open Implementation approach [57]. In the following section, we shortly present this initiative as well as other, more recent approaches.

Open Implementation

The main motivation behind the open implementation approach [57] is that "it is impossible to hide all implementation issues behind a module interface. Some of these issues are crucial implementation-strategy decisions that will inevitably bias the performance of the resulting implementation. Module implementations must somehow be opened up to allow clients control over these issues as well." This indicates two conflicting software development principles:

1. The encapsulation principle, or black-box, traditionally used in software development for obtaining software qualities, such as reuse, or portability.
2. Internal module information, on the implementation strategy, needed for achieving performance.

Consequently, the best module implementation strategy, with respect to performance, cannot be determined unless the module developer knows the way the module is to be used. However, the black-box principle forces the developer to decide the implementation strategy early in the module development and then lock this decision in the black box. This decreases the probability of successful module reuse by different clients, with different performance requirements, in different usage scenarios. This problem is sometimes referred to as the Encapsulation Performance Problem (EPP).

The Open Implementation approach addresses this problem by enabling clients to select the implementation strategy of the modules they want to use. This approach aims at overcoming the problems of the black-box principle, while maintaining its advantages. The open implementation solution involves the creation of software modules with certain characteristics. First, for the same offered functionalities, a software module provides different implementation strategies, each one suitable for a different usage context and performance requirements. Second, a software module presents two different interfaces :

- a primary interface, for specifying module functionalities. Clients can use this interface to call functions, or methods on the software module
- a meta-interface, for specifying the available implementation strategies. Clients can use this interface to select the most suitable implementation strategy of the software module they want to use

By providing these characteristics, the open implementation initiative aims at allowing programmers to:

- use a module's default implementation strategy when possible
- be able to select a module's implementation strategy when necessary
- deal with a module's functionality and implementation strategy decisions in largely separate ways

Dynamic Selection of Component Implementations - Local Performance Optimisations

Adaptive components are proposed in [101] with the purpose of optimising application performance in conditions of workload variations. The problem addressed is similar to the one presented in the Open Implementation approach - the inability of clients to control a component's implementation strategy, so as to obtain optimal performance in a specific application context. The main difference is that in the Open Implementation clients are made responsible for selecting the implementation strategy they deem optimal for use. Clients can only select the implementation strategy once, when implementing the application. After this, the selected strategy will be used throughout the application's lifetime. In [101], the selection is performed at the server, or provider side, rather than by the client. In addition, the selected strategy is dynamically swapped during runtime, to accommodate changes in the running application context's workload variations.

As defined in [101], *adaptive components* have multiple implementations, each one optimised for a certain incoming workload. A mechanism for switching between the available component implementations is also provided at deployment time. The swapping mechanism needs to be implemented by component developers, separately for each swappable component pair. A management mechanism is to be implemented at the component platform, middleware, or running-environment level. This management mechanism is responsible for monitoring a component's incoming request workload and dynamically switching component implementations, using the swapping mechanism provided, so as to achieve optimal performance. An important issue is *when* to switch implementations and *which* implementations to switch in order to achieve optimal performance. In [101], this issue is referred to as the *adaptive component problem*. Therefore, the focus is to find an algorithm for calculating the potential benefits of swapping two component implementations. The Delta algorithm is proposed for this purpose, to determine *when* to perform a swapping operation, if two alternative implementations were available.

2.8 Self-Adaptive Software Systems

Self-adaptive software is a relatively novel approach [61] that promises to enhance robustness and performance characteristics of software systems, in conditions of changing resources or requirements. According to the Defence Advanced Research Projects Agency (DARPA)[22]:

> "Self-adaptive software evaluates its own behaviour and changes behaviour when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible. ... This implies that the software has multiple ways of accomplishing its purpose and has enough knowledge of its construction to make effective changes at runtime. Such software should include functionality for evaluating its behaviour and performance, as well as the ability to re-plan and reconfigure its operations in order to improve its operation. Self-adaptive software should also include a set of com-

[22]DARPA Broad Agency Announcement on Self-Adaptive Software, BAA-98-12, December 1997: www.darpa.mil/ito/Solicitations/PIP_9812.html

ponents for each major function, along with descriptions of the components, so that components of systems can be selected and scheduled at runtime in response to the evaluators. It also requires the ability to impedance match input/output of sequenced components, and the ability to generate some of this code from specifications. In addition, DARPA seeks this new basis of adaptation to be applied at runtime, as opposed to development/design time, or as a maintenance activity."

The aforementioned proposal indicates that self-adaptive software is able to evaluate its behaviour and performance at runtime. When evaluation results do not conform to the system goals, or in case optimisations are possible, the system is able to change its structure, configuration, or behaviour, in order to improve its functionality or quality characteristics. This implies that the system has knowledge of its objectives and intended behaviour, as well as means of monitoring and correspondingly modifying itself at runtime, for meeting these objectives.

As stated in [61], most self-adaptive software approaches use concepts from two main (research) domains: dynamic planning and control theory. The ideas promoted in these two fields are complementary. Consequently, most research initiatives on self-adaptive software employ notions from both areas, to various degrees. In dynamic planning systems, goal-oriented *operation plans* are used to dictate and schedule the actions to be taken by such systems. Plans can be inspected, evaluated and dynamically modified at runtime. Dynamic plan changes are reflected in the system runtime operation. Hardware, communication links and software modules are considered computational resources that a plan can (re-)configure and schedule (section 2.6.6).

Regarding control theory-based approaches, various research initiatives propose that concepts, architectures and techniques used in control theory should be mapped to software engineering, for specifying and designing self-controlling software systems [60, 69]. Control theory concepts that would be used in this way include controllability, observability, stability and robustness. It is considered that accumulated experience and solutions devised for solving certain problems in control systems, such as oscillating states, or chain reactions, can be used for solving similar problems in the software engineering domain. Emphasis is placed on adaptive control theory considerations. Adaptive control systems implement monitoring, evaluation and re-configuration mechanisms separately from the functional system that is managed and controlled.

Most self-adaptive software approaches separate the system adaptation logic and adaptation mechanism from the system functional code [23, 72, 60, 37]. In most self-adaptive approaches, adaptation mechanisms use centralised system models for representing knowledge on the system operation, structure and purpose. Such models are employed in system evaluation, re-configuration and adaptation processes.

System architecture is commonly used as a base for building system models (e.g. [23, 72, 37, 44, 43, 73]). That is because system architecture provides a useful system abstraction, hiding unnecessary details, while preserving the essential aspects. Architectural models can generally show:

- a system's structure, including the component interfaces and interconnections
- configuration information, indicating the component implementations for each interface and the intercommunication protocols used

- data and control patterns, specifying the system behaviour

An architectural model can also constrain the permitted reconfiguration operations on the system [23]. Several adaptable-software projects employ architecture-based models for representing knowledge on the system structure, behaviour and goals. These models are maintained during system execution and reflect changes in the underlying system. At runtime, the system architectural model is augmented with system monitoring information. Specialised models are being created depending on the targeted quality parameters. For example, specific performance and reliability models are created in case performance and reliability are the targeted quality parameters. Specialised models are derived from the architectural model and augmented using relevant monitoring information. For example, response times and throughput information is used for performance models, whereas the number of failures per time interval would be used for reliability models. Created system models are evaluated during runtime in order to identify functional or quality related problems. For solving detected problems, available repair or optimisation alternatives are considered. Re-factoring solutions are devised, based on the current model(s), evaluation results, prediction results for the new considered models and possibly a history of previous re-factorings and their corresponding outcomes. Analytical methods, such as queuing theory-based techniques, are used for predicting the values of the targeted quality parameters for a specific model. The selected adaptation operations are enforced, or reflected, into the running system, while preserving system integrity. Possible system adaptation operations consist of model re-factorings such as repair, or optimisation procedures. System integrity is preserved by correctly transferring state information between component versions and keeping client references consistent. Some adaptability approaches aim at repairing software systems in case the system functional or quality-related objectives are not being met [23]. In addition to this, other approaches also attempt to optimise software systems whenever possible [72]. In such cases, a system is being optimised whenever an improvement opportunity is detected, rather than only when system goals being infringed.
In reflective or self-adaptive systems, changes performed on the abstract system model are automatically reflected in the operational system. Similarly, modifications in the operational system, including component updates, or architectural changes, are reflected in the (abstract) system model. Models can be augmented with system quality information, such as performance or reliability related information. Runtime system monitoring is generally used for this purpose to collect the system runtime data. Evaluation operations can then be performed on the resulting performance or reliability models, in order to determine whether the system is meeting its performance or dependability goals. Reconfiguration operations are correspondingly devised or selected for improving the system when necessary.

2.9 Hot-Swapping in Software Systems

Component hot-swapping refers to the ability of replacing components at runtime, even if they are being actively used by the system [50]. The ability to hot-swap components can be used as part of automatic system adaptation solutions to increase system availability, optimise performance and improve reliability. Component hot-swapping supports dynamic system upgrades, allowing for a system's behaviour to be modified without interrupting the system's execution. As suggested in [42], component hot-swapping can be realized by:

- *Interposition* of code: inserting a new component between two existing components, at runtime
- *Replacement* of code: switching an active component with a different one, at runtime

According to one of the research initiatives at IBM [42] hot-swapping can be employed for enabling autonomic features in software systems, as follows:

> "Autonomic computing systems are designed to be self-diagnosing and self-healing, such that they detect performance and correctness problems, identify their causes, and react accordingly. These abilities can improve performance, availability, and security, while simultaneously reducing the effort and skills required of system administrators. One way that systems can support these abilities is by allowing monitoring code, diagnostic code, and function implementations to be dynamically inserted and removed in live systems. This "hot swapping" avoids the requisite prescience and additional complexity inherent in creating systems that have all possible configurations built in ahead of time."

As the best strategy for obtaining optimal performance critically depends on system workload and available resources, components with different implementation strategies can be dynamically replaced when the system execution conditions change. A trade-off exists between the benefits obtained by employing extensive system monitoring and data analysis, for detecting and solving performance or security problems, and the performance overheads caused by such monitoring and analysis operations. Interposition can be used to enable adaptive monitoring techniques [66], which allow extensive monitoring to be activated when general problems are detected and removed after no longer needed. Additional hot-swapping benefits include: imposing system modularity, increase system availability by enabling dynamic upgrades, support system evolution and simplify testing processes.
The main actions involved in performing component-swapping operations include [42, 50]:

- Triggering the hot-swapping operation - either the hot-swapped component instance itself or the supporting system infrastructure can determine when a certain component needs to be replaced. Monitoring and data analysis operations are required for determining when to hot-swap components. Extra monitoring can be enabled if needed, by means of (object) interposition.
- Choosing the target component - which component to hot-swap
- Performing the hot-swap operation. Various aspects need to be addressed when performing this action. One concern is related to state transfer between the hot-swapped component instances. First, a component state must be established when it is safe to transfer the state and hot-swap components. A component instance's state will not be accessible by any thread in the system when it is being transferred. Second, the component instance state must be transferred from the old component instance to the new one. Another hot-swapping concern involves the corresponding modification of all client references from the old component (instance) to the new one.
- Dynamically introducing new components (types)

2.10 Dynamic Component Versioning

Component versioning aims at replacing current system components with new component versions, in order to provide new functionalities, remove bugs, or improve quality characteristics. An important concern is verifying whether new component versions are indeed better than the old ones. This verification should be performed before dynamically upgrading the system. A second concern is the actual system update procedure, which needs to be performed during runtime, while preserving the system's integrity. The component versioning approach presented in [77] focuses on increasing the confidence in new component versions, before allowing them to operate in the system. The approach is based on testing new component versions online, in parallel with the old component versions that operate in the system. Online test results, from new and old component versions are compared and the best component version determined. For the online testing procedure, new component version candidates are deployed and run in parallel with the old component versions. Candidate versions receive and handle client requests from the running system, the same way the old versions do. However, candidate versions under test are not allowed to influence the running system in any way. Only results yielded by the old versions are propagated into the system. New component versions are allowed to replace old versions only if they are evaluated during the testing period to be correct and superior in some sense to the old versions. In its current specification though, this online testing method does not handle situations in which tested components need to use and get results from other components in order to execute their tasks and provide results. This is a serious constraint, since it limits the applicability of the proposed method to leaf components that do not affect the system in anyway. The possibility of multiple component versions being kept is also considered. However, the circumstances in which this would occur, or the way such versions would be managed and used in parallel is no further elaborated. The presented dissertation research considers cases in which it is hard, or even impossible, to devise and manage a single monolithic component that yields optimal performance in all possible execution contexts (e.g. workloads, or available resources). Such cases are addressed by using different component variants yielding optimal performance characteristics in different execution contexts. For this reason, the aim is to maintain a number of such component variants at runtime, and be able to use them alternatively so that they 'complement' each other and provide optimal quality characteristics, such as performance, reliability, or correctness, at all times.

CHAPTER

THREE

Using Component Redundancy for Automatic Performance Optimisation

Chapter Summary

This chapter describes the redundancy-based optimisation solution proposed as part of the thesis research. It clearly defines the thesis goals, delimits its scope and shows its main contributions. The presentation introduces the component-redundancy concept and explains the way it is used to dynamically optimise software applications. Related concepts and terms required to describe the presented management solution are defined. The proposed AQuA management framework is presented, as a means of supporting the redundancy-based optimisation solution. Namely, this chapter illustrates how the AQuA framework automatically manages redundant components so as to continually adapt and optimise applications, during runtime. The description explains AQuA's main functionalities and shows the way they work together in order to reach the system's management goals. AQuA's main functions include system monitoring, learning, anomaly detection, component evaluation, adaptation decision and component activation. The approaches adopted to design the management framework and its main functional modules are discussed. AQuA was specifically designed for managing enterprise systems built using component technologies. In particular, component technologies based on contextual composition frameworks were targeted, such as Enterprise JavaBeans or CORBA Component Model.

Goals of this chapter:

- redundant components can be used to dynamically optimise system performance, as well as other QoS attributes
- separating redundant components from each other and from the adaptation logic managing them ensures system modularity and improves flexibility
- using an automatic management framework to adapt and optimise applications without requiring human intervention increases system management efficiency and reduces administrative costs
- the current expertise of human system administrators can be captured and used to automate performance management tasks
- data mining and machine learning techniques can be used to automatically infer performance information on managed system components and to augment and improve the system's management adaptation logic over time

3.1 Research Goals, Proposed Solution and Scope

The presented research aims to enable complex software systems to manage themselves, so as to dynamically self-optimise and adapt to changes in their internal configurations and external execution conditions. The thesis goal is to propose a solution for reaching this aim and exemplify how the solution can be implemented and used. The thesis also seeks to identify and discuss the main challenges that must be addressed in order to provide a complete, fully reliable management solution that can be employed in a real system scenario.

The proposed management solution is based on the alternate usage of multiple component variants with equivalent functional characteristics, each one optimised for a different running environment. Namely, in the proposed approach, different component implementations are provided at runtime to supply equivalent functionalities based on different design and configuration strategies. Each component implementation strategy is optimised with respect to a different execution context. Part of the same approach, applications are enabled to automatically analyse and select the optimal component strategies to use at anyone time, in each particular execution context. A solution for meeting this goal is proposed based on the following requirements. First, different design and implementation variants for software components must be available at runtime. If multiple component variants are not available, the system functions normally, without being automatically optimised based on the proposed management solution. As a second requirement, the solution must provide a mechanism that is capable of automatically alternating the usage of the available implementation variants. This activity must be performed during runtime, so as to meet the software applications' high-level quality goals, at all times. The *component redundancy* concept is introduced for addressing the former solution requirement (section 3.3). The second solution requirement is addressed by proposing an automated management framework, referred to as AQuA (Automatic Quality Assurance) (section 3.9). AQuA's role is to administer the available component variants, so as to capitalise on their redundancy and continuously optimise applications, while constantly meeting the system's quality goals (e.g. performance goals: response times and throughputs). A prototype implementation of the proposed management framework is also provided, for the J2EE component technology. The AQuA_J2EE prototype shows how the proposed solution can be implemented and used to fully-automate performance management tasks.

The thesis scope includes proposing the redundancy-based management solution and devising the AQuA framework for automating system management based on this solution. It also comprises the AQuA_J2EE framework prototype, showing how the solution can be implemented and used. Nonetheless, a complete, optimal implementation of the proposed management framework is out of the thesis scope. The AQuA_J2EE prototype presented as part of the thesis work is intended to show how the framework functionalities can be implemented and integrated together. Further research, development and optimisation work is required on each of the framework's management functions in order to produce a fully-functional, reliable product. The runtime management of complex software systems involves highly complicated procedures, with a myriad of interconnected aspects to consider. The thesis does not attempt to solve all problems implied by a system's dynamic adaptation and performance optimisation. Rather, the aim of this research is to set a direction for a feasible automatic management approach, which can be subsequently extended and integrated with other approaches so as

to provide a complete system management solution. Extensive research efforts are currently being carried out in related areas of the autonomic computing field, including system monitoring, policy-based management, data mining, machine learning, application adaptation and evolution. The proposed management approach and framework provides an integration point for bringing together the results obtained from these various domains and obtaining a complete autonomic management solution. While the focus of the presented research is on performance optimisation, many of the proposed concepts and functionalities can be applied for managing other QoS attributes, such as availability and reliability.

3.2 Uniqueness of the Approach

This section discusses some of the key features of the proposed performance optimisation solution. These features differentiate the proposed redundancy-based approach from related management frameworks in the area.

First, an important characteristic of the provided optimisation scheme is that it maintains a clear separation between the application's business logic and the performance management code. This feature increases system flexibility and manageability (subsection 2.6.2). Another important characteristic is that the presented management solution imposes no specific requirements on the component technologies used to implement the managed applications. Consequently, no major conceptual-level changes are needed when porting the AQuA framework across component technologies or component technology providers. Though, each particular AQuA implementation will clearly differ from other implementations, depending on the targeted component technology and adopted design strategy. However, no extra-requirements will be imposed on component developers or providers, as the AQuA framework design allows it to manage any standard component. Namely, no detailed information on the performance characteristics and/or resource requirements of the managed components is necessary at deployment time. Also, it is not compulsory for multiple component variants to be provided and available during runtime. Component variants can be acquired from multiple providers and added, modified or deleted during system execution.

The framework's modular design allows for each of its constituent parts, namely the monitoring, adaptation logic and component activation, to be independently modified, without affecting the other functions. Thus, AQuA's modularity allows for each of its functional modules to be built separately and/or subsequently replaced without impacting on the other modules. As such, AQuA's adaptation logic can be designed based on whether a centralised, decentralised, or hierarchical topologies (subsection 2.6.4). With respect to AQuA's monitoring function, two main instrumentation solutions can be adopted. These are application-level instrumentation, based on component proxies, and server-level instrumentation, based on container interceptors. For the two instrumentation solutions, the trade-off is between portability across application servers and the effortless management of any new application on a certain server (subsection 2.6.5). The AQuA_J2EE prototype implemented as part of this thesis adopted the server-level instrumentation approach. Consequently, no extra framework implementation or installation efforts are required for managing new applications. When server-level instrumentation is used, the framework is implemented once for a certain application server and then used without any further modifications for managing all applications subsequently de-

ployed and run on that server. Nonetheless, the framework implementation can be configured to meet the particular management requirements of each deployed application. Finally, various approaches are possible for implementing AQuA's component activation function, for performing component-swapping operations on running applications. The current solution adopted for AQuA's implementation was based on the hot-deployment facilities provided by the application server platform.
Finally, a significant characteristic of the proposed management solution is that external clients remain completely unaware of the management operations performed on the system at runtime.

3.3 Component Redundancy - Concepts and Terminology

The *component redundancy* concept is central to the thesis solution. Thus, a clear definition of this concept is necessary to state the exact meaning and implications of this term, for the scope of the presented dissertation.
Component redundancy is defined as the runtime availability of multiple component variants, providing equivalent functionalities, but each one optimised for a different execution environment. The design and implementation strategy of each component variant is conceived so as to be optimal under certain execution conditions. A software component's execution conditions include the incoming workloads, inter-component communication patterns, or available software and hardware resources for that component. Software resources may consist of other components and applications, such as relational databases, or of underlying threads and processes. Hardware resources include the available CPU, memory, disk, or network bandwidth.
The component variants in the proposed solution are referred to as *redundant components*. All redundant components that provide certain functionality are considered to be part of the same *Redundancy Group (RG)* with respect to that functionality [30], [29], or [31]. Each RG exposes a well-specified set of externally visible functions. This is referred to as the *RG interface* (Figure 3.1). The RG interface defines the set of RG functions that external clients can call on the RG. Consequently, all redundant components in a certain RG must implement the RG's provided interface. Thus, any redundant component in a certain RG can be functionally replaced by any other redundant component in the same RG.
In a more complex scenario, more than a single component can be used to implement the different functionalities of a RG's interface (Figure 3.2). In such a case, each component would implement one or more of the RG's provided methods, so that any of the declared methods in the RG interface is implemented by at least one component. A RG dispatcher is needed in this case to correctly distribute incoming client calls to the respective component that is capable of handling them. Even though feasible, this scenario was no further considered when developing the presented research solution.
Each redundant component in a RG should be optimised for a different running context. For example, one redundant component could provide optimal performance under increased incoming workloads, while a different redundant component can be optimised for lower incoming loads. Performance optimisation is considered with respect to performance metrics

such as response times or throughputs, but also as gains in resources consumption. It is also possible for different redundant components in a RG to provide certain functionality at different degrees or levels. As such, a RG can provide to its clients full-functionality or degraded functionality, in order to cope with unfavourable execution conditions, such as resource congestion, while still meeting its quality requirements. For this reason, redundant components in a RG are defined to provide *identical* or *equivalent* functionalities.

A single redundant component providing certain functionality is assigned, at anyone time, for handling a client request for that functionality. More precisely, the client request is forwarded to an instance of the selected redundant component upon arrival. This aspect differentiates the thesis' solution from other approaches, such as N-version programming [5], [6], or agent-based systems [47], where multiple component variants work together, or in parallel, towards a common result.

In the presented solution, the selected redundant component that handles client requests at a certain time is referred to as the *active* redundant component. The active redundant component is the one the system currently uses, sending client requests to instances of that component. Similarly, redundant components available in a RG but not currently used for handling client requests are referred to as *inactive* redundant components. Redundant components can be dynamically added, updated, or removed from a RG without disrupting the normal system functioning.

If instances of the active redundant component perform poorly in a certain execution context, the redundant component can be *deactivated* and replaced with an alternative member of the same RG. This process is performed via emphcomponent-swapping, which is performed at runtime without disrupting system execution. The component-swapping procedure can also be applied in case the active component is detected to throw exceptions or introduce integration faults, such as deadlocks. In the presented solution, the application adaptation process is based on anomaly detection and performance optimisation via component-swapping procedures. Redundancy Groups (RGs) use this process to continually optimise themselves, adapting to changes in their execution environments and dealing with context-driven faults.

It could be argued that an alternative approach would be to specify all behaviours for all possible execution conditions in a single, monolithic component, together with the logic for selecting which behaviour to use at each time. However, using separate redundant components for different running conditions provides radically improved modularity and flexibility over the aforementioned approach. The reason is that the separate redundant components are clearly isolated from each other, and from the adaptation logic that decides which one of them to use at each point. Thus, redundant components providing system functionality can be separately added, updated, or removed, without affecting the adaptation logic. Similarly, adaptation logic policies can be independently added, modified, or removed from the running system, as needed. In short, the use of component redundancy facilitates the separation between the application business logic and the system management logic. On the contrary, building monolithic components optimised for all possible running contexts would be unfeasible in most cases. Most importantly, it would be unlikely for system developers and deployers to envisage all possible execution conditions under which the system will run. Secondly, even in the unlikely scenario in which all possible running contexts could be known, a component optimised for all contexts would be excessively hard and costly to design, implement and maintain.

Regarding redundant components' granularity (subsection 2.6.3), an important property of

the proposed solution is that a Redundancy Group (RG) can contain atomic components, as well as composite components or sets of components. For example, Figure 3.1 illustrates a RG containing three redundant component of different types:

- a: A is an atomic component
- b: B is a composite component, containing sub-components B' and B''
- c: C1, C2 and C3 form a set of components. Component C1 advertises (some of) the same functionalities as components A and B do. However, in order for component C1 to provide its functionalities, as defined in its advertised interface, it requires functionalities from C2, which in turn requires functionalities from C3.

As shown in Figure 3.1, redundant components A, B and C all implement the same RG interface.

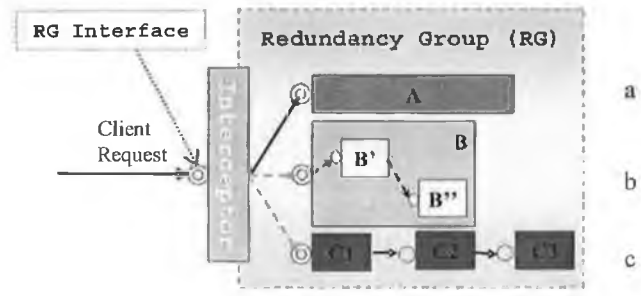

Figure 3.1: granularity of redundant components:
a) atomic component; b) composite component; c) set of components

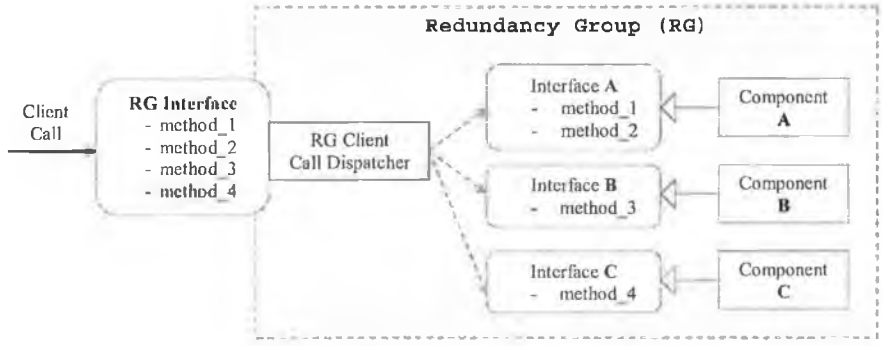

Figure 3.2: implementing the RG Interface with Multiple Components

Figure 3.3 shows how RGs can be used as components in a functional application. One redundant component will be active in each RG for handling client requests. Changing the active redundant component in a certain RG may require the use of RGs that were not previously employed. In the Figure 3.3 example, changing the active component in RG1 from C1.1 to C1.2 would cause the RG4 and RG5 to be utilised instead of the previous RG2 and RG3.

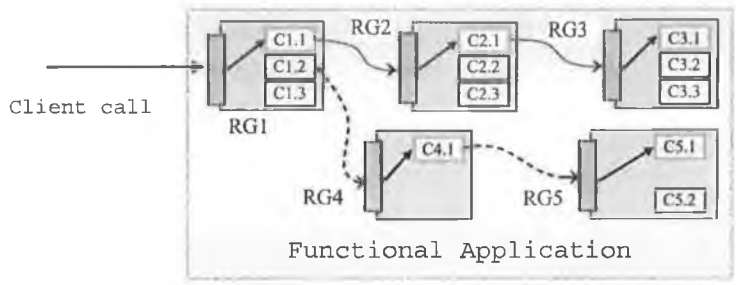

Figure 3.3: using Redundancy Groups to implement a functional application

3.4 Using Component Redundancy to Optimise Performance

The thesis proposes the use of component redundancy to automate the performance optimisation of component-based applications. Component redundancy contributes to the developed management solution by supporting runtime modifications in the applications' code and configurations. Namely, the main idea behind the presented approach is to have multiple redundant components prepared at runtime and only use the one that is optimal under the currently executing environment. More precisely, a number of redundant components are provided and made available during system runtime. Each component is optimised for a different range of environmental conditions, such as incoming workloads or available resources. At any one time, for providing certain functionality the adaptable system selects and uses a single redundant component from the RG providing that functionality. The selected redundant component is the one that is most likely to yield optimum performance under the targeted execution environment. If the execution environment changes another redundant component is selected from the same RG, so as to optimise the application's performance under the new execution environment. This allows the software system to dynamically adapt to variations in its running environment and maintain its performance at optimal levels at all times. In certain cases, knowledgeably alternating the use of redundant components optimised for different environments can yield better application performance than any one of the individual redundant components could provide (chapter 5). The alternative of merging all redundant components and their management control logic into a single monolithic component, optimised for all possible environments, would prove in most cases an unfeasible solution (section 3.3).
At a general level, the goal is to simultaneously alternate the use of redundant components in different RGs so as to obtain optimal application implementations and configurations overall, and constantly meet system performance objectives. In this context, the availability of accurate knowledge on the redundant components' performance characteristics becomes critical for a successful management process. The thesis introduces a learning mechanism for automatically determining the optimal software configurations of each RG and each managed application, for the targeted system and execution environment. The learning process is based on analysing monitoring data collected for each available component and each specific software application. An important characteristic of the proposed solution is that the learning process executes while the application is run in the targeted execution environments. This is differ-

ent from statically performing such information collecting processes on a testing platform. The proposed solution allows for information on optimal system configurations in different contexts to be optionally provided at application deployment time. Such information could be initially obtained by testing the application offline, on the targeted execution platform. It can also be acquired from previous experiences with the managed system, when the system was executing in similar contexts. Such initial information can be used as a starting point for adapting applications immediately after their deployment. In time, the learning mechanism progressively validates and updates the initial information, based on accurate monitoring data from the targeted execution context. Thus, the reliability of the performance information used to take system optimisation decisions increases constantly over time. As soon as performance information is available, the managed software applications can be dynamically adapted to varying QoS requirements or execution conditions. This is achieved by accordingly changing the applications' optimal configurations, considering the targeted QoS requirements and current running conditions.

In the proposed optimisation solution redundant components are used to dynamically adapt software applications at two granularity levels. First, component redundancy is used to optimise individual RGs. At this level, the optimal strategy is detected and activated for providing a RG's functionalities. This is equivalent with local optimisations of individual components. At a higher level, component redundancy is used to optimise component assemblies or applications. At this level, the optimal configuration of a component assembly, consisting of multiple adaptable components, is determined and activated. In this context, an optimal assembly *configuration* refers to the optimal combination of active redundant components in the RGs involved. As such, global system optimisation is achieved when the entire software application is optimally configured.

In the targeted component technologies (e.g. Enterprise JavaBeans (EJB) - subsection 2.4.2, [91]), components are typically deployed as a bundle of component implementation and configuration files. Thus, redundant components can consequently differ at the component implementation and/or the configuration levels. A component's implementation represents the business logic the component provides. Redundant components with differences at this level can be obtained from different component providers. Component configurations, or deployment descriptors, are used to instruct the application server on how to manage components at runtime. Variations at this level are specified by component deployers. Examples of redundant components with variations at both the implementation and configuration levels are presented in chapter 5 of the thesis.

In the context of the proposed redundancy-based solution, an important concept is that of *cross-points*. Cross-points are defined by the performance characteristics of pairs of redundant components and indicate the points where redundant components in a RG should be swapped. More thoroughly, the 'cross-point' term is defined for the scope of this thesis as follows. A *cross point* is an execution context in which the optimal redundant component in a RG changes. Execution contexts are defined by the values of all performance metrics considered. Cross points can be considered at different levels. At the finest grained level, a *basic* cross point is considered between two redundant components only, with respect to a single performance metric (e.g. response time). In this case, a cross point is the metric value at which the optimal redundant component changes.

More general cross points can be subsequently calculated by comparing and analysing basic cross points. Thus, when considering more than two redundant components in a RG, with

respect to a single metric, a cross point is defined as the metric value at which the optimal redundant component changes in that RG. Furthermore, in case more than one performance metrics are considered, overall cross points are computed based on single metric cross points. At the most general level, cross points delimit the execution contexts in which different redundant components are globally optimal, with respect to a certain RG. In case different redundant components are optimal with respect to different performance metrics, one final optimal component is selected, based on various possible criteria. System administrators are responsible for specifying the system's high-level performance goals. Thus, they also define the criteria for determining optimal redundant components. Namely, the optimal redundant components are those that best meet the system's high-level performance goals, at each particular time. A selection criterion can be specified as sets of rules, indicating the conditions in which one redundant component is considered optimal over other redundant components in a RG. For example, a rule can state that a certain metric has precedence over the other performance metrics. For example, suppose a rule is defined for a RG, stating that response time has precedence over CPU availability. Also suppose that the RG contains two redundant components, A and B. In this example, if component A is optimal with respect to response time and component B is optimal with respect to CPU consumption, then component A will be selected as globally optimal and possibly activated. More complex rules can be defined to calculate a global optimality factor for each redundant component, based on multiple performance metrics. For example, a weighted sum of performance values, one for each considered metric, can be used to compute the optimality factor for each redundant component. Based on this, for each execution context, the redundant component with the maximum optimality factor is selected as the optimal one in that context. Global cross points are those at which the globally optimal redundant component in a RG changes, as a result of an alteration in one or more of the execution context parameters.

3.5 Applicability of the Redundancy-Based Performance Optimisation Solution

The presented thesis proposes using component redundancy for automating performance optimisation processes in component-based, enterprise systems. Part of this work, several scenarios were identified in which component redundancy can be applied for dynamically optimising system performance. These scenarios were divided into three main categories, based on the nature of the runtime variations that require the system to be adapted. Namely, variations can occur whether in the components' execution contexts (subsection 3.5.1), QoS requirements (subsection 3.5.2), or functional requirements (subsection 3.5.3). These scenarios are discussed in more detail over the following subsections. In addition, a fourth possible case can be considered, in order to deal with initial performance optimisation of complex applications. This scenario is likely to occur when optimising the performance of large-scale applications, for the initial environmental conditions in which the applications are run. The main reason behind this scenario is the considerable complexity of large-scale, distributed applications to manage. Namely, such applications are typically built using a considerable number of interconnected components. In principle, any large-scale application can be initially

optimised, when its constituent components are being integrated or when the application is being deployed. Nonetheless, system complexity might cause application optimisations to prove to be a costly and possibly unreliable task to perform manually or offline. The situation is exacerbated if COTS (Component-Off-The-Shelf) components are being used as part of the application, as the implementation code may not be known in this case. Thus, the fourth adaptation scenario would be concerned with automatically finding initial application designs, implementations and configurations that are optimal under the initial running context. When using component redundancy, this process involves determining the optimal combination of redundant components, one from each available RG, for yielding optimal overall application performance in the initial deployment context. Even though valid, this fourth scenario is akin to the one addressing adaptations to changes in the components' execution contexts. In other words, optimising the system to an initial execution context is similar to adapting the system to dynamic variations in its execution environment. Therefore, the fourth adaptation scenario is not separately discussed.

3.5.1 Using Component Redundancy to Adapt to Varying Execution Contexts

The *execution context* of a certain component refers to the environmental conditions in which the component is deployed and runs. A component's environmental conditions are characterised by the incoming workload on the considered component and by the software and hardware resources available to the component.

The incoming workload on a component can be further characterised by several aspects, such as the request load and request mix. The request load represents the number of incoming requests per time interval. The request mix, or usage pattern, represents the type of incoming requests and the order in which they arrive. The incoming workload on a certain component is generated by the component's clients. Clients can be external users, other software applications, or other components of the same application. Thus, for example, an EJB component's client can be a servlet in a web application, or a different EJB in the same application.

Also part of a component's execution environment, are the software and hardware resources available to the component for providing its functionalities. Hardware resources include CPU, memory, network bandwidth and disk. Software resources can be part of the considered software application (internal resources), or they can be external to the application (external resources). Internal software resources for an application component are typically other components in the same application. This situation can occur as a component can use other application components to complete its tasks. External software resources can consist of other applications, databases or file systems. Software resources a component uses can also be considered at different, lower system layers. Namely, a software component at the application level may be mapped to resources at lower system layers, such as threads and processes in the underlying Virtual Machine (VM) and Operating System (OS). These in turn can be mapped to CPU, disk and memory usage at the hardware platform level.

Dynamic variations can occur in a component's execution context, at any of the above software and hardware levels. In effect, the performance characteristics of a software application may substantially change with the variations that occur in its execution environment. Thus,

a software application designed and configured for optimal performance in certain execution environment might no longer be optimal when running in different execution environments. As a consequence, dynamic changes in the initial environmental conditions of a running application might result in sub-optimal performance behaviour of that application. Different aspects of a component's execution context can change over the component's lifetime. Such changes can arise as the component is integrated in different applications and run on different platforms. In addition, environmental variations can dynamically occur while the component is executing as part of the same, long-running system.

The thesis focuses on automatically adapting applications to changes that occur dynamically, at system runtime. Possible dynamic changes in a component's execution context include modifications in the component's incoming workloads, or available software and hardware resources. The number of client requests received per time interval may fluctuate over time, causing corresponding variations in the amounts of required software and hardware resources. Furthermore, modifications can arise in the functionality of a component's clients. This would in turn have a direct impact on the component's usage pattern, or work mix. Such situation can occur as a result of updates or versioning operations performed on a component's clients. Changes in the implementation of a component's client can subsequently impact on the component's incoming load or usage patterns. Workload variations can directly influence the amounts of available software and hardware resources a component can use. Additionally, dynamic fluctuations in the available resources can occur independently from changes in the incoming workloads. This can happen as a result of variations in the resource usage of other software entities sharing the same underlying platform. For example, new applications can be deployed and run, or existing applications can be updated in a manner that considerably impacts their resource usage patterns. In another scenario, resource availability can change as a result of hardware failure or of resources being physically added to the managed system.

The component redundancy concept can be used to build dynamic adaptation solutions that address the optimisation problem in the context of varying environmental conditions. Such management solutions are based on alternatively activating redundant components individually optimised for different execution contexts. An example scenario was implemented for the EJB technology to test component redundancy applicability in the case of varying amounts of available resources (i.e. network bandwidth). The example implementation and test results are presented and discussed in section 5.1. Further examples are available from related work in the area of dynamic component replacement for performance optimisation purposes (e.g. [93], [3], [64], [101], [8], and [83]).

3.5.2 Using Component Redundancy to Adapt to Varying QoS Requirements

Component redundancy can be used to accommodate changes in the QoS requirements of an application's components, during runtime. A component's Quality of Service (QoS) attributes include performance, reliability, robustness, availability and security. As discussed in subsection 2.2.3, optimising one system QoS attribute may deteriorate another QoS attribute. For example, performance optimisations can be achieved at the cost of application security or

reliability. In addition to functional requirements, an application commonly needs to meet certain QoS requirements for its external clients. Application requirements can be formally represented as application or component contracts [91], that provide functional and QoS guarantees to external clients. Even in cases where system requirements are not formally represented and contracts are not explicitly signed with clients, certain requirements still have to be met for a feasible system.
Global, application-level requirements for an externally-provided functionality map to component-level requirements for the individual components implementing that functionality. Consequently, if an application's global QoS requirements change, the local QoS requirements on individual components accordingly change. QoS requirements can be modified for example so as to favour security over performance or performance over reliability. Performance can also be optimised with respect to different performance attributes, such as response times, or throughput and the precedence of these attributes can vary over time.
Component redundancy can be applied for dynamically adapting applications to changes in their required QoS levels. This is achieved by acquiring redundant components individually optimised with respect to different QoS attributes, or QoS goals. At runtime, the redundant components are alternately activated, so as to always match the system's QoS requirements. In other words, the redundant components used by an application change with the system's QoS requirements, during runtime.

3.5.3 Using Component Redundancy to Adapt to Varying Functional Requirements

Component redundancy can be used to accommodate changes in the functional requirements of an application's components, during runtime. Such functional changes do not necessarily refer to cases in which new functionalities are being added, or when deprecated functionalities are being removed from components. Rather, component redundancy can be used to address cases in which certain functionality is only required for limited time intervals. For example, logging or debugging functionalities, implemented as part of a component's code, can be temporarily activated to detect faulty components. Component redundancy can be applied in this scenario by using two redundant components. Namely, one redundant component only provides the 'normal' functionality, whereas the alternative redundant component also implements logging and/or debugging functions. The debugging-enabled redundant component contains debugging-specific code inserted in between the 'normal' code. In this scenario, the debugging-enabled redundant components can be used for example to detect component-level functional faults. More precisely, debugging information can be used to determine the cause of a detected fault. If a fault is detected at the application-level, the debugging process will help narrow down and pinpoint the fault at the component-level. Once a faulty component is identified and subsequently replaced with a correct version, the normal (debug-disabled) redundant component can be reactivated. The component-swapping process that supports component redundancy in a runtime management system (section 3.12) can be used in such cases to update faulty components with their correct versions. The component versioning topic and related challenges are out of the scope of the thesis. Nonetheless, the impact on a system's performance that component versioning might have can be automatically con-

trolled by using the proposed component redundancy based solution. For example, suppose that one application component is replaced with a newer version during runtime. This might require the application to be re-optimised in order to accommodate the new component. The reason is that the resulting integrated application may no longer perform optimally as a result of having to use the new component version. An overall optimisation with respect to the new component version being used would be needed in this case for remedying the situation.
System monitoring provides another example in which functionality may need to be dynamically added to an application for a limited period. Specifically, monitoring capabilities may sometimes need to be introduced at the application component-level, in order to identify performance bottlenecks. For example, the COMPAS monitoring tool uses component-level monitoring probes for extracting performance information from running applications. Monitoring probes are automatically inserted in the targeted EJB classes at application deployment time. At runtime, COMPAS is able to switch the monitoring probes between 'on' and 'off' states, in order to minimise induced monitoring overheads. Based on this procedure, COMPAS can automatically adapt its application's instrumentation to changing monitoring requirements. As such, when no performance anomalies are detected at the application level, only entry-point components are being actively monitored. When a performance anomaly is detected, monitoring is extended to all the components that are used by the entry-level component for which the anomaly was detected [28]. However, even when in the 'off' mode, COMPAS probes can still induce some performance overheads at component level, as they are never actually removed from the application. If this overhead becomes significant, component redundancy can be used as an alternative method for switching component monitoring on and off. This is achieved by providing two redundant components, only one of which contains the monitoring probes. Switching the monitoring facility on and off is achieved in this case by alternating the activation of the monitoring-enabled redundant component with the normal (monitoring-disabled) variant. The monitoring-enabled redundant component contains an instrumentation probe in addition to its normal code. The advantage of using the component redundancy based approach in this case is that monitoring probes can be completely removed from the running system when the monitoring functionality is switched-off.

3.6 Business Application Scenarios

This section discusses several business application scenarios that would benefit from redundancy-based performance optimisations. The redundancy-based optimisation solution proposed in the thesis was designed for managing complex applications that experience significant fluctuations in their running environments. Such dynamic fluctuations can occur in an application's incoming workloads and in the amounts of available software and hardware resources. Workload is used in this context to indicate both the load on the application, as well the application usage patterns. The application load is given by the number of incoming client requests per time interval. An application usage pattern is determined by the particular sequence of incoming client requests of various types. Hardware resources available to an application include for example CPU, memory, disk, or network bandwidth. Software resources can consist of other software applications, such as databases, as well as of threads, processes and connections from the underlying middleware, VM and/or OS platform.

The proposed redundancy-based adaptation solution targets scenarios in which an application's environmental conditions vary significantly during the application's lifetime, but remain relatively stable for considerable periods between subsequent changes (e.g. more than an hour intervals between changes). The technical scenarios relevant for the proposed solution were presented in the previous section, showing how component redundancy can be applied to address variations in a system's execution context, QoS requirements and provided functionality. This section discusses several business scenarios in fluctuations would occur in a system's running environment. The goal of this discussion is to show how businesses experiencing the illustrated scenarios would benefit from the redundancy-based optimisation solution.

A first set of relevant business scenarios are characterised by customer behaviour variations. Such changes may influence the incoming loads and usage patterns on the supporting software applications. Several business-oriented examples relevant for this category were identified, including regular and irregular behavioural changes.

Load fluctuations on a business application can regularly occur, depending on the time of day, week, month, year, or around certain predictable events. For example, most enterprise applications would experience increased numbers of customers during working hours and decreased customer activity over night. A more concrete example, banking applications would have fewer customers over weekends and bank holidays than during normal working periods. Also, e-commerce applications would receive more requests prior to specific events than during normal periods and possibly diminished demands immediately following such events. As such, depending on the applications' business specifics, higher demands may occur before official holidays, sport events, cultural manifestations, or during sales periods. Anther example, online travel applications would experience load fluctuations depending on the relevant seasons. News agencies would typically receive increased request numbers during predictable public events, such as referendums, or presidential elections.

Load fluctuations on a business application can also occur due to unexpected events. In the e-banking application example, increased loads may be caused by large numbers of customers deciding to extract funds, close bank accounts, or convert foreign currencies as a result of sudden, dramatic changes in the financial market, or other external events. E-commerce applications may have to deal with unpredicted load fluctuations caused by a product promotion, or an unexpected market need. Incoming loads on online news agencies may grow dramatically when unexpected events take place (e.g. security alerts, or natural calamities). In such scenarios, redundant components can be activated to provide limited functionality and consume fewer resources. This allows the news agency application to support higher user loads on the given available resources. Limited functionality in this example would mean the agency only provides a static page with the most important news updates, rather than the normal interactive, detailed and dynamic news content.

Besides fluctuations in the incoming loads, an application's usage patterns can also dynamically change over a system's lifetime. Business scenarios representative for these cases include the example scenarios provided above. Namely, an application's usage patterns can dynamically change during predicted periods (time of day, week, month, or year), or as a result of unpredicted events. Causes triggering changes in load can also cause variations in an application's usage patterns. As additional examples, banking applications can experience increased demands for particular services, such as transaction account listings, for a several hours, at the end of each month. Such demands would be required in order to print out and send personal

bank account or credit card statements to account holders. Brokerage applications would receive large numbers of requests for retrieving current stocks information at the beginning of each working day. This pattern would occur as all brokers start by updating their local stock information, so as to get up to date with the overnight changes. Such read-only usage patterns on the brokerage application would be followed by mixed request for selling, buying and updating prices of stocks on the market.
Another category of relevant business scenarios includes cases in which the amounts of resources available to a managed application vary significantly over the application's lifetime. Business-related factors influencing this environmental characteristic include the acquisition of new software or hardware resources, or the allocation of available resources to other distinctive business applications.

3.7 Redundant Implementation and Configuration Examples

Several examples were identified to indicate how redundant implementations and configurations would address the thesis's performance optimisation problem. Cases in which performance optimisations were required by dynamic environmental changes were exemplified and discussed in the previous sections. The examples were selected to be representative for enterprise systems and the associated performance problems. This section focuses on showing how different implementation strategies should be used to address the performance optimisation challenges raised by such dynamic environmental changes.
A first example involves a component whose functionality is to repeatedly retrieve data from a remote location. Two possible strategies are available to implement this component, as follows. One strategy uses a local cache, while the other repeatedly retrieves the requested remote data upon each demand. Each redundant implementation can be optimal depending on a number of external factors. The amounts of available resources of various types can be a decisive factor when selecting the optimal redundant implementation. Namely, sufficient local storage capacity and processing resources may favour the caching-based implementation solution. The redundant solution not using a local cache is optimal otherwise, provided sufficient network bandwidth is available and no resource contention occurs at the remote storage location (e.g., remote DB). In addition, the optimal implementation choice in this example is directly influenced by the average hit rates on the local cache. More precisely, high hit rates would favour the local cache usage, while low rates would render the caching-related overheads inefficient. As such, the two redundant implementations can be used alternately in response to changes in the available relevant resources and usage patterns. An example application implementing the scenario described above is presented in more detail in section 5.1. The example shows how the response times of two implementation strategies are affected by network bandwidth variations. Test results were used to draw the response time curves of the two implementation strategies in rapport to variations in the available bandwidth resources. The results indicate that a cross point is given by the intersection of the two performance curves. The cross point indicates the execution context in which the two redundant implementation strategies should be swapped to achieve optimal performance.

Another example involving local caching choices is given by different possible usage patterns. The example involves the same functionality that repeatedly retrieves data from a remote storage location, such as a remote DB. Two possible implementation strategies are possible and the optimal one critically depends on the way this application function is used. One implementation strategy involves sequentially retrieving the data from the remote DB, in response to each individual request. In this case, every time the local application requires a piece of data, the data is retrieved from the remote DB. Required data can consist of a customer's account information in a banking application, or a product's details in an e-commerce application. This implementation strategy is particularly suitable for cases in which only small amounts of information are required from the DB, over certain periods. For example, only information for several bank customers or e-commerce products is required during a certain transaction. Nonetheless, a second possible usage pattern involves large amounts of sequential data being required over most business transactions. For example, the account details of all bank customers are retrieved at the end of each month, in order to print out and send monthly personal bank statements. Or, information on all products in a certain catalogue category is required for display in an e-commerce application. In this case, a different implementation strategy can be used to optimise the data retrieval process. Namely, when the first data item is required from the DB, an entire bulk of data can be retrieved together with the required item. The data bundle is cached locally and readily available for future use. In this case, subsequent data item requests find the required data already available in the local cache. Thus, this strategy saves the resources needed for repeatedly accessing the remote DB. This strategy is based on predicting the need for a large set of successive data items upon receiving the first data item request. Based on this, data is pre-fetched from the remote DB and readily available when the predicted requests are received. In this usage scenario, the cache-based implementation strategy potentially yields better performance than the first strategy that did not use a cache. Nonetheless, using the local cache to store pre-fetched data in usage cases when the data is not actually required causes unnecessary resources to be consumed. Thus, using a local cache in this case may not be the optimal strategy. Hence, the optimal implementation strategy directly depends on the way the targeted application function is used during runtime. Consequently, dynamic variations in the runtime usage patterns will require corresponding changes in the implementation strategy used. Alternating the use of the two redundant strategies so as to match the current usage patterns can provide optimal performance at all times.

Redundant component applicability is not limited to enterprise systems, but can be used to optimise applications from different domains. For example, different redundant algorithms can be used in scientific applications to meet various performance and processing resource requirements, as well as to react to changes in the application usage patterns (e.g. characteristics of the data sent to the scientific application for processing) [93]. Another example, redundant solutions can be adopted for storage applications and used alternately in response to usage pattern variations. The example provided in [101] shows how storage solutions based on relational DBs and LDAP directories are optimal depending on client usage patterns. Namely, a directory-based storage support is optimal for mostly read access types, while relational database solutions are optimal for mixed read and write access patterns.

Besides differences in the possible implementation strategies, optimisations can also performed at an application's configuration level. As such, caching and pooling configurations can be dynamically tuned so as to best match the current execution conditions, at all times. Such configurations can include cache and pool sizes, component instance lifetimes, or resiz-

ing policies. Dynamic configurations can also be envisaged to optimise the transaction types used for business sessions, depending on the current usage patterns. As such, in read-only usage scenarios, data can safely be cached locally without the risk of becoming obsolete. In this case, resources needed to synchronise local data with the database before and after each operation are saved. Nonetheless, in a mixed read-and-write usage scenario, various transactions both read and write data to the database. In this case, locally cached data risks to become outdated and should consequently be re-periodically synchronised with the shared DB. The optimal transaction and synchronisation policy critically depends on the current application usage pattern. Thus, the caching, pooling or transaction configurations should be dynamically changed so as to match the current application usage patterns. Redundant component configurations can be prepared and alternately used for this purpose. For certain component technologies, such as J2EE and CCM, the components' business logic is clearly separated from the middleware-related management services (e.g. life-cycle, transactions, or security) (subsection 2.6.2). In these cases, configuration settings at the middleware service level are specified via the components' deployment descriptors. These descriptors are generally xml documents that are bundled together with the component implementations into deployable component packages. In such cases, the deployable packages are considered as the basic redundant entities, with possible differences at both the implementation and configuration levels. An example scenario involving redundant configurations was implemented and tested as part of the thesis experimental work. The example shows how caching configurations are dynamically modified so as to be optimal under different workloads (section 5.2).

The redundant implementation and configuration strategies exemplified above pertain to adaptation scenarios that respond to execution context variations. Example scenarios also exist for using redundant strategies in response to functional requirement variations. As such, cases in which performance is traded off for more extensive functionality can be envisaged. For example, logging or debugging functions can be added to a component's normal functionalities, in order to track the component's activity and/or detect bugs. Such additional functions would detriment the component's performance, but may still be desirable in certain cases, for limited periods. Redundant components with normal functions and with added debugging functions can be alternately employed to address such dynamically changing functional requirements. As another example, redundant components with limited functionality may be used to address resource contention cases, where the normal component functionalities would be impossible to provide. Providing reduced services may be a desirable alternative to a completely unavailable service. In this case, redundant components provide different levels of service, with different resource utilisation requirements. The redundant component used is selected based on the current resource availability of the running environment.

The redundancy based adaptation mechanism can also be used to manage QoS attributes other than performance. For example, system reliability can be improved by using redundant components as functional backups for running components identified as faulty. Certain types of functional faults can be discovered at runtime in an application-independent manner. This can be achieved by catching thrown exceptions, detecting deadlocks, or sensing serious memory-leaks. Available redundant components can be used in these cases as replacements for the faulty components, until updated component versions are available to correct the existing faults. The presented examples show how the redundancy-based solution can be employed to address different adaptation requirements, at different system levels, in a uniform way.

3.8 Component Redundancy Costs

Cost-related concerns may be raised in association with the proposed redundancy-based optimisation solution. It may be argued that not many organisations would have the luxury of deploying multiple components that have identical functionality and different implementation characteristics. This is indeed a valid point that must be taken into account when adopting a redundancy-based optimisation solution. A trade-off must be considered between two major costs, as follows. On one hand, there is the cost of acquiring multiple redundant components and providing the system management and adaptation support. On the other hand, there is an associated cost incurred by a non-optimal system, maybe even to the point of loss of availability. A successful business scenario is one whose redundancy-based management solution's cost is largely exceeded by the revenues in performance and availability obtained by subsequently utilising the solution. In other words, a redundancy-based solution should be implemented in case the loss of revenue caused by running a non-optimal system is greater over time than the cost required to provide redundant components for the essential system parts.

The redundancy-based optimisation solution proposes a flexible way to performing system adaptation operations. It aims to provide improved support for system adaptation operations that are (or should be) already carried out at present, in order to avoid loss of business revenue. Namely, in many cases, no single component implementation or configuration exists that provides optimal performance under all possible execution scenarios. If this situation is detected at component design time, developers will attempt to implement different behaviours, each one optimised for a different scenario. All behaviours would be written as part of a single component and a certain if-then-else construct would be used to select amongst them, at runtime. In this case, all the supported behaviours, along with the adaptation logic for selecting which behaviour to use at each point are mingled in one single monolithic component. This approach is highly inflexible and proves rather costly and error-prone to manage. In this case, adding, deleting or modifying the (if-then-else) adaptation logic and/or various behaviours becomes an unnecessary complicated task. Furthermore, at design time, it is impossible to envisage all possible configurations, workloads and platforms under which a component will run during its entire lifetime. A component's behaviour needs to be seamlessly adaptable to unexpected changes in its running environment. Adding, deleting and modifying component behaviours while the system is running is hardly possible with monolithic components, but can be achieved when using redundant components. A redundancy-based management solution can allow component variants to be dynamically added, without interrupting system functionality. The AQuA management framework is proposed to automate system optimisation tasks, by knowledgeably swapping redundant components at runtime.

In a possible scenario the system does not initially make use of redundant components. In this case, functionality is provided by a single component implementation, as in any 'normal' system. This scenario is supported by the redundancy-based solution, as it is not compulsory for multiple redundant components to be available at runtime. Subsequently, supposing that while the system was running, it was observed that in several execution environments the current system configuration performance was unexpectedly poor. If this situation was rendered unacceptable from a business perspective, a new component can be deployed for optimising system performance in the detected cases. This is a viable choice if the cost of acquiring the new component is evaluated to be lower than the cost of maintaining the system unchanged.

In this case, continuing to run a non-optimised system would possibly result in loss of business clients and revenue. It can also lead to an inefficient use of system resources, which may in turn affect other business applications on the same platform. In this case, replacing an inefficient component with a new one may solve the initial performance, availability or reliability problems. Nonetheless, such system adaptation may also induce new quality or functionality problems, for previously satisfactory execution scenarios. As such, the redundancy-based adaptation solution proposes maintaining the two redundant components available and providing the ability to alternate their use so as to provide optimal system behaviour. It can also be argued that acquiring a new component which is only optimised for a certain execution environment is less costly than ordering a new complete replacement component that optimally handles all execution scenarios. In addition, completely replacing a partially-working component may also be more risky than only replacing the functionality of its faulty parts. This is because a complete substitute (e.g. a new COTS component) may also replace the functional component implementations that have already proved to work in the current system context.
In addition, an important point related to component redundancy costs is that, in some cases, providing multiple component variants is not necessarily expensive. For example, redundant components can merely differ in their deployment configurations (e.g. section 5.2). In such cases, the cost of providing multiple redundant components is insignificant, as it merely implies different deployment descriptor settings.
Another possible scenario is that in which the redundancy-based management framework is used to support component versioning operations. This process involves the complete replacement of current system components with newer, updated versions (e.g. [77]). Such operations may be desirable in order to add new functionality, fix a detected bug, or provide better performance. The redundancy-based solution can support the component versioning process so as to minimise risks and increase system reliability. Namely, considering the case in which the system works correctly under the current configuration, but is non-optimal. For this reason, a new component version is acquired and deployed to optimise the system. In a typical component versioning scenario, the old component version is simply discarded. When the redundancy-based management is employed, the old component version is also kept, as a redundant component variant. The reason is that there a situation may occur where the new component version proves to be incorrect or not integrate properly with the rest of the application. Even if the new component was thoroughly tested, the incorrect or non-optimal behaviour may have not been detected during the static testing procedures. If this happened, the redundancy-based management framework could detect the problem and consequently reverse the system to its initial configuration. The old component is already known to work, even if performing poorly in certain scenarios, or providing limited functionality. When the new component version is determined to meet its functional and QoS requirements the old redundant component can safely be removed from the system, as in the typical component versioning approach.

3.9 Overview of the AQuA Framework

A management framework was devised to support the component redundancy concept and automatically optimise component-based system performance. The developed framework is

referred to as *AQuA (Automatic Quality Assurance)*. AQuA's goal is to enable applications to fluidly mould to their constantly changing running environments. For this purpose, AQuA automatically optimises and adapts applications to variations in their execution environments so as to maintain system performance levels optimal at all times. Although the current focus is on managing system performance and availability, the general concepts and design of AQuA can be extended to support additional system quality attributes, such as dependability.

The AQuA framework was devised as an approach towards automating the performance management of complex, component-based software systems. The proposed solution is based on the tested assumption that there are frequently no unique component implementations or configurations that can yield optimal performance under all possible execution environments (e.g. [30] and [33]). Based on this consideration, the thesis proposes the use of component redundancy to address the problem and permanently provide optimal system implementation and configuration solutions. The AQuA framework was developed to capitalise on component redundancy and automate system performance management. The framework was designed to support and manage redundant components, using them to continuously optimise and adapt software applications at runtime. As such, AQuA dynamically modifies the managed applications' implementations so as to optimise them for the current environmental conditions and persistently meet application performance goals.

The AQuA framework aligns with the autonomic computing initiative proposed by IBM for automating management processes in complex software systems. It also conforms to general management frameworks proposed in related literature, such as in [72], [43], or [54]. Thus, the main management functionalities provided by AQuA involve system monitoring, performance anomaly detection, component evaluation, adaptation decision and component activation. The performance anomaly detection, component evaluation and adaptation decision functionalities form AQuA's adaptation logic. The adaptation logic is used to decide on the application adaptation strategies that are to be taken based on the available information. A set of decision policies are specified as part of the adaptation logic to express the desired system management behaviour. At a design level, AQuA's management functionalities are grouped into three main logical modules, namely the *monitoring and detection*, *evaluation and decision* and *component activation* modules (Figure 3.4). The functionalities these modules provide enable self-managing software systems to:

- Monitor themselves and their execution environments during runtime
- Analyse collected monitoring data and detect performance problems
- Evaluate available adaptation and optimisation alternatives
- Decide on changes to perform in order to overcome detected problems and improve system performance
- Dynamically enforce taken decisions by modifying running applications

The monitoring, adaptation logic and action functionalities are interconnected in a closed control feedback loop (Figure 3.4). In short, the monitoring module collects performance data from the running application. Collected data is analysed and performance anomalies detected or predicted. The available optimisation solutions are evaluated and an adaptation decision taken. Consequently, adaptation decisions are enforced into the running application by means of redundant component activations. The application is subsequently re-monitored

and re-evaluated for assessing the benefits of the implemented adaptation strategy. This way, AQuA can learn and improve its management behaviour for each particular application, over time. Special-purpose evaluation and decision logic is used to address potential stability issues, which may be induced by the adaptation feedback-loop.

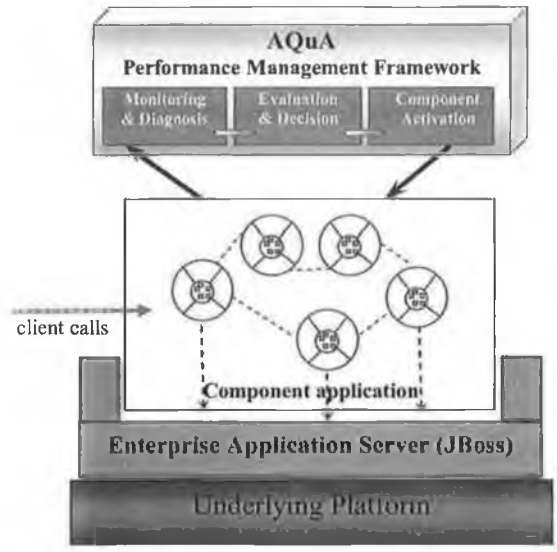

Figure 3.4: architectural overview of the AQuA framework

The main roles and functionalities of the AQuA framework are presented over the following sections. A framework prototype, AQuA_J2EE, was implemented in order to test the way these functionalities work together for managing the performance of J2EE applications (chapter 4). AQuA's modular architecture allows for each of its functional parts to be designed and implemented independently, as well as subsequently replaced without affecting the other modules. Thus, various strategies can be separately selected for detecting performance problems and their causes, defining the adaptation logic, or implementing the component activation functionalities. As such, various instrumentation approaches can be adopted for supporting AQuA's monitoring function. Also, diverse centralised, distributed, or mixed solutions can be devised for AQuA's adaptation control topology. The adaptation logic can be implemented based on decision policies, plan-based, or goal-oriented schemes. Multiple solutions are possible for enabling AQuA to perform dynamic component-swapping operations that remain transparent to its external clients. The goal of the thesis is to propose a redundancy-based solution for the automatic performance optimisation of complex component-based systems. Part of this objective, the thesis also seeks to provide a proof-of-concept design and implementation for the proposed solution. The thesis does not attempt to provide an optimal, fully-functional management framework product. The implemented framework prototype shows how the redundancy-based solution works for automatically managing component-based enterprise applications. Further, any of the framework's functional modules can be augmented and developed towards obtaining a robust solution for a real management scenario.

3.10 Runtime Monitoring

AQuA's *monitoring* functionality is responsible for collecting runtime data from the managed components and their execution environment. This implies that data is collected exclusively for the active redundant components. Monitored events can be collected from running applications and used to calculate the monitoring metrics of interest. For example, method invocation and method return events can be intercepted and analysed for this purpose. These events can be used to compute incoming workloads and performance data such as average response times and throughputs. Component instantiation or removal events can also be obtained from the running system. These events can be used to indicate to current number of instances that are available at each point for each application component. Monitored data on the applications' running environments includes the amounts of available resources, such as CPU, memory, disk, or network bandwidth. Monitoring data is used for detecting performance anomalies and important variations in the components' execution environment. In addition, as part of AQuA's learning function, monitoring data is stored and analysed so as to infer higher-level information on the quality characteristics of the managed redundant components.

Two main implementation strategies can be adopted for instrumenting the system and acquiring runtime monitoring data. These are application-level instrumentation, based on component proxies, and server-level instrumentation, based on container interceptors (Figure 3.5). For the two instrumentation solutions, the trade-off is between portability across application servers and the effortless management of any new application on a certain server (subsection 2.6.5). Namely, the proxy-based solution is portable across application servers, but requires an instrumentation effort for each new managed application. On the other hand, the server-level instrumentation must be reimplemented for each different application server used, but is subsequently available for all managed applications deployed and run on that server. A similar choice is available for implementing the component-swapping function, to use either application-level proxies to indirect client calls to the currently active redundant component, or to modify the application server containers to perform such operations. The server-level instrumentation approach was adopted for the AQuA_J2EE prototype, which was implemented as part of the thesis experimental work (chapter 4).

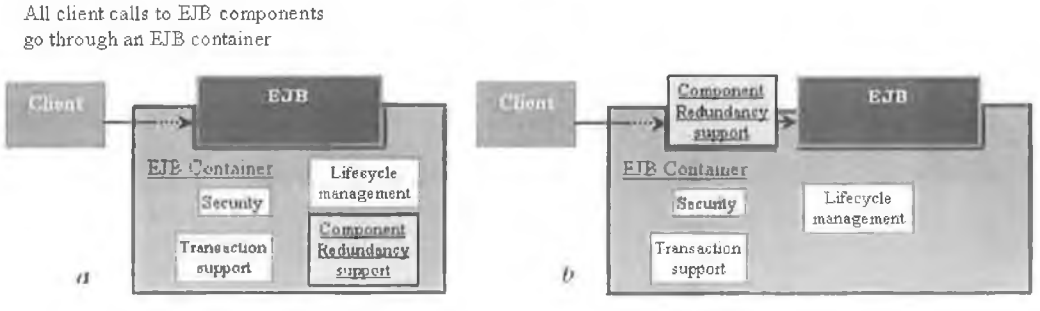

Figure 3.5: possible instrumentation approaches:
a - container level; b - application level

3.11 Adaptation Logic

AQuA's *adaptation logic* uses runtime monitoring data to automatically find optimisation solutions to detected performance problems. In the context of the proposed redundancy-based solution, the adaptation logic decides *which* redundant components should be activated and *when*, in order to optimise application performance. The decision process involves two main activities, namely, accumulating and processing system information. The type of information acquired and the manner in which it is analysed depend on the particular aim of the different types of adaptation logic involved. Specifically, AQuA's adaptation logic is used for different purposes. These include the performance anomaly detection, the component evaluation and the adaptation decision functions. In short, anomaly detection logic is used to identify and/or predict possible performance problems. Component evaluation logic is used to identify and propose remedial solutions to identified performance problems. For this purpose, the evaluation logic assesses the available redundant components and proposes the optimal ones for activation. Finally, adaptation decision logic is used to evaluate the available optimisation solutions overall, and conclude on the adaptation actions to be carried out in the system.
System information is required to support all the aforementioned adaptation processes. As such, runtime monitoring data is collected and analysed in order to support the performance anomaly detection process. Monitoring data on the system's current status and information on the redundant component's performance characteristics are used for the component evaluation process. More precisely, first, information is acquired on the performance characteristics of the available redundant components. This information describes the components' respective behaviours, with respect to performance, in various execution contexts. Additionally, monitoring data on the current execution environment of managed components must also be obtained. The component evaluation process analyses the available information and determines the optimal redundant components, in the targeted execution contexts. Performance information on redundant components is obtained from collected runtime monitoring data and then processed and stored in formal component descriptions (subsection 3.11.2 and section 3.13). Finally, the adaptation decision process is based on monitoring data from the framework's control feedback loop, as well as on history information on previously taken adaptation decisions and their outcomes. To summarise, component description information along with monitoring data from the current system and its running environment are used as input to AQuA's adaptation logic. The available input information is critical to support the performance optimisation decision process.
Two main approaches can be considered with regard to the control topology of AQuA's adaptation logic (subsection 2.6.4). These are based on fully centralised or decentralised control architectures. Mixed solutions are also possible and indeed desirable in most cases for combining the benefits of the centralised and decentralised solutions. First, the *centralised* approach can be adopted to globally analyse, evaluate and adapt the entire application. In this case, in order to optimise a certain business transaction, all RGs involved in implementing that transaction are evaluated in a centralised manner. The optimal combination of redundant components, one from each RG involved, is selected for activation. The redundant components to activate are selected so that the entire considered transaction is optimal overall. The fact that a certain RG can be involved in multiple, separate transactions, each one with different performance requirements, also needs to be considered when selecting the optimal redundant component to activate. Various analytical methods can be adopted when considering

the centralised management approach (subsection 2.6.4). For example, graph optimisation techniques can be applied to find the most favourable application configuration based on a centralised application model. Another possible approach is to apply expression optimisation methodologies such as the ones used in relational databases for query optimisations [14]. Nonetheless, when considering large-scale component-based applications, global optimisations may not always be needed. Evaluating an overall application, potentially consisting of hundreds of components, whenever an individual component or a group of components does not meet performance expectations, might induce unnecessary overhead and not scale well.
Therefore, a second approach is to implement AQuA's adaptation logic in a *decentralised* manner. In this approach, if a problem is detected at an individual component level, the problem is managed locally, by means of redundant component replacement. Thus, only components exhibiting performance problems are analysed and affected by local optimisations. This approach is potentially more scalable than the centralised one, as it avoids repeated and possibly unnecessary optimisations of the entire application. Nonetheless, exclusively concentrating on local optimisations might lead to a non-optimal global application. Also, certain problems such as deadlocks, oscillating states or chain reactions, cannot be detected or solved at an individual component level. Hence, a more high-level view is needed to detect and solve such cases.
For the aforementioned reasons, a mixed solution is optimal for the adaptation logic control topology, combining the benefits of both centralised and decentralised approaches. For this purpose, a *hierarchical* topology can be designed for the adaptation logic, as a combined solution which is both scaleable, as well as capable of achieving the system management goals. In this combined approach, framework instances with different scopes are organised in a hierarchal manner. Possible management scopes include the single component-level, component group-level and global application level. In this scenario, detected anomalies can be managed locally and/or signalled vertically up the management hierarchy to the global level. A clear protocol must be specified in this case for allowing management instances with different scopes to communicate. This approach allows for local, component-level anomalies to be solved locally, when possible, while also supporting global optimisations, when necessary. Framework management instances at various hierarchical levels can be dynamically activated or deactivated, in order to reduce overheads, while meeting system management goals.
Another combined approach is possible for adding global management capabilities to a decentralised solution. As with the previous, hierarchical approach, local management operations are performed at the individual RG level, independently from other RGs. Also as before, a separate framework instance is created for this purpose to locally manage and optimise each separate RG. Global system optimisation is achieved in this solution by enabling the local framework instances to communicate and combine their actions. The communication protocol and local framework behaviour must be designed in such a way that the emerging management behaviour provides global optimisation solutions.
The current AQuA_J2EE framework prototype uses anomaly detection, component evaluation and adaptation decision functionalities with exclusively local, component-level, scopes. This means that each individual RG is being optimised separately from other RGs. One of the previously described approaches can be adopted and added to AQuA's implementation for achieving global optimisation solutions. Figure 3.6 shows an overall view of the way AQuA_J2EE is integrated with an EJB server. As indicated in the figure, multiple framework instances are created, one for each managed component. Nonetheless, instrumenting and managing

all application components at all times might not always be necessary. The unnecessary performance overheads can be avoided by enabling AQuA to only manage a selected subset of application components. The current AQuA implementation allows system administrators to statically specify the set of EJB components and methods to be managed. This functionality can be augmented so as to also support dynamic or automated selection of managed components. With this approach, framework instances are only created to manage selected components, as specified in AQuA's configuration settings. All framework instances use the same type of control cycle for managing the components for which they were created. Control cycles include monitoring and detection (M), evaluation and decision (E) and component activation (A) functionalities, working in a feed-back-loop manner.
AQuA's adaptation logic was designed based on decision policies, as described in the following subsection. The adaptation logic and corresponding decision policies can be divided into three main categories, based on the functions they need to provide. These are the performance anomaly detection (subsection 3.11.3), component evaluation (subsection 3.11.4) and adaptation decision (subsection 3.11.5).

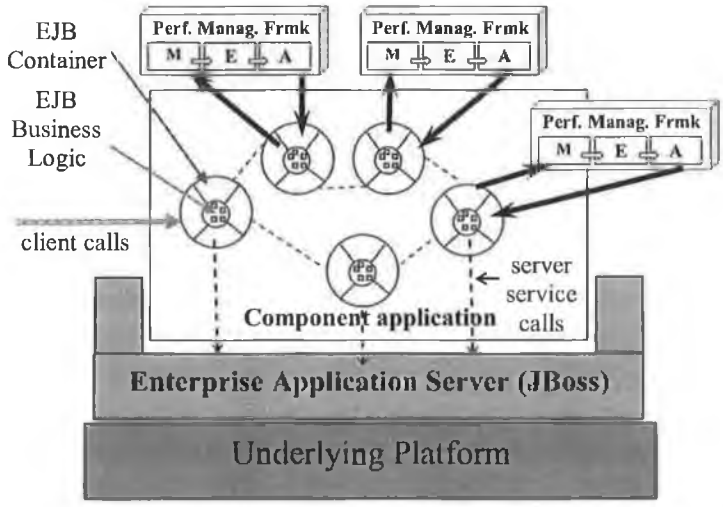

Figure 3.6: decentralised, component-level management: M - monitoring and detection, E - evaluation and decision, A - component activation

An important issue that arises with respect to AQuA's adaptation logic is the *trust* that can reliably be placed on a framework's automated management functionality for successfully adapting software systems. As pointed in section 2.6.6, an expected scenario is one in which a human manager initially performs such tasks, assisted by AQuA's automated monitoring and basic anomaly detection facilities provided. Subsequently, AQuA's component activation functionality can be used, via a special-purpose management GUI, to manually enforce taken adaptation decisions. In succeeding phases, the framework is progressively enabled to take automatic adaptation decisions. First, basic decision policies start to be specified, so as to automate simple administrative tasks. Over time, AQuA's adaptation logic is gradually extended, so as to automate increasingly complex management tasks. Thus, human decisions are steadily replaced by automated policies. The final goal is to automate as much of the adaptation decision process as possible. The goal is to achieve an automated management behaviour that was the same or better than in the case human system managers were in charge. When

this is attained, the automatic system adaptation framework will provide a less costly and less error-prone solution to system management. This goal is pursued by progressively defining AQuA's adaptation logic, based on repeated system observations and management experience. Behavioural patterns that are clearly and repeatedly observed in the human administrators' management behaviour are formally represented as policies in AQuA's adaptation logic. AQuA subsequently uses these policies to automatically suggest, or take, simple adaptation decisions. Suggested adaptation decisions may initially need to be verified and confirmed by a human administrator, before they can be enforced into the running system. Initially, basic adaptation decisions will solve common, unambiguous situations. More complex policies are incrementally added in time, enabling the adaptation module to automatically deal with more complicated, unpredictable conditions. AQuA's learning process can be performed by a human manager, based on their observations and expertise. In a more complex scenario, decision policies can be automatically inferred by AQuA's adaptation module itself (supervised by a human manager) based on automated data analysis and policy specification processes (subsection 3.13).
The current design of AQuA's adaptation logic is based on decision policies, further discussed over the following subsections. Other approaches are also possible for implementing the adaptation logic, such as based on activity plans or goal-oriented schemes (subsection 2.6.6). AQuA's design modularity allows for any of its functional parts to be independently modified, without affecting the other modules.

3.11.1 Decision Policies

Decision policies are sets of *rules* that dictate the actions to be taken in case certain conditions are satisfied. In AQuA, decision policies are used to implement the framework's adaptation logic, for taking performance management decisions. In other words, decision policies are used to specify AQuA's adaptation logic and coordinate its management behaviour during runtime. This approach clearly separates the application management strategies represented by decision policies from the application data and business logic. As a consequence, human system administrators can use decision policies to formally specify their management expertise, in a format that automated management frameworks can interpret. In addition, specifying adaptation logic via decision policies does not require a thorough understanding of the underlying framework mechanisms and implementation. Policies can be added, modified or deleted independently of other AQuA functionalities or implementations. System administrators can also use decision policies to specify high-level management goals, such as system performance or reliability objectives. Thus, AQuA's decision policies are designed and configured separately for each managed application, so as to serve the specific application goals. Management goals can be specified in terms system attributes and their corresponding value ranges. For example, performance goals can be stated in terms of response times and throughput values. A criticality factor can also be associated with a goal, or goal attribute. The criticality factor would indicate the importance of each goal with respect to other goals. Thus, criticality factors would allow certain goals to take precedence with respect to the other goals, thus solving conflicting situations in which different adaptations are needed to achieve separate goals.
Decision policies, or rules, are split into two main categories: *basic rules* and *high-level rules*. The separation is based on the decision policies' management scope, or level at which they

operate. Basic rules control the managed software application. They react to events in the system by evaluating the events' relevance and computing remedial solutions. High-level rules are used to control the behaviour of basic rules. They analyse the basic rule behaviour, so as to detect and rectify undesirable management behaviour. Thus, high-level rules can decide when it is necessary to interrupt a basic decision process. For example, a basic management procedure can be suspended in case it became too costly (in time or in consumed resources) or seemed to have entered an infinite loop, such as an oscillating state or a chain reaction. The basic rules' management behaviour is evaluated by analysing the state of the basic decision process, as well as the way it evolves over time. For example, a simple high-level policy could state that a basic decision process will be interrupted if it took more than a certain amount of time to compute without reaching a conclusion. In a more complex scenario, a higher-level process records a history of the adaptation decisions taken by the basic process. A high-level policy subsequently analyses the recorded history and identifies any cyclical patterns in the basic decision process. The basic decision process is interrupted, and/or updated, in case an undesired pattern was discovered in its management actions. The current framework prototype provides support for specifying and interpreting basic rules. Several basic rules were defined to implement the prototype's adaptation logic for the performed tests (section 5.2). High-level policies were not implemented in the current framework prototype version.

Basic rules can be further classified into different types, depending on their intended functionality. Four such types were identified for AQuA's adaptation logic, as follows. The four basic decision policy types are described in more detail over the following subsections. First, *anomaly detection* rules are used to analyse monitoring data and discover performance problems. They can also sense important variations in the system performance, or system execution environment. Second, *component evaluation* rules are used to determine the optimal redundant components for a given execution environment. Their objective is to evaluate the current system state and find optimal solutions for overcoming detected or predicted problems. In a third category, *adaptation decision* rules are used to take final decisions on the system optimisation solutions. Final optimisation solutions are subsequently enforced into the running system. A fourth, separate category of rules was defined for specifying AQuA's learning capability. The goal of the learning function is to allow AQuA to improve its management behaviour over time. Rules in this category are used for inferring new facts, or information, from existing, collected data. Enabling the management framework to automatically improve its knowledge and behaviour avoids imposing extra requirements on system administrators, component providers, or testers. In its current design, AQuA uses a learning process to acquire accurate information on the available redundant components. The manner in which this learning functionality was designed, implemented and tested is discussed in sections 3.13, 4.5 and 5.3. Additional learning capabilities can be devised for AQuA, in order to automatically update its adaptation logic and consequently improve its management behaviour. Such learning processes would be implemented based on a closed control feedback loop, such as designed as part of AQuA's general architecture (Figure 3.4). Part of the feedback loop, runtime monitoring data is analysed in order to assess the results of previous system adaptations. The framework's decision logic can be accordingly modified and tuned in effect. As such, learning procedures can be implemented to automate the creation and configuration of rules of various types. Namely, learning strategies can be designed for automatically managing component evaluation, or adaptation decision rules. Theoretically, any management rule, whether at the basic or at the higher levels can be automatically updated and improved based on a learn-

ing process. To extrapolate, the learning processes themselves can be automatically tuned by higher-level learning procedures. The level of complexity for such learning functions should be limited depending on the systems' management requirements and the costs involved.

3.11.2 Component Descriptions

Component descriptions are used to store and provide non-functional information on the managed components. This can include information on a component's performance characteristics, or on other attributes, such as reliability, security, or correctness. In the proposed redundancy-based management solution, each redundant component has its own component description associated with it. The information stored in each component description allows the system management process to evaluate the available redundant components and accordingly take optimisation and adaptation decisions. The available component information is critical when selecting the redundant components to be activated in the system in a certain context.

Component performance information is essential for the proposed redundancy-based optimisation solution. This information is acquired by repeatedly collecting and analysing monitoring data samples, on the relevant performance-related metrics. Monitored metrics of interest include response times, throughputs, workloads and resource availability. The process can be initially performed offline, whether in testing environments or on the targeted execution platform. The information acquisition process can be subsequently extended during runtime, when more relevant and accurate monitoring data can be collected. The actual process of analysing monitoring data and inferring information on the components' performance characteristics can be initially performed manually and then progressively automated. Conceptually, this process is carried out as follows.

Performance data samples are obtained by AQuA's monitoring module, at runtime. Collected data is subsequently processed and formally represented as a component description, or metadata, facilitating its automatic interpretation, analysis and modification. Component providers can optionally supply initial component descriptions, at deployment time. An initial component description can indicate the implementation strategy used, or the running context for which a component was optimised. For example, a component description can specify that the component implementation was optimised for running under increased workloads and having certain amounts of CPU resources available. A component description can also provide relative values for performance attributes such as delay or throughput, and/or their sensibility to execution context variation. For example, a component description can indicate that the response time for a certain component method increases exponentially with the incoming workload on that method. This sort of information can be acquired from test results, estimations, or previous experience with provided components. As such, initial information is to be considered as general guidelines rather than as absolute figures. This is because the data from which the initial information was inferred was obtained in execution conditions that were different from the current system conditions. As such, performance results will significantly differ when a component is run under different workloads and on different application servers, Java Virtual Machines (JVMs), Operating Systems (OSs) and hardware platforms.

Another possibility exists for obtaining initial performance information on a managed application and its constituent components. This approach involves testing the application while it

is deployed and run in its targeted execution environment. However, the testing procedures are performed while the application is kept offline, unavailable to external, business clients. In other words, the application is deployed and run on the targeted software and hardware platform on which it is to be made available when online. Testing workloads used are designed so as to resemble the predicted workloads that the application is to experience when online. This approach provides a testing execution environment that is very close to the targeted execution conditions, which are to be encountered by the system when online. This fact is also valid for component-level performance tests, since each component can be tested while integrated in the targeted application. This has the potential to provide more accurate and reliable predictions on the components' performance characteristics, than if each component was tested in isolation. The offline testing approach was adopted as part of this thesis' experimental work, in order to obtain performance information on the tested application components (section 5.2).

Runtime learning procedures can also be used for automatically analysing monitoring data and obtaining reliable component information. Part of the learning process, initial component descriptions are repeatedly verified and updated during runtime, whenever new system monitoring information becomes available. This way, component descriptions become progressively more accurate and reliable over time. This is because the monitoring information obtained during runtime, in the actual targeted execution context, provides an accurate view of a component's performance characteristics. This is not always the case with performance results obtained in testing conditions.

AQuA can use runtime monitoring data to automatically 'learn' about the performance characteristics of the software components and application it has to manage. This in turn can progressively improve AQuA's management behaviour over time. The reason is that the adaptation logic used to automatically manage the system critically depends on the available knowledge that exists on the redundant components' performance characteristics. This includes information on the redundant components that are optimal in certain execution contexts, as well as on optimal combinations of redundant components, in different running environments. Component descriptions include formal representations of value set lists. Each value set represents a component's characteristics in a given execution environment. Namely, a set comprises multiple values, one for each considered execution context and associated performance metrics. For example, a set can comprise parameter values for the available CPU, memory and bandwidth, associated with the observed response times and throughput values. These values indicate the component's performance characteristics under the associated execution environment. As such, in a component's description there will be a different performance record associated with each distinct execution environment encountered.

Component performance descriptions are obtained and verified by the learning process based on collected monitoring data samples. As such, besides the component performance information, raw monitoring data samples are also collected and stored as part of the component descriptions. Specifically, runtime monitoring data samples are collected and stored in component descriptions, forming histories of data, separately for each provided method. Thus, any component description stores a separate history of monitored data samples for each externally visible component method. Each history in the component's description stores a (sliding window) sequence of chronologically-ordered data samples. The maximum history size, indicating the maximum number of data samples that can be stored in a history, is a configurable parameter. Each monitoring data sample contains the date when the sample was collected,

external performance data, including response time and throughput, and data on the environmental conditions in which the monitored component was running at the moment the sample was collected. The environmental conditions related data includes incoming workloads and available resource amounts, such as CPU, bandwidth, memory, or disk. Additional environmental data can be obtained and stored as needed, or as relevant, including for example the identities of the neighbouring active redundant components. An initial implementation of the automated learning process was completed and tested, as presented in sections 4.5 and 5.3.

3.11.3 Anomaly Detection Policies

AQuA's *anomaly detection* functionality is responsible for identifying and signalling the occurrence of performance problems, or of relevant variations in the application's performance or execution environment. Availability or reliability concerns can also be raised in case exceptions are being caught by the monitoring module (e.g. out-of-memory Exception). Performance anomalies are generally signalled when performance metrics such as response times and throughputs do not meet the system's performance requirements. A number of basic anomaly detection strategies have been implemented. New strategies can be seamlessly plugged into the framework as they become available [28]. When performance anomalies are detected, it means that the system is already experiencing performance problems, which need to be promptly eliminated. This situation can be avoided in some cases by identifying and analysing the variations that occur in the execution environment and predicting how these variations will potentially affect the system's performance.
In case AQuA detects that a certain component might generate performance problems under a new execution environment, it acts immediately to adapt the application. The potential problem component is replaced with a more suitable one, if available, so as to prevent performance difficulties before they occurred. Relevant variations in the execution environment are detected by constantly monitoring the environmental metrics of interest and periodically analysing the monitored data. Detected environmental variations are used as triggers to the automatic application evaluation and adaptation process. Variations in the incoming workloads and resources availability are considered for this purpose. The principal idea behind this strategy is that if at a certain point a system is meeting its functional and performance-related requirements, then the system will generally continue to do so unless a change intervened to alter this state. In one of the performed experimental testing scenarios (section 5.2), AQuA detects changes in the incoming workload and uses them to trigger the application evaluation and adaptation processes.
The occurrence of relevant variations in the execution environment is detected by constantly monitoring the environmental metrics of interest and periodically analysing the monitored data. Environmental variations that can be used as triggers to the application adaptation process include changes in the incoming user load, work mix, or resource availability of the system. For example, significant changes in the incoming user load can be detected and used to trigger the application adaptation. Another possibly useful variation to consider is a change in the application's incoming work mix, from a read-only operations mix to a mix of both read and write data accesses. Such variation can be useful to detect, as different redundant implementation or configuration approaches can be adopted for each case. For example, for Entity EJBs, the transaction commit option in the EJB's container can be set to option A for

read-only work mixes and to option C for read and write work mixes (appendix B.1), with a dramatic effect on the resulting EJB response times. Time intervals during which certain work mixes occurred can be identified by monitoring and analysing sequences of incoming method requests. A trigger as simple as the arrival of a first 'write' operation can be used to signal the start of an interval with mixed 'read' and 'write' operations. Conversely, the lack of write operations for certain duration can signal the start of an interval with read-only operations.

A further possibility exists for predicting performance anomalies based on execution context variations. This opportunity occurs when the environmental variations themselves can be envisaged. In such cases, applications can be pre-emptively adapted to deal with the foreseen changes before they actually happened. For example, certain applications may experience workload variations with the time of day, week, month or year. For example, online banking applications may expect reduced user loads during non-working hours. E-commerce applications would expect increased loads before certain events or during sales periods. When such distinctive intervals can be predicted with sufficient accuracy, a human system manager can instruct AQuA (via decision policies) to automatically activate a different redundant component during each period.

AQuA's anomaly detection functionality is specified based on decision policies. Anomaly detection policies indicate the conditions in which performance alarms are being raised. The analysed conditions generally consist of a certain set of monitored metrics and a corresponding set of value ranges. Based on these specifications, a condition is met when monitored data matches the metric values that define the condition. Different types of performance anomalies can be defined, along with the corresponding detection conditions, or patterns. Thus, when certain conditions are met, detection policies raise an alarm, indicating the occurrence of a certain performance anomaly type.

Detection policies are evaluated periodically, as new monitoring data on the components' performance and on the running conditions becomes available. Detection policies analyse histories of monitored metric values, in order to identify various anomaly types. A history of sequential values is maintained for each monitored metric. In general, the detection policies analyse new received metric values together with the stored history values. The analysed value sequence is formed by adding newly monitored values to the stored history values. Detection policies search the available sequence data for patterns that would indicate the occurrence of a performance problem. Possible opportunities for performance optimisations, caused by significant variations in the running environment, can also be signalled.

Various anomaly detection patterns can be defined and configured by system managers for each particular application [31]. For example, one pattern can be set to detect when current performance values exceed certain preconfigured thresholds. Another pattern can add the constraint that thresholds need to be exceeded for a certain period of time. Patterns for detecting relevant running environment changes can also be set. Detection policies analyse newly monitored data together with previously stored metric values. If the available value sequence matches one of the predefined anomaly patterns, a performance anomaly alarm is raised and the evaluation process is subsequently alerted.

In the current AQuA implementation, a single value is stored as part of each metric's history value sequence. In other words, for each metric, only the most recently monitored value is being stored as history data. When a new metric value is received as input it is compared against the history value stored for that metric. A threshold-based approach was adopted for detecting performance anomalies based on the two available metric values. Specifically, an

alert is considered in case the two metric values are on different sides of a certain configured threshold. Additional detection policies are specified to avoid cascading or false alarms. For example, decision policies should avoid raising repeated alarms in cases in which small oscillations occurred around the specified threshold value. An insensitivity interval is specified for this purpose around the targeted threshold. Detection policies perform an extra verification for determining whether newly monitored metric values actually exceed the predefined threshold with more than the specified insensitivity interval. If this is the case, it is considered that a performance anomaly may have occurred. Consequently, the component evaluation process is executed to remedy the situation. The implementation details of this process are more thoroughly described in subsection 4.6.1.
When identifying a performance problem, detection policies trigger the execution of the available evaluation and adaptation policies. These policies are used to process existing information on current component descriptions and monitoring data and to take optimisation and adaptation decisions. They can also be scheduled to run periodically, for optimisation purposes, provided that sufficient resources were available.

3.11.4 Component Evaluation Policies

AQuA's *component evaluation* functionality is responsible for determining the redundant components that are optimal in a given execution environment. The component evaluation process is based on the performance information that exists on the available redundant components, at the time the process is being executed. Thus, the accuracy and reliability of the information available to the evaluation process for predicting optimal redundant components becomes a critical factor. Another important factor is the actual logic used to interpret the available information and take evaluation decisions. Thus, the component evaluation functionality requires:

- accumulating information on components and their running environments, and
- processing the available information so as to determine the optimal redundant component(s), in certain execution contexts.

Component performance information can be initially obtained at component deployment time and/or dynamically inferred from runtime monitoring data (section 3.11.2). Initial component information is typically acquired based on test results and/or previous experiences with the considered components [30], [28] and [31]. Subsequently, as part of AQuA's learning process, the existing component information is dynamically validated and constantly updated, based on accurate monitoring data, collected from the targeted managed system. Additionally, in case initial component descriptions are not provided, the learning mechanism is used to obtain this information from scratch.
Component evaluation results represent possible application optimisations in the targeted execution context. Such optimisation solutions are given in the form of a set of redundant components that are considered optimal in the targeted execution context. Evaluation results have a confidence level associated with them, depending on the reliability of the information used in the evaluation process. The more data is used to test and reconfirm a piece of information, the higher the information's reliability factor and thus the higher the confidence level associated with decisions taken based on this information.

The component evaluation process uses existing component information to predict the component's behaviour in the future. As such, when a component is being evaluated, performance data that was collected for that component in a certain execution environment is used to predict the performance of the same component when running in similar environments. At runtime, monitoring data samples are repeatedly collected for the active redundant components and their execution environments. As such, a considerable number of data samples can be available to describe a component's previous behaviour under a certain execution context. These data samples are used to predict the component's behaviour when running in similar execution contexts. For this purpose, a first solution is to select and analyse all data samples relevant for the prediction process each time a component evaluation is required. Namely, all the available data samples on a certain component in a targeted execution context are evaluated whenever the component's performance needs to be predicted for a similar running context. This solution can be optimised by periodically analysing the available data samples and inferring higher-level information on the components' performance characteristics in different contexts. This is done by merging monitoring data collected in similar execution conditions into clusters of information. Information clusters can then be readily used when evaluation decisions are needed. Part of this optimised solution, monitoring data collected in certain running conditions is compared and merged with existing monitoring data recorded in similar running conditions. The more monitoring data samples are available for inferring the performance characteristics of a component in a certain running context, the higher the reliability of that performance information and the higher the confidence level when predicting the performance of that component in a similar context. An information-inference learning process was designed for AQuA to implement this automatic procedure (section 3.13). A further optimisation can be achieved by also performing the evaluation process periodically, at the RG level. This way, information on the optimal redundant component in each RG, in each execution context, is also readily available. AQuA's component evaluation function can then directly use this information as needed.

These possible solutions differ in the timing when computational resources are being consumed for the evaluation process. In the first solution, processing resources are being consumed upon each evaluation request. In this case, all the available data samples are being processed each time an evaluation decision is needed. In the second solution, processing resources are being periodically utilised so as to pre-emptively obtain information clusters. In this second case, when an evaluation decision is needed, the required information for each redundant component is already available. The evaluation logic only compares the existing information on the available components so as to identify the optimal redundant variant. Information clusters are maintained up-to-date by periodically analysing new monitored data samples and merging them with the existing information. In the third solution, information on the optimal redundant component in each RG is periodically attained. Consequently, this information is already available when a component evaluation decision is needed. The three solutions are optimised with respect to the time required to take a component evaluation decision. The trade-off lays in the pre-emptive consumption of resources, needed to periodically process available data and infer clustered information. Namely, instead of processing collected data whenever a decision is needed, these solutions analyse available data periodically, maintaining an up-to-date result of the data analysis process.

The choice on which evaluation solution to use depends on a number of factors. One factor is the utilisation pattern of the component evaluation process. Another factor is the rate and

manner in which monitored data influenced to the clustered information. For example, rare changes could occur in the information clusters if collected monitored data mostly reinforced the conclusions already contained in these clusters. At the same time, component evaluation decisions could be required at a much higher rate than the frequency at which clustered information changes occurred. In such a case, the optimal evaluation solution would be to periodically process monitored data and have clustered information readily available for each of the frequent evaluation demands. The times at which monitoring data was processed can be configured in this case at suitably rare intervals. On the contrary, in case monitoring data frequently impacted on information clusters and evaluation decisions were rare, the optimal solution is to process available monitoring data only when needed by an evaluation decision. The component evaluation process is typically triggered by the anomaly detection policies, upon detecting or predicting a performance problem. Nonetheless, the component evaluation function can also be called periodically, for optimisation purposes, provided that sufficient resources were available. Component evaluation results are sent to the adaptation decision module for further processing. If a decision to optimise the application is taken, the corresponding adaptation operations are sent to the component activation module, so as to be implemented into the running application.

3.11.5 Adaptation Decision Policies

AQuA's *adaptation decision* functionality is responsible for finding optimal solutions to detected or predicted performance problems. Solutions consist of application adaptations that involve the dynamic swapping of one or multiple redundant components. Potential optimisation solutions are initially identified by the component evaluation module. More precisely, the evaluation process identifies the optimal redundant components in the targeted execution context. The potential optimisation solutions are sent to the adaptation decision module for further processing. The adaptation decision function analyses the received optimisation proposals and determines whether to actually adapt the running system. For this purpose, the adaptation decision logic takes into consideration the additional factors that may influence the overall outcome of the proposed applications adaptation. This is because even if a redundant component was evaluated as optimal in the current execution context, the final outcome of the system adaptation operation needed to activate the component may not be optimal overall. Other factors, such as the cost of the actual adaptation operation, or the risks taken when dynamically updating the application are also considered at this level. Possible negative effects are compared against the potential benefits that an implemented optimisation solution could bring.

Adaptation decision policies are also responsible for selecting a final optimisation solution, in case multiple possibilities are proposed by the evaluation process. This can happen in case different redundant components are found to be optimal based on different evaluation criteria. For example, different redundant components can be optimal with respect to different quality metrics. Namely, one redundant component may yield higher throughputs, while another component may be more reliable or secure. Also, different redundant components in a RG may be optimal with respect to the various methods provided in the RG's interface (subsection 3.3). In such cases, adaptation decision policies are used to choose the final optimal redundant component to activate. The costs and risks associated with each optimisation so-

lutions can be evaluated at this level. An overall score can be calculated for each potential solution to support the final adaptation decision process. Adaptation policies should be specified so as to resolve any potential conflicts and be able to select a unique optimal redundant component at any time.
The adaptation decision function is also used to prevent undesirable adaptation behaviour, such as reactions to false alarms, infinite adaptation loops, or cascaded optimisations. For example, decision policies can be specified at this level to prevent the application from being adapted too often, or optimised based on monitoring data collected while the system was undergoing a previous adaptation.

3.12 Component Activation

AQuA's *component activation* functionality is used to dynamically adapt managed applications, as dictated by system optimisation decisions. Adaptation decisions are taken whether manually, by human system administrators, or automatically, by AQuA's policy-based adaptation logic (section 3.11). AQuA's component activation operations involve swapping redundant components, while the managed application is running [30], [28] and [33]. The optimisation decisions sent to AQuA's component activation function indicate the redundant components to be activated and deactivated at each point.
As stated in related research on component hot-swapping (e.g., [3] or [77]), two main issues occur when replacing component variants during runtime. One issue is concerned with the state transfer between the swapped component variants. This operation involves porting the state of the currently executing component instance to the replacement component instance. Such state transfer operations are only needed when the same client request (or session) must be handled by different component variants. This situation implies that a client request starts being handled by a certain redundant component and subsequently finishes being handled by a different redundant component. Thus, the handling process of a certain client request sequentially involves multiple redundant components.
The state transferring operation between redundant component variants may particularly benefit software applications in which client requests took significant amounts of time to execute. In such cases for example, the time required for a sub-optimal component to handle a client request may be greater than the time required to transfer the current component state to an optimal redundant component and allow the new component to finish processing the request. State transfer solutions such as proposed in the ongoing research in the area [77], [64] can be considered to benefit such cases. However, the problem domain targeted by the thesis is typically characterised by rather short-lived client requests. That is to say, that in Internet-based enterprise applications client requests usually take of an order of seconds (or in some cases minutes) to execute. In such cases, transferring state between redundant components would bring little performance benefits to requests already being handled when the component replacement occurred. For this reason, the approach adopted for AQuA's component activation function does not involve state transfer operations between redundant component instances. Rather, running component instances finish executing all started client requests before being removed by a component-swapping operation. Hence, a particular client interaction always finishes execution with the component instances it started with.

A second concern when dynamically swapping components is how to maintain the consistency of existing client references. In other words, clients holding a reference to a redundant component in a RG should not break when a component-swapping operation is performed on that RG. On the contrary, the clients' references should be transparently changed so as to point to the currently active redundant component, at all times.

Two main approaches are possible for implementing AQuA's component activation solution. A first approach only allows one single redundant component in a RG to be available in the system at any one time. In this case, when a new redundant component is activated in the RG, it completely replaces the old redundant component. Nonetheless, as no state transfer is performed, old redundant components must finish executing started requests before they can be replaced. For this reason, component-swapping operations must be delayed in this case until all started client interactions have been completed. This situation may consequently cause certain delays in the component activation process. Such delays directly depend on the nature and duration of the existing client sessions. As such, notable delays may be experienced during a component activation process, even though the actual component-swapping operation does not induce significant overheads. The reason is that the new redundant component to be activated cannot be made available before the old redundant component has finished handing current client sessions. Thus, new incoming client requests are blocked waiting for the component activation process to complete. The waiting time directly depends on the type of client interactions with the system, at the time the component activation process is started. Nonetheless, even if this approach may induce noticeable delays, it can be successfully used to implement the component activation function, in the context of Internet-based enterprise systems. An important reason is that the proposed redundancy-based optimisation solution was not devised with the aim of performing frequent component-swapping operations. Applications should only be adapted when major, possibly unpredicted changes occurred in their execution environments. Thus, the management framework should only react to cases in which the application could be significantly optimised, or when it risked failing to meet its quality requirements. AQuA's adaptation logic should always be specified considering these goals. It should not be configured to constantly fine-tune applications to small variations in their execution contexts and obtaining marginal performance benefits. Considering these management goals, it can be stated that induced component-swapping delays should only be experienced on rare occasions, and by a limited number of client sessions.

Nonetheless, a second approach can be adopted, to remedy the described problem and optimise the component activation process. However, this optimisation comes at the expense of a more complex component-swapping implementation. This second component activation solution differs from the previous one in that it allows for instances of different redundant components to coexist, as part of the same RG. In this case, incoming client requests are being directed to instances of the currently active redundant component, upon arrival. A request-indirection mechanism is used for this purpose, dispatching client requests to the appropriate redundant component instances. When the active component is changed, new incoming requests are directed to instances of the new active component. In parallel, instances of the redundant component to be deactivated finish handling existing client sessions before being removed. As a variation of the same approach, all redundant components in a RG can be maintained and made available in the system, at all times. As before, incoming client requests are directed to one of the available redundant components upon arrival.

In the current AQuA implementation, the component activation function is implemented

based on the component hot-deployment facility of the application server. This solution aligns with the first presented approach, where a single redundant component in a RG is allowed to be available in the system, at anyone time. As discussed, this solution implementation can be optimised, by allowing multiple redundant components in a RG to work in parallel for handling different client requests. In certain scenarios, this optimised approach could significantly reduce delays in the component activation process. The optimisation would particularly benefit cases in which client sessions took significant periods to complete. A proxy-based solution can be adopted to implement the optimised approach for AQuA's component activation function. In this solution, all incoming client requests are being intercepted by a proxy, which then dispatches the requests to the targeted component instance. Thus, clients do not in fact hold direct references to the actual components they want to use, but rather to the intercepting component proxies. In this case, when a redundant component is swapped, the local reference that the proxy holds to the old component must be updated so as to point to the new redundant variant. On the contrary, references that external clients hold to the component proxies maintain their validity and thus require not to be changed. Component technologies based on contextual composition frameworks [91] provide a straightforward way of implementing this proxy-based approach. The reason is that in these technologies clients can only call component instances through a component container, in which the targeted component was deployed and run. The component container can consequently be modified, so as to transparently (re)direct client requests to instances of active redundant components. In the context of the EJB component technology, the EJB Object implementation, proprietary to each EJB application server provider, can be modified and used to fulfil the role of component proxies for managed EJB components (sections 2.4.2 and 4.3).
AQuA's modular design allows the component activation implementation to be seamlessly changed, independently from other management functionalities. Nonetheless, providing an optimal implementation of the proposed AQuA framework was out of the thesis' scope. The thesis proposes a management framework that uses component redundancy to automatically optimise component-based systems. Additionally, it aims to provide a proof-of-concept example of how this framework can be implemented and used.

3.13 The Learning Mechanism

AQuA's *learning* capabilities were devised for analysing collected monitoring data and improving the framework's management behaviour over time. The goal was to incrementally automate system management-related tasks and progressively reduce required human interventions. The main learning process currently specified for AQuA is used to automatically analyse raw monitoring data and infer higher-level performance information on the system's behaviour (subsection 3.13.1). Other learning procedures can be envisaged for augmenting AQuA's adaptation logic, based on acquired performance information and the outcomes of previous system adaptations (subsections 3.13.2 and 3.13.3).
AQuA's inference learning mechanism is used to build accurate performance descriptions for the managed redundant components, in the current deployment context. Component descriptions (subsection 3.11.2 are used to store information on the components' performance characteristics. Thus, a component description provides a specification of the component's

performance qualities, or its behaviour in various execution contexts. Based on this information, the framework can successfully decide on how to adapt the application in different running conditions. More precisely, the framework can analyse the description information of the available redundant components and decide which component would be optimal under the current running conditions. AQuA's learning functionality enables the management framework to:

- avoid requiring initial performance descriptions to be supplied at components' deployment time
- avoid completely relying on monitoring results obtained when components were integrated in a different system (e.g. a testing platform)

The goal of the learning mechanism is to automatically acquire performance information on managed components, in the current managed system. This capability is designed to complement, or replace certain tasks that component providers and testers commonly perform at present. The paramount complexity of such tasks may cause the motivation of this approach seem naive. However, the more realistic intent is to start by identifying and automating the most straightforward data analysis and processing tasks and subsequently use the experience to incrementally increase the complexity of automated procedures.
The adopted inference learning strategy was implemented and partially tested.

3.13.1 Inferring Performance Information from Monitoring Data

A learning mechanism is proposed to infer performance information on the managed components based on runtime monitoring data. The goal is to model and automate the process that a human tester would normally perform in order to obtain performance information on a certain system. Thus, the proposed learning process collects the raw data samples provided by the monitoring facility and merges the data of similar samples into *clusters* of information that have a certain emphreliability factor associated with them. These clusters of information represent the cumulated result of extensive monitoring data and are used in the evaluation process to reliably determine optimal system configurations.
Performance information is inferred at the component method level. The information associated with each method consists of a set of inferred data elements where each element belongs to a different cluster. The set of clusters acquired for a component's method characterise the inferred performance behaviour of that component method, in different execution contexts.
A component method's performance behaviour is represented as a collection of information clusters. Each cluster is characterised by several elements as follows (Figure 3.7). First, a cluster contains a *cluster centre*, which basically indicates the execution context associated with this cluster. Then, each cluster has an *admission interval*, which delimits the acceptance of monitored data samples to the cluster. The admittance interval can also be defined based on a *no similarity inteval*, where the admission interval is twice the value of the no similarity interval. The *no similarity interval* represents the maximum difference between the cluster's centre and the execution context of a new monitoring data sample for which the new sample is accepted

into the cluster. A certain *similarity function* is used to compare different execution contexts, in order to decide whether a new data sample should be added or not to a cluster. A single function is used for all clusters to calculate the similarity between data samples and the available clusters and to control data sample admission to clusters. A triangular function was used for this purpose in the current implementation, but trapezoid, bell-curve or gauss shapes can also be employed. Finally, each cluster has an *information element* associated with it, representing the inferred information obtained from merging together all similar data samples accepted into this cluster. The central information element in a cluster represents the inferred performance information of that cluster. It is obtained by repeatedly merging monitoring data samples that are accepted into the cluster based on their similarity with the cluster. The information element provides information on a component method's performance characteristics in a specific execution environment. Thus, the set of available clusters for a component method provides performance information on the method's performance behaviour under multiple targeted execution environments.

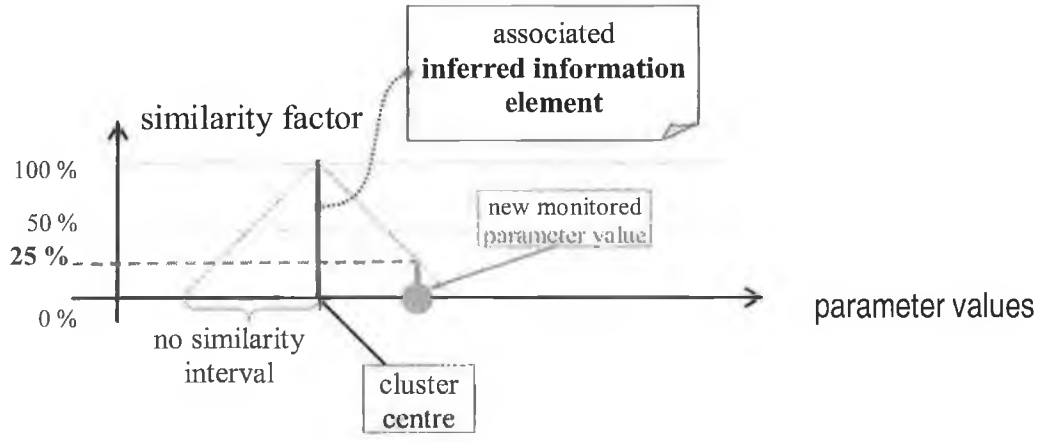

Figure 3.7: clustering monitoring data to infer performance information

Monitoring data samples are being repeatedly collected from the running system, at fixed time intervals. Each monitoring data sample contains the date when it was created and a certain set of monitored parameter values. Monitored parameters include CPU, memory and network bandwidth usage, as well as response time, throughput and incoming workload. Such data is collected separately for each method a RG provides, and for each redundant component available in that RG. This data is stored as part of a redundant component's performance history data. It is also analysed and used to infer higher-level performance information for that component. Thus, data samples are stored in two different formats. First, monitored samples are stored as raw monitoring data, as collected from the running system. Secondly, they are stored into clusters of inferred performance information, obtained by analysing and processing the raw monitoring data. Information inferred during runtime is used to verify the validity of initial or current information. Thus, non-accurate information can be detected and modified in effect. The adaptation logic for taking optimisation decisions favours the usage of inferred information, if available, rather than the repeated analysis of unprocessed monitoring data. The inference learning process is responsible for regularly performing this task instead. This approach can reduce overheads and improve the efficiency of the system adaptation pro-

cess. The methodology used to infer performance information from raw monitoring data is described as follows.

Monitored data samples are grouped into clusters, based on an *overall similarity factor (o_SimF)* calculated between them. The idea is that it only makes sense to group and merge data samples that were obtained in similar execution environments. For this purpose, the o_SimF between two data samples is used to represent the degree to which these two samples can be compared and merged to infer reliable performance information. The o_SimF takes values between 0% and 100%, where 0% indicates no similarity at all and 100% indicates complete similarity. The o_SimF between two data samples is determined as follows. First, for each considered parameter, the values of that parameter in the two samples are being compared. A *parameter similarity factor (p_SimF)* is calculated for each such parameter. Thus, there will be an individual memory usage p_SimF, a workload p_SimF and so on, for each monitored parameter. The p_SimF of a parameter is calculated using: $p\_SimF(p_{i1}, p_{i2}) = (|p_{i1} - p_{i2}| * 100)/no\_sim\_int$, where p_{i1} and p_{i2} are the values of the parameter p_i in the two compared samples s_1 and s_2 respectively; $p_i \in$ P, where P is the set of considered, or monitored parameters contained in each sample and $i = \overline{1,n}$; n is the number of parameters considered; the no_sim_int is the maximum difference between two values that have some degree of similarity; if the difference between two values is greater than the no_sim_int, then the p_SimF of the two values is 0%. The current function selected to calculate the p_SimF is a triangular one (Figures 3.7 and 3.8), but other functions can be used instead, as appropriate (e.g. trapezoid, or bell curves).

As a next step, the o_SimF of the two samples is calculated based on the individual p_SimF values set. The *minimum* function was selected for this purpose: $o\_SimF(s_1, s_2) = \min(p\_SimF(p_i))$, where: s_1 and s_2 are the two compared data samples: s_1 is a new data sample and s_2 is the inferred performance information in an existing cluster. Using the *minimum* function to calculate the overall sample similarity means that if one parameter in the data samples has a 0% p_SimF, then the o_SimF of the samples will also be 0%; in this case the two data samples will not be merged as part of the same cluster. If necessary, other functions can be used to calculate the o_SimF from the individual p_SimFs.

Each cluster contains one inferred performance element, which is obtained by merging all similar data samples collected up to that point. This performance element has the same format as a raw monitoring data sample. Nonetheless, the inferred information has a higher reliability factor than a single monitored data sample would. The more data samples are used to infer a performance element, the higher the reliability factor associated with that element.

A set of performance information clusters is built for each method of each redundant component, as follows. Whenever a new monitored data sample is received for a certain RG method, it is used to update the existing set of inferred clusters of that method, for the currently active redundant component. First, the o_SimF is calculated for the new data sample with respect to all the existing clusters. Second, the new data sample is used to update the inferred information in those clusters for which the calculated o_SimF is greater than 0%. If the new data sample cannot be used to update any of the existing clusters, because all calculated o_SimFs are 0%, then a new cluster is created for the new data sample. In addition, if all the o_SimFs that are greater than 0% are also smaller than a certain threshold (e.g. 50%), then the identified similar clusters are updated as before and a new cluster is also created for the new data cluster. This situation is exemplified in Figure 3.8. The manner in which new data samples are used to update the existing information of similar clusters is described next.

New data samples are used to update the existing information of similar clusters. The value of

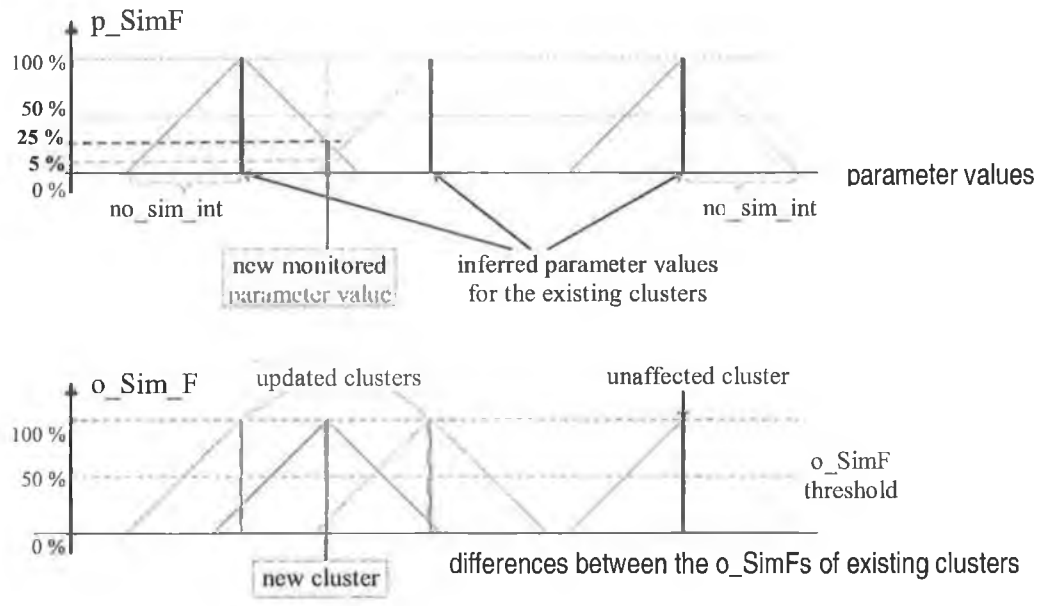

Figure 3.8: the inference learning process:
updating current performance information with new monitoring data

each parameter in a new data sample is used to update the value of the corresponding parameter in the existing information element of the updated cluster. The following formula is used for this purpose: $updated\_parameter\_value = (old\_value + w * new\_value)/(1 + w)$, where w is a weight factor, taking values between 0 and 1: $w = o\_SimF/100$. This formula dictates that a new data sample influences the existing inferred data in a cluster in a manner that is directly dependent on the o_SimF between the new data and the cluster. As such, new data will hardly influence existent data that was monitored in dissimilar environmental conditions. If the new data sample has a small p_SimF for even one parameter when compared with a cluster, then this sample will only have a small influence in updating the values of that cluster, even if the rest of its parameter values are extremely similar. However, new monitored data will have a significant influence on existing data that was monitored in similar or identical environmental conditions.

Figure 3.9 shows how the value of w influences the way new data sample values influence existent inferred data values. For higher values (closer to 1), an old value converges to the new value in only a few iterations. Each iteration corresponds to a new data sample becoming available and being used to update inferred data. For lower weight values (closer to 0), it takes a much higher number of iterations for an old value to slowly converge towards the new value. This means that a new monitored data sample only has a small or insignificant influence on the inferred data of a cluster if the similarity factor between the new data and the existent cluster data was small. In other words, new monitored data will hardly influence existent data that was monitored in dissimilar environmental conditions. However, new monitored data will have a significant influence on existing data that was monitored in similar or identical environmental conditions.

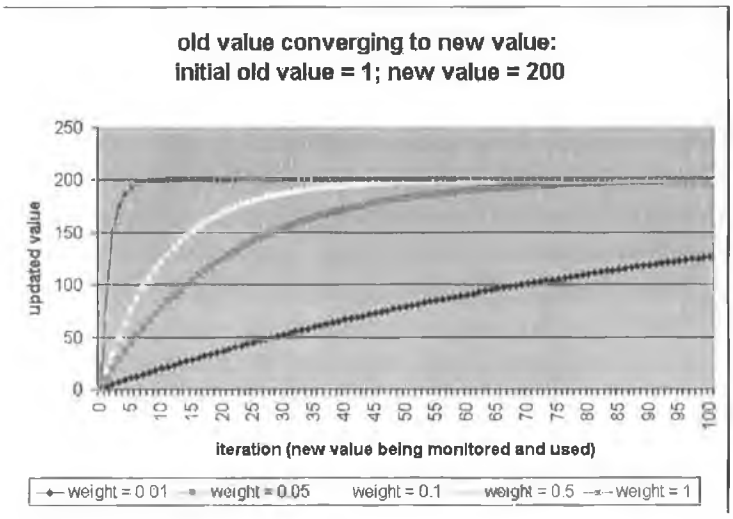

Figure 3.9: convergence trends of inferred values with new values, depending on the calculated weights (w)

Alternatively, the process of inferring performance information from monitoring data can be triggered periodically, or only upon request, rather than with each new available data sample. This would decrease the impact on performance the learning process would have if executed more often, for each newly collected sample. Both options were integrated in the current framework prototype and AQuA_J2EE can be configured on which one to use at start-up.

3.13.2 Using Performance Information for Component Evaluation Decisions

The component evaluation process uses the available performance information to determine the optimal redundant components in specific execution contexts. Performance information is available from the inference clusters associated with each component method. The component evaluation process accesses and uses this information as follows.

When the evaluation process is triggered, a monitoring data sample is first collected from the current execution environment. This sample is compared with all the existing clusters, of all the available redundant components, and the o_SimFs are calculated as previously discussed. Clusters with o_SimFs lower than a certain predefined threshold (e.g. 20%) are considered irrelevant and are being discarded. If no relevant clusters are found, for any of the available redundant components, it means that no performance information exists in the RG for the current execution conditions. An adaptation is only triggered in this case if considered that increased risks can be taken.

In cases in which more than one relevant cluster is found for the same redundant component, reoccurrence probabilities are calculated for each cluster, based on their associated reliability values and calculated o_SimFs. Higher probabilities are associated with clusters that have high reliability factors. For example, the evaluation process will predict that, for a certain redundant component, performance characteristics similar to the ones recorded when 'spikes' were monitored in a previous execution context will have a small probability of reoccurring in

a similar context. On the contrary, there will be a high probability for the performance characteristics monitored in 'normal' conditions to reoccur.
In the simplest implementation case, a single cluster is selected to represent each redundant component (in a certain environmental context). From these representative clusters, the ones with information that indicates possible availability problems (e.g. thrown exceptions) are being discarded. The performance parameter values of the remaining clusters are being compared (i.e. response times, throughput and resource usage) and an 'optimum' cluster is then selected. The selection is based on comparing the parameter values in the considered clusters, and then using whether an overall evaluation of all parameters, or taking into account specified priorities among parameters. The redundant component associated with the selected 'optimum' cluster is considered to be the currently 'optimal' component by the evaluation process.
A different approach is possible for implementing the evaluation process, for determining which redundant component is optimal under a certain execution context. This involves evaluating redundant components periodically, rather than only upon request. This means that the optimal redundant component for each monitored execution context is determined periodically. Optimal redundant components can be re-evaluated with each newly collected monitoring data sample. However, this might induce increased and possibly unnecessary performance overheads. At the other extreme, the alternative presented in the previous paragraphs was to only evaluate redundant components upon request. This is done by comparing the performance information in the component descriptions available, for the current execution context. In both cases, the redundant component determined to be optimal for the current execution context is selected as a potential candidate for activation. The identity of the optimal redundant component determined is sent to the adaptation decision module for further processing.
The main difference between the two approaches is in the timing for establishing the optimal redundant component. In one case, the optimal component for the current context is determined upon request. Component descriptions are used in this case to compare the performance characteristics of available redundant components. In this case, the current execution context is matched against the execution contexts available in the component descriptions. If inferred performance information is available for the current execution context for more than one redundant component in the targeted RG, then the redundant component with optimal performance characteristics is selected. In the second case, the comparison between the performance characteristics of redundant components in a RG is performed periodically, for each execution context that has been monitored. Thus, in this approach, optimal redundant components are already determined at the time a request arrives for the RG's evaluation. As such, when an evaluation request arrives, the evaluation process merely needs to match the current execution context against the stored execution contexts for which inferred data and the associated optimal redundant component are available. As before, the redundant component indicated as optimal for the matching execution context is returned by the evaluation process. In case the evaluation process is executed periodically, method-level performance data in the component descriptions is used to infer performance information at the RG level. Performance information at the RG level is represented and stored as RG descriptions. RG descriptions are organised in a similar manner with component descriptions. This means that there is one set of clusters for each RG method. The way RG cluster sets are created in this case is described as follows. Existing clusters from the available redundant components are compared

and merged into a new set of RG-level clusters. One cluster set is formed for each method in the RG interface. Each set of clusters for a RG method represents the performance behaviour of that RG, in different execution contexts, when the optimal redundant component is used in each context. A different redundant component can be assigned to each cluster in a RG cluster set, as different redundant components can be optimal in different execution contexts. Method-level clusters from different redundant components are only compared if the similarity factor between them is higher than zero. The new set of RG-level clusters contains data on the performance behaviour of the optimal redundant components available, in certain environmental conditions. Subsequently, when handling performance alerts, or optimising the system performance, inferred data at the RG-level is used for determining the optimal redundant components in the current environmental conditions.
In short, if the evaluation process is executed periodically, the optimal redundant components for all execution contexts are repeatedly determined based on the collected monitoring data and will be readily available when application optimisation solutions are needed to overcome detected performance anomalies. In this case, when the evaluation process is required to return the optimal redundant component, the current execution context is matched against the execution contexts for which inferred information is available. When a match is found, the optimal redundant component associated with that context is returned.

3.13.3 Learning Procedures for Adaptation Logic

Special-purpose learning procedures can be devised for automatically augmenting AQuA's adaptation logic. Such learning procedures would be devised so as to analyse newly acquired system information, such as inferred component performance descriptions, as well as the results of past adaptation decisions. The different types of decision policies, namely anomaly detection, component evaluation and adaptation decision, can be updated and tuned in effect. Learning procedures that can modify the adaptation logic would allow AQuA to not only recognise and act on a given set of known scenarios, but also react to novel situations and to solve performance problems not previously encountered.
Learning procedures can be employed to automatically infer and configure anomaly detection policies. This can be achieved by correlating performance problem incidents, with the monitored system states that preceded the problem occurrence. The incidence of similar conditions can be subsequently used to predict the same performance anomalies in the future. Based on such observations, anomaly detection policies can be created and configured so as to recognise known problematic conditions by analysing monitoring data and to signal the predicted problems to the component evaluation function.
Different learning procedures can also be devised to automatically configure and create component evaluation rules. Such learning procedures would be based on the results of the inference learning process. The inference learning process synthesizes component performance information from runtime monitoring data. Obtained performance information is further analysed in order to detect potential cross points in the redundant components' performance curves. Cross points represent execution contexts in which the optimal redundant component in a certain RG changes. In the presented testing scenarios (chapter 5), the learning process for performance cross point detection was manually performed. The procedure involved analysing available performance information and detecting cross points with respect to the

various performance metrics considered (e.g. response times, or memory consumption). This process can be automated as part of AQuA's learning facility. An automated learning procedure can be implemented to analyse the available component information and detect performance cross points. This is achieved by comparing the performance values yielded by the considered redundant components. Clearly, only redundant components of the same RG are compared. Also, it is only correct to compare performance values measured in identical or similar execution contexts. The procedure for determining an optimal component based on available performance information is described in the previous subsection. Results obtained from this procedure can be formally stored as new decision policies. In this manner, when a component evaluation is required, one of these inferred decision policies can directly indicate the optimal redundant component in the targeted execution context. This would save the time required to determine the optimal component upon each request. The inferred component evaluation decision policy would contain the information on the optimal component at all times. Inferred evaluation policies can be continually updated, as new performance information becomes available. More precisely, the threshold values for swapping optimal redundant components can change, so as to become more accurate over time.

In conclusion, automated learning processes can be used to infer new component evaluation policies and configure existing policies. This process identifies situations in which one redundant component is optimal with respect to a certain performance metric in certain execution contexts, and another redundant component is optimal with respect to the same metric in other execution contexts. An evaluation policy can be inferred for this situation to indicate the respective optimal component in each execution context.

Learning procedures can also be implemented to automatically create and configure adaptation decision policies. For example, the values of the risk factors taken when deciding to adapt the system can be automatically tuned based on monitored results from previous adaptations. Supposing an application is being adapted based on performance information that has a certain reliability factor associated with it. In case the monitored adaptation result significantly differs from the predicted result the adaptation decisions can be accordingly updated to increase the reliability requirements for valid information. This would mean that the adaptation policies will take fewer risks in the future, when deciding to optimise the application.

CHAPTER

FOUR

Framework Implementation - Prototype for J2EE

Chapter Summary

This chapter presents AQuA_J2EE, a prototype implementation for the AQuA framework. AQuA_J2EE was implemented as part of the thesis research work, to show how an instantiation of the AQuA framework can automatically manage a component-based application. J2EE was selected as the component technology and the JBoss application server as the J2EE implementation used.

The chapter explains how the main framework functionalities were implemented, including the runtime monitoring, adaptation logic and component activation modules. An intrusive, server-level instrumentation approach was taken for implementing AQuA's monitoring function. Decision policies were specified using the ABLE Rule Language for implementing AQuA's adaptation logic. The decision policies currently specified for AQuA_J2EE were designed and configured for managing the particular J2EE example application tested. Policies were specified for the anomaly detection, component evaluation and adaptation decision parts of the AQuA framework. The component activation function was implemented based on the hot-deployment facility of the underlying application server. An introduction to the JBoss application server characteristics relevant to AQuA_J2EE precedes the framework implementation description.

Goals of this chapter:

- AQuA_J2EE: a prototype implementation for the AQuA management framework, targeted at the J2EE component technology and the JBoss application server

4.1 JBoss J2EE Application Server

4.1.1 JBoss Overview

JBoss[1] is an open source application server for J2EE, exclusively implemented in the Java programming language. The JBoss project was started in 1999 as an open-source EJB container. As the project progressed and new versions of the JBoss server were implemented, new functionalities and services were incrementally provided. In addition, novel technologies such as J2EE, JMX[2], SOA[3], JAAS[4], or AOP[5] were adopted and supported by the JBoss server.
JBoss has been quite largely adopted due to some of its particular characteristics. These include the fact that it is free, open source and simple to use when compared to other J2EE servers. In addition, as a result of its modular structure, JBoss is also easily modifiable and configurable. The increased popularity of the JBoss server is reflected in over 5 million downloads, making it the most downloaded J2EE application server. The latest JBoss release is version 4.0, which uses AOP to add support for different types of java components, in addition to J2EE.
The following subsections present the JBoss server, starting with the general architecture and then focusing on the EJB support provided via EJB containers. The main characteristics of the JBoss architecture, such as modularity and loose-coupling are also discussed, showing how they enable the seamless customisation of the server. The way JBoss can be modified and configured to optimise application performance in various scenarios is also discussed.

4.1.2 JBoss Architecture

JBoss provides a full J2EE implementation. It includes the JBoss Server and the EJB Container, and it is based on a JMX infrastructure. It also contains JBossMQ - for JMS messaging, JBossTX - for Java Transaction API and the Java Transaction Service (JTA/JTS) transactions, JBossCMP - for CMP persistence, JBossSX - for JAAS-based security and JBossCX - for Java Connection Architecture (JCA) connectivity. JBoss features a modular architecture, in which the provided services - transactions, persistence, security, or connectivity - are implemented as separate modules, which are integrated through a JMX core. This architecture allows any JBoss module to be replaced by a custom implementation module as long as the customised module is JMX-compliant and features the same Application Programming Interfaces (APIs) as the original one. The JMX technology enables seamless integration and interaction of JBoss modules. As such, the JBoss standard elements can be extended or replaced with customised versions in order to accommodate the needs of various infrastructures.

[1]JBoss open-source J2EE application server: www.jboss.org
[2]Java Management Extensions (JMX) technology, from Sun Microsystems: java.sun.com/products/JavaManagement
[3]Service Oriented Architectures (SOA): www.service-architecture.com
[4]Java Authentication and Authorization Service (JAAS), Security: java.sun.com/products/jaas
[5]Aspect Oriented Programming (AOP): www2.parc.com/csl/projects/aop

4.1.3 EJBs on JBoss - the EJB Container

EJB Containers

An EJB *container* is a software entity that manages a certain type of EJB. When an EJB module or application, such as an EJB.jar archive is deployed, JBoss creates a number of containers and connects them internally to be able to handle references between the deployed beans. Detailed EJB information is needed for deploying a certain EJB into a container. An .xml deployment descriptor file, named *ejb-jar.xml*, is used to provide this bean information (metadata). JBoss creates one separate container instance for each separate EJB deployed. Most settings of EJB containers can also be configured via deployment descriptor files (e.g. subsection 5.2.3 and appendix B). The EJB container architecture and the various configuration options in JBoss are described below.

EJB Container Configuration

JBoss externalises most of its EJB container settings using container configuration files in XML format. These are the *standardjboss.xml* file and the *jboss.xml* file. The standardjboss.xml is the default configuration file and it applies to all instantiated containers. The jboss.xml configuration file is specific to each deployed EJB application and comes bundled with the application archive files. Container configuration files have to conform to a certain JBoss specific format, which indicates the container elements that can be configured, their names, valid attributes and default values. The main configurable container elements in JBoss include the composition of the container interceptor chain, instance pool and instance cache settings, persistence managers, commit options and security configurations. This allows JBoss containers to be customised for optimal behaviour and performance of particular deployed applications. In addition, special-purpose interceptors with functionalities that are not provided with the standard JBoss distribution can be implemented and integrated as part of customised JBoss containers. EJB deployment descriptors in JBoss allow a particular container configuration to be used for each deployed EJB component. In case a customised container configuration is not specified for a certain EJB, a standard container configuration is selected for that EJB by default. Default container configurations are specified in the standardjboss.xml file. The default configuration selected for a certain EJB depends on the EJB's type: Stateless Session, Statefull Session, BMP or CMP Entity, or Message-Driven bean. Appendix B illustrates a more detailed example of a jboss.xml file. The example shows how JBoss containers were customised for part of the performed experimental work.

EJB Container Instantiation

When an EJB application is deployed on the JBoss server, an EJB container is instantiated for each EJB component in the deployed application. When a container instance is created, all its specific configuration attributes are read from the corresponding container configuration files: the generic standardjboss.xml file and the application-specific jboss.xml file. Interceptors are added accordingly to the instantiated container.

In JBoss, EJB applications can be deployed manually or by using the automatic deployment facility that JBoss provides. Automatic deployment allows applications to be deployed by simply copying the application directories or archives in a certain directory location. The JBoss server automatically detects the application files, whether at server start-up or during

runtime and performs the necessary deployment tasks. This facility allows applications to be *hot-deployed*, which means applications can be deployed while the JBoss server is running.

EJB Container Plug-in Framework

JBoss EJB containers are designed as frameworks into which various functional parts can be plugged-in. Consequently, to a large extent, the behaviour of a JBoss container is implemented and provided by container *plug-ins*. The main functionality of the container itself is to manage its plug-ins, connecting them and providing them with the information they need to perform their implemented functions.

JBoss provides several predefined container types, each one designed for managing a different type of EJB component. Separate EJB containers are provided for managing CMP and BMP Entity beans, Stateless Session beans, Statefull Session beans, or Message-Driven beans. For example, Stateless Session containers are specially designed for managing Stateless Session beans. Consequently, they do not use instance caches, as Stateless Session beans do not maintain state between client calls. An additional example is Entity containers for managing Entity beans that use a persistence manager for making the Entity bean's state persistent based on a certain storage mechanisms, such as a relational database.

JBoss uses various container plug-in types, including invokers, interceptors, instance pools and cashes, or bean persistence managers. Each plug-in can be included, customised or excluded from a container configuration, by using the corresponding container configuration file(s). Invoker plugins are used for supporting distributed client-server communication over various transport and network protocols. Interceptors, instance pools, caches and persistence managers are briefly discussed below.

Interceptors

Interceptor plugins are used to capture incoming method calls and perform certain tasks before forwarding them to the actual EJB instances. Such tasks may include logging, security checks, or transaction processing. Interceptors are linked in a chain-like structure, such that all incoming and outgoing method invocations must pass through this container interceptor chain. Namely, incoming method calls are forwarded from the first interceptor down the container plugin chain until they are being dispatched to the targeted EJB instance. The last interceptor in the chain forwards method requests to targeted EJB instances. The order of interceptors in the linked chain is important. For example, security interceptors should always be placed before EJB instance-acquisition interceptors.

JBoss' plugin chain design clearly separates the functionalities of the different interceptors. This increases container flexibility by allowing various interceptors to be added, modified or removed, as well as rearranged in the interceptor chain.

Instance pools

Instance pools are used to manage EJB instances that are not associated with any identity. Containers can be configured to use an instance pool for recycling EJB instances, or configured to only instantiate and initialise EJB instances on demand. Instance pool policies and capacity can be configured in the container configuration files. Instance caches use instance pools to obtain free EJB instances for activation. Certain interceptors use the instance pools to obtain stateless EJB instances and return them in response to 'create' requests on EJB home interfaces.

Instance caches

Instance caches are used to manage 'active' EJB instances, which are EJB instances that have identities associated with them. Only Entity and Statefull Session beans are cached, as these are the only EJB types that maintain state between client calls. Instance caches maintain a list of active instances and perform instance activation and passivation operations.

Persistence managers

Persistence managers are used to activate and passivate statefull EJB instances and to manage their states. When a statefull EJB instance is activated, its state is retrieved from the persistent storage used and is associated with a free EJB instance. During EJB passivation, EJBs' states are persisted and the EJB instances are freed and returned to the instance pools. Relational databases or files can be used as persistent storage mechanisms. Different persistence manager types are used depending on the managed EJB type (i.e. whether the managed EJB type is a Statefull Session or Entity bean).

4.2 Framework Implementation Overview

A fully-automated AQuA prototype was implemented in order to test and show the framework's management capabilities and potential. The prototype is referred to as AQuA_J2EE, and it was implemented and tested for managing the performance of J2EE systems at the EJB component level [33].

The JBoss application server[6] was selected as the J2EE application server to integrate with AQuA_J2EE. However, the way AQuA_J2EE was designed allows it to be seamlessly modified to work on any J2EE-compliant application server. This aspect is described in more detail later in this section.

In the context of the EJB technology, AQuA_J2EE uses JNDI names to identify each EJB component it has to manage. This allows each EJB component to be uniquely identified, even when the EJB's internal implementation changes. Using the component redundancy terminology introduced in section 3.4, each Redundancy Group (RG) represents one EJB component. Each RG is uniquely identified based on the EJB component's JNDI name. The EJB's public interface is the RG's provided interface. In this context, activating different redundant components in a RG is equivalent to changing the implementation of the corresponding EJB component (i.e. the EJB class binary file). In case multiple EJB classes are used together to provide the functionality of a single RG, the public interface of the entry-level EJB represents the RG's provided interface. Multiple EJB implementations, providing an identical EJB public interface, can be available as part of a certain RG. These multiple EJB implementations constitute the redundant component variants of that RG.

Regarding AQuA_J2EE's portability, certain AQuA functionalities, such as the performance anomaly detection, component evaluation and adaptation decision, can be implemented independently of the targeted J2EE platform, or component technology used. Conversely, other framework functionalities, such as the monitoring and the component activation, need

[6]JBoss J2EE application server: www.jboss.org

to interact with the managed component-based system, in order to obtain monitoring data and perform component-swapping operations, respectively. These are system-dependent functionalities that need to be customised for each targeted platform. Figure 4.1 provides a high-level view of AQuA's logical architecture and indicates AQuA's platform-dependent and platform-independent functionalities. Several approaches exist for implementing AQuA's system-dependent functionalities, namely the monitoring and component-swapping related functions [28] (section 2.6.5). For the current AQuA_J2EE prototype, the adopted implementation approach is based on modifying the application server on which managed applications are deployed and run. JBoss was selected as the J2EE application server for prototype implementation. Server-independent solutions can also be implemented for AQuA, based on application-level instrumentation proxies, as discussed in section 2.6.5. In such cases for example, the COMPAS monitoring tool[7] can be used to extract runtime performance data, and a proxy-based component swapping solution can be implemented for adapting the application (appendix A.1).

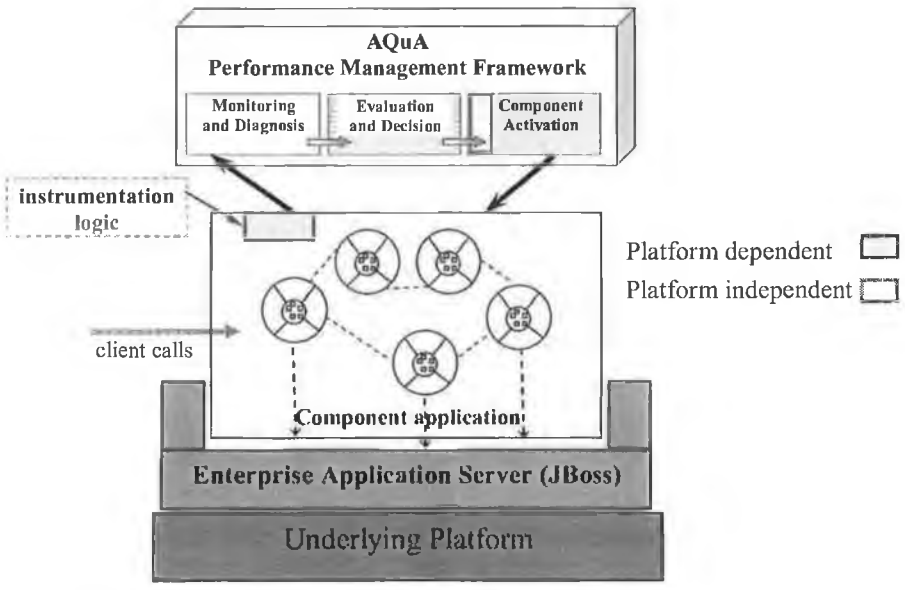

Figure 4.1: AQuA's platform-dependent and platform-independent parts

Regarding the framework's control management architecture, a decentralised approach (section 2.6.4) was adopted for the initial implementation of AQuA_J2EE (Figure 2.5). This implies that adaptation logic and management operations are performed locally at the individual component level, independently from other components. As discussed in section 2.6.4, it is crucially important to additionally provide global management operations, at the overall system level. Two main approaches can be adopted for achieving this goal. First, global system optimisations can be achieved by enabling the decentralised, local framework instances to communicate with each other and influence their actions (Figure 4.2). In this case, the individual framework instances and their inter-communication protocol are designed in such a way that the emerging, global management behaviour directs the managed system towards

[7]COMPAS monitoring and analysis tool for J2EE systems: http://compas.sourceforge.net

meeting its overall quality goals. As an alternative solution, a centralised framework instance can be used to supervise all local decentralised framework instances, so as to form a hierarchical control topology (Figure 2.6). Both approaches allow local optimisations to be performed when possible, while supporting global optimisations when necessary. Nonetheless, higher-level control functions were not required for the performed experimental work (chapter 5) and were not implemented for the current version of AQuA_J2EE.

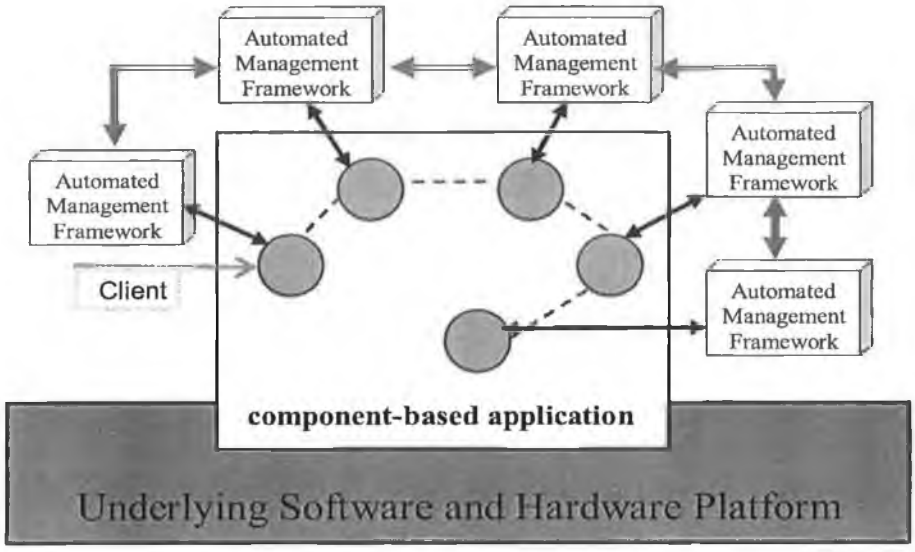

Figure 4.2: connected decentralised control topology

The current implementation of AQuA_J2EE automatically detects changes in the managed application's execution environment, such as the incoming workloads, decides on how to adapt the application, and performs the corresponding EJB component-swapping operations while the application is running [33]. In addition, monitoring data is analysed by the framework's learning mechanism, which infers higher-level performance information on the managed redundant components.
Obtained test results showed that system performance and availability were visibly improved when AQuA_J2EE was used to adapt applications, compared to the case when no adaptation was used. Experimental results from testing AQuA_J2EE in various scenarios are presented in chapter 5. This chapter describes the way the general framework capabilities were implemented in the AQuA_J2EE prototype. The rationale behind the choices made during the design and implementation of AQuA_J2EE are also explained.

4.3 Portability Considerations for AQuA_J2EE

The current AQuA_J2EE prototype was implemented to work on a modified version of the JBoss J2EE application server. The AQuA features that need to be implemented in a platform-dependent manner are part of the monitoring and component activation modules. Both of these functions need to be able to intercept communication messages between the clients and

the managed component instances. The monitoring module needs this capability for extracting runtime data, whereas the component activation must be able to delay and indirect client requests as part of its component-swapping operations. In the modified JBoss version, these requirements were met by instrumenting JBoss's container interceptors and using a modified version of JBoss's hot-deployment facility. A different implementation solution would be needed for porting AQuA_J2EE to other application servers. The reason is that J2EE servers from different providers feature dissimilar designs and implementation characteristics. Therefore, separate customised solutions need to be accordingly implemented for each targeted application server to support AQuA's platform-dependent functions. Nonetheless, the general design principle used for implementing AQuA_J2EE on JBoss remains unchanged for all J2EE-compliant servers. More precisely, the solution designed for AQuA_J2EE is based on using the EJB Home and EJB Object entities that any J2EE-compliant server must implement (subsection 2.4.2). These entities are used for intercepting, monitoring and controlling client access to managed EJB instances. As explained in section 2.4.2, EJB clients can never access EJB instances directly. Conforming to the J2EE specification, all client accesses to instances of an EJB class must be intermediated by the EJB Home and EJB Object associated with that EJB class. The actual implementation of the EJB Home and Object entities largely depends on the specific solution adopted by each J2EE server provider. For example, in JBoss, the EJB Home and Object elements are implemented as part of the EJB container itself. Part of this solution, each deployed EJB has its own EJB container instance associated with it. Instrumenting EJB containers in this case consequently provides EJB-level instrumentation for the JBoss server. However, this solution would not apply to other J2EE server implementations, such as JOnAS, WebSphere or WebLogic. In JOnAS[8] for example, all EJB classes deployed as part of an application archive are managed by a single EJB container instance. Therefore, EJB-level instrumentation cannot be obtained in this case by instrumenting EJB containers. Nonetheless, any J2EE-compliant server must provide an implementation of the EJB Home and Object elements. Conforming to the J2EE specification one EJB Home and Object instances must be available for instances of each deployed EJB class. Therefore, a general solution based on modifying these compulsory elements is always applicable and portable to any J2EE server. Figure 4.3 shows how component redundancy is implemented in the EJB technology. The figure shows how redundancy implemented at the EJB class level or the EJB container level is masked from external clients via the EJB Home and EJB Object entities.

The AQuA framework can also be implemented to manage J2EE applications at different component granularities, or to manage applications based on component technologies other than J2EE. Namely, AQuA can be adapted to manage other J2EE components, such as middleware services, or coarser-grained components such as entire web servers or application servers. Existing component technologies generally provide a level of indirection between clients and the component instances they access. For example, in FRACTAL[9], this level of indirection is provided by a management membrane that surrounds a component's business-logic implementation. As such, FRACTAL components consist of an outside membrane and of one or more internal component implementations. Membranes are used for intercepting, monitoring and controlling access to the managed component content inside each membrane. As such, membranes provide a good opportunity for implementing additional management functions

[8]JOnAS J2EE application server: jonas.objectweb.org

[9]Fractal Project: fractal.objectweb.org

needed for AQuA's platform-dependent parts.

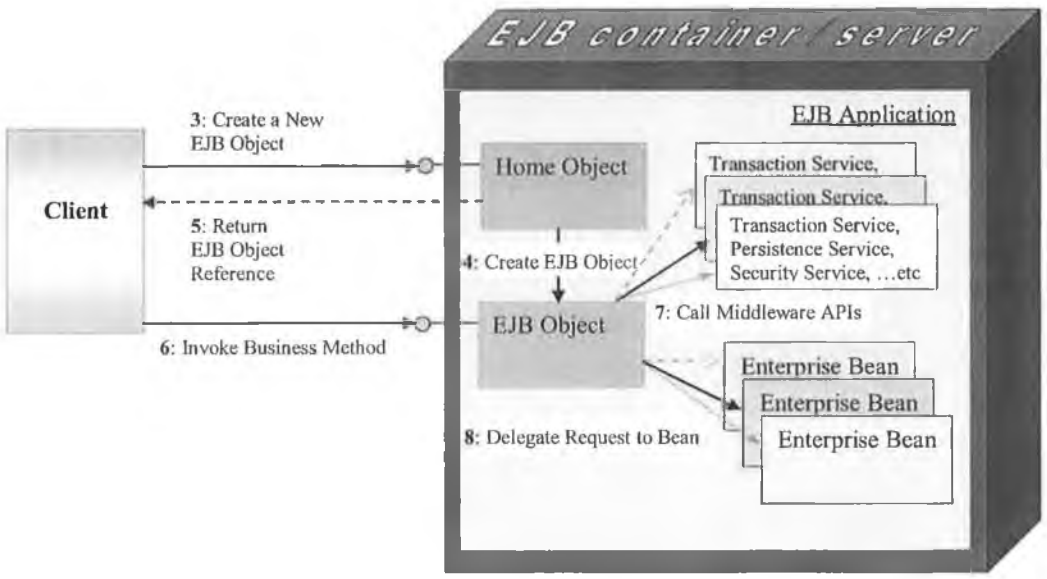

Figure 4.3: EJB remote access with component redundancy

4.4 System Monitoring

The role of AQuA's monitoring functionality is to extract runtime data from the managed application and its execution environment. AQuA analyses collected monitoring data in order to take adaptation decisions and update component description information (sections 3.11.2 and 3.13).

In AQuA_J2EE, the monitoring functionality was implemented by instrumenting the JBoss application server, on which managed applications were deployed and run. The approach is based on a J2EE specification constraint, which stipulates that all client requests to instances of an EJB must go through the application server container that manages that EJB (section 2.4.2). Thus, containers can be modified to intercept such requests and extract specific information of interest. Based on these considerations, one of JBoss' container interceptors (section 4.1.3), the LogInterceptor, was instrumented to extract monitoring data from all incoming and outgoing EJB method calls. Using this approach, monitoring data is collected at the EJB method level. Primitive monitoring data includes method request and response time stamps, the identity of the caller and called EJBs and the name of the initiating and targeted methods. This data is used to calculate EJB method response times, workloads and throughputs, at the EJB server side. It can also be used to determine method call paths through the J2EE application, as discussed in subsequent paragraphs.

A call-path consists of a sequenced set of EJB identities and methods, called as part of a certain business transaction. For example, a business transaction in an online banking system could consist of operations for listing all banking transactions on a client's account over a certain period. The call-path for this example business transaction can consist of a succession of requests for the client log in, client accounts listing, banking transactions listing (for a selected account

and period) and client log out methods.
Additional data is collected at the EJB client side, in order to provide better support for determining accurate application call-paths. Namely, JBoss Client Containers were also instrumented, in order to extract monitoring data on outgoing client requests (Figure 4.4). Whenever a method call is made, the identities of the initiating and targeted methods and of the initiating and targeted EJBs are extracted from the intercepted call at the client side. This information should be identical to the one obtained at the server side from the instrumented EJB containers. Comparing and matching client-side and server-side call-path data guarantees the correctness of created application call-paths. In addition, duplicated information would be necessary for cases when the client and server components were running in separate JVMs. Method request and method response time stamps are also collected at the client container level, allowing performance metrics such as response times to also be calculated at the client side. Obtained call-path information can be used to dynamically create accurate models of the running managed application [28]. This approach contributes to achieving AQuA's goal of placing no extra requirements on application assemblers to provide such application models at deployment time.
The presented monitoring approach is able to extract the necessary data for constructing accurate system call-paths or application models. Nonetheless, more work is needed in order to analyse and correlate extracted monitoring samples and to obtain meaningful call-path or model information. However, creating accurate application models based on runtime monitoring data is a complex problem, which is out of the scope of the presented dissertation.

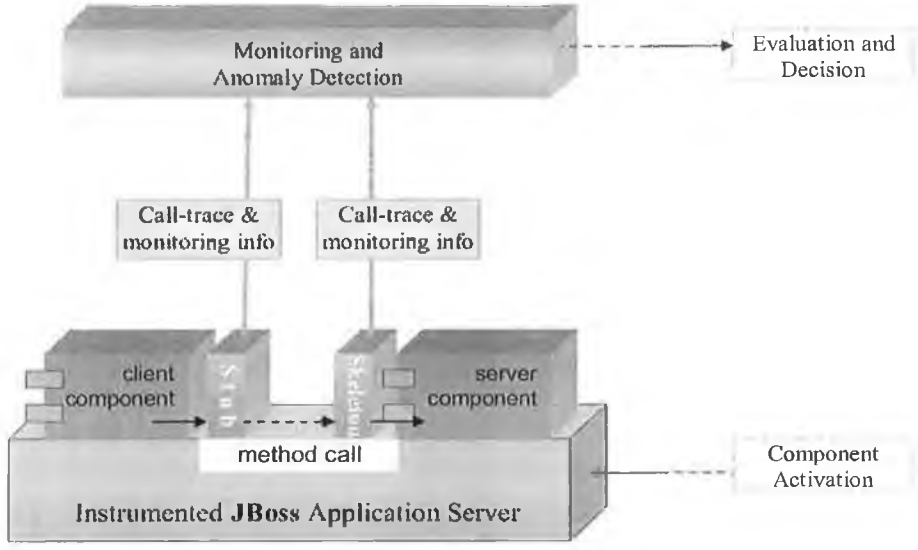

Figure 4.4: instrumenting the JBoss application server

The adopted instrumentation approach has certain advantages over a non-intrusive, proxy-based approach since data can be obtained from the application server more accurately than from application-level proxies. As an example, an important advantage of the server-level instrumentation approach over the proxy-based interaction recorder presented in [28] is the capability of deterministically identifying the initiator of a certain method call. Using the server-level instrumentation, this information can accurately be obtained even in the presence

of multiple, simultaneous calls for the considered method [28]. In addition, in the presented approach information is obtained dynamically while calls are being made, rather than during special testing scenarios. Dynamically obtained data can be used to detect runtime changes in the application components' interaction. Such dynamic changes can commonly occur in EJB applications, as EJBs are allowed to bind to each other at runtime. The main disadvantage of the adopted server-level instrumentation approach is that it is intrusive, since the server container needs to be separately modified for each particular server used. Therefore, a separate instrumentation implementation is needed for each particular application server considered.
Modified interceptors can also be used to capture and signal exceptions thrown during system execution. This data is needed for detecting and avoiding certain types of functional application problems. For example, redundant components that were detected to raise runtime exceptions in the past should not be activated in similar execution contexts in the future. Dynamically monitored data samples are sent to the anomaly detection and learning modules for further processing. Additional monitoring data types can be collected as needed, from other application components or from the underlying execution platform. Servlet monitoring data was collected via JBoss' JMX infrastructure, in order to calculate incoming client workloads at the web server side. The code used for monitoring a JBoss servlets via JMX is exemplified in appendix A.2. Additionally, data on hardware resource availability, such as CPU, memory, disk, or bandwidth, can also be obtained, by employing platform-dependent techniques or tools.
Regarding the management framework's footprint and runtime overheads, a main performance concern is that constantly monitoring all EJB methods of a large-scale application would induce excessive and unnecessary performance overheads. In order to avoid this situation, AQuA_J2EE can be configured to only monitor and control a certain, clearly specified subset of application components, or EJBs. A specific list of methods that are to be monitored from each of these managed EJBs can also be configured. In AQuA_J2EE, the set of EJB components and methods to be monitored is statically specified. In a more complex scenario, such management configurations can be dynamically modified to suit runtime changes in the application performance characteristics. For such cases, the approach presented in [28] can be used to determine runtime management requirements and dynamically adjust the set of monitored elements, based on the current system state. More precisely, the management system can be configured to only monitor entry-level components during periods when the system's QoS requirements are being met. This configuration induces minimum performance overheads. Problems detected at the entry-level components are considered to represent global system problems, since they may reflect performance anomalies caused by one or multiple internal components. When such problems are detected at the entry-level components, the monitoring scope can be extended. This allows the management framework to also extract data from internal EJBs and be able to identify the internal components causing the performance anomaly. Once the detected problems are solved, for example by swapping the problem components with redundant variants, the monitoring scope can be restrained again to the top- or entry-level components only. This dynamic approach maintains monitoring overheads at a minimum during normal system functioning, while also allowing for extensive monitoring to be used when needed to detect problem components.
The instrumented JBoss server can be integrated with other management tools for further monitoring and data processing. The way JBoss was modified to work with the COMPAS tool is explained in appendixA.1. In this scenario, monitoring data extracted from the run-

ning JBoss server is dynamically sent to the COMPAS client module. COMPAS subsequently processes the received data, constantly calculating and displaying performance metric graphs and the EJB instances status. In addition, COMPAS can also use anomaly detection logic to recognise problems and raise performance alerts during runtime.

4.5 The Learning Mechanism

AQuA's learning mechanism was devised for analysing monitoring data and inferring high-level performance information on the available redundant components. The methodology used to infer performance information from raw monitoring data was described in section 3.13. A proof-of-concept implementation of this mechanism was provided for AQuA_J2EE. The implementation uses system files for storing monitoring data and inferred performance information. Two separate files are used for each monitored method of each component. One file is used to store raw monitoring data, as collected from the running system. This data represents the performance history of the associated component method. The second file is used to store the component method's performance information, as inferred by analysing the raw monitoring data.

Two alternative solutions were actually implemented, in order to allow the learning process to be executed whether constantly during runtime, or only upon request. As such, the two implementations differ in the manner in which the learning inference process is triggered. In the first approach, the inference process is repeatedly triggered during runtime, as new monitoring data becomes available. In this case, performance information is constantly inferred at runtime. The existing inferred information is repeatedly updated, based on the latest monitoring data available. As such, inferred performance information is updated whenever new values are calculated for the considered monitoring metrics. Such monitoring metrics include for example response times, throughput and workloads. When this first solution is used, the learning module is notified each time new measurements become available. The inference process is subsequently triggered, to use the recent measurements and to update the existing inferred information. The code implementing this process is shown in Listing 4.1. The monitoring class that obtains new raw data from the running system is the `MonitoringDataHandler` class. Namely, the 'calculate' method of the `MonitoringDataHandler` is used to compute average response times, throughputs and workloads over recent intervals. New calculated values are subsequently used to update the component description of the currently active redundant component. The newly available data samples are added to the active component's history, as well as used to update its existing performance information. This process is shown in the code line: `this.rgManager.addMonitoredDataSample(dataSampleJB);`. This command sends a newly calculated data sample to the associated RGManager instance. As discussed, the RGManager instance subsequently uses the data sample to update the component description of the active redundant component. The manner in which existing inferred information is updated based on newly available data samples is described in subsection 3.13.1.

In the second approach, performance information is only inferred upon request. In this case, at runtime, monitoring data is merely stored as component description histories, in a specially allocated system file. When the inference process is triggered, all the available data in the his-

tory file is sequentially processed. Performance information is progressively inferred, as each data sample is read from the history file. AQuA_J2EE can be configured to use either of the two available learning approaches.

Listing 4.1: Inferring performance information from calculated performance metrics

```
MonitoringDataHandler.calculate(){

//...

////code for inferring performance information from calculated performance metrics
//verify whether this manager is updating performance descriptions at runtime
if( this.rgManager.isUpdatingActiveVariantDescription() ){
            //add method data sample to the method's description
            //get the redundant components in this RG
            this.rgVariants = this.rgManager.getRGVariants();
            if(null != rgVariants ){
            //get the currently active redundant component
            //this is the redundant component whose description
            //  will be updated
            String currentActiveVariantKey = this.rgManager.getActiveVariantKey();
            //get the VariantDescriptionManager for the currently active redundant
                component
            VariantDescriptionManager variantDescriptionManager =
                    (VariantDescriptionManager)this.rgVariants.get(
                        currentActiveVariantKey );
            if( null != variantDescriptionManager ){
                    //get the current date
                    Date currentDate = new Date();
                    //instantate new VariantMonitoredDataSampleJB   used to update
                        the variant description
                    VariantMonitoredDataSampleJB dataSampleJB = new
                        VariantMonitoredDataSampleJB();
                    //set current date
                    dataSampleJB.setSampleDate( currentDate );
                    //set the MethodMonitorDataJB
                    dataSampleJB.setMethodData( this.methodMonitorDataJB );
                    //set the EnvironmentData
                    //currently using the default instance, all values set on zero
                    dataSampleJB.setEnvironmentData( this.envMonitorDataJB );
                    //add the method monitored data sample to the variant's
                        descripton
                    //  data sample is added to the variant's data samples history
                        list
                    //  data sample will be used to update the variant's inferred
                        performance description
                    this.rgManager.addMonitoredDataSample( dataSampleJB );
                }
                else{
                    //warning message, new monitored data not considered //...
                    return;
                }
            }
}
```

As part of the learning process, data samples collected from the running system are used to infer higher-level performance information on the managed components. Data samples provide values for the considered performance metrics, such as workload, response times, throughput and available resources, as measured from the running system. These values are repeatedly calculated over fixed intervals, which are signalled by a pre-scheduled timer task. Performance values that are calculated at the end of each interval take into consideration all measured data collected during that interval. For example, the average response time in a certain interval is calculated by adding all response times of all method calls received during that interval and dividing them by the number of received method calls. Workload is calculated by counting the number of method requests received over the considered interval. Performance information is inferred with each new set of calculated metric values, rather than with each new collected measurement. A set of metric values calculated over an interval is referred to as a raw data sample. Inferred information samples have the same format and store the same type of data as raw data samples. Nonetheless, the critical difference is that inferred data represents the result of analysing and merging multiple raw data samples into fewer, more reliable and higher-level data.

The performance information inference process works as follows. Initially, if no inferred information is available, the first available raw data sample is considered as the current inferred sample. After this, the current inferred sample is being updated with each newly available raw data sample. New raw data samples are being obtained by the inference process in different manners, depending on the way the learning process was configured at start-up. If it was configured to update inferred data at runtime, then this process is executed whenever a new raw data sample is calculated, at the end of each interval. In this case, an `EndPeriodNotifier` timer task instance triggers the information inference process. Otherwise, in case the inference process is configured to only be performed upon request, raw data samples will be sequentially read from the history file in the order which they were stored during runtime. The inferred data is updated with each new raw data sample read from the history file.

Figure 4.5 shows the UML sequence diagram for the case when performance information is being inferred during runtime. The `EndPeriodNotifier` instance associated with a managed EJB triggers the inference learning process, at the end of each programmed interval. This is done by signalling the `MonitoringDataHandler` instance to calculate a new set of performance metrics values over the last interval and to create a new raw data sample. Raw data samples are represented by the `VariantMonitoredDataSample` class. The `MonitoringDataHandler` creates a new instance of the `VariantMonitoredDataSample` class and sets its monitored Method, the calculated performance information - workload, response time and throughput - and the environmental conditions information. Data on the current execution environment includes the available CPU, memory and bandwidth. However, execution context metrics are not currently collected in AQuA_J2EE. The `MonitoringDataHandler` subsequently asks the `RGManager` to add the new raw data sample to the currently active component's description. The newly created `VariantMonitoredDataSample` instance is sent as a parameter for this call. The `MonitoringDataHandler` finds the identity of a RG's currently active redundant component by enquiring the associated `RGManager` instance. Upon receiving a request for adding a new raw data sample, the `RGManager` forwards the request to the `VariantDescriptionManager` instance associated with the currently

active redundant component. The `VariantDescriptionManager` instance adds the new raw data sample to the history list and then updates the inferred information of the managed component, as described in section 3.13.

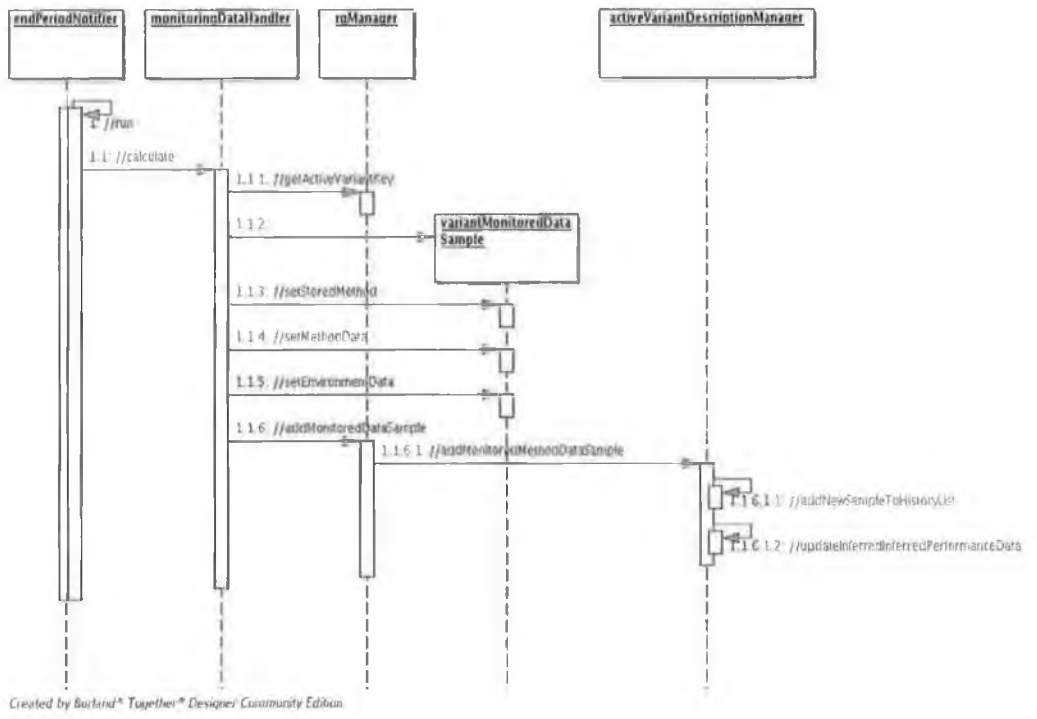

Figure 4.5: UML sequence diagram for the performance information inference process

4.6 Decision Policy-Based Management

AQuA_J2EE's adaptation logic was implemented using a decision policy-based solution. Decision policies were specified using IBM's ABLE Rule Language (ARL)[10]. In ARL, rules are specified in dedicated `.arl` files, separated from the actual underlying system they have to manage. In AQuA_J2EE, the adaptation logic policies are specified in special-purpose `.arl` files, separated from the rest of the framework implementation. Rules can be added, deleted and modified by human system managers without the need to understand, modify, or re-compile any of the underlying framework implementation functions. When using ARL, multiple rule sets can be specified as part of a certain ARL rule base. Different inference engines can be used for executing each separate rule set in the ARL rule base. This means that each rule set can be executed by a different inference engine, as necessary. Possible inference engines include forward or backward chaining, script, or fuzzy engines. The `Script` inference engine was used for interpreting the adaptation logic rules in AQuA_J2EE. A `fuzzy` engine can also be used for example as part of AQuA_J2EE's learning mechanism.

[10]The ABLE Rule Language from IBM: www.research.ibm.com/able

Several types of rules, or decision policies, were devised as part of AQuA_J2EE's adaptation logic. Namely, rules were implemented for the anomaly detection, component evaluation and adaptation decision functionalities. The main management goals of these policy types were presented in section 3.11. In short, *detection policies* are responsible for triggering the component evaluation process. They are used to detect the circumstances in which the available redundant components should be evaluated and the optimal redundant component identified. *Evaluation policies* are responsible for finding the optimal redundant components at the time and in the running context the evaluation process is executed. Finally, *adaptation decision policies* establish whether the managed application should actually be adapted. Adaptation decisions are taken considering the optimal redundant components indicated by the evaluation policies, as well as other factors, such as the cost and risk of the adaptation operation and its predicted outcome. The decision policy-based adaptation logic implemented for AQuA_J2EE is discussed in more detail over the following subsections.

4.6.1 Anomaly Detection Policies

Detection policies are used to analyse the incoming monitoring data and determine the circumstances in which system optimisations are necessary, or possible. Potential problems detected are subsequently signalled to the evaluation module.

In the current AQuA_J2EE implementation, detection policies were designed to analyse monitoring data and sense when thresholds were being exceeded. More precisely, detection policies were implemented to spot cases in which certain parameters values crossed certain predefined thresholds. The performance metrics presently considered are response time, throughput and incoming load on the monitored methods. Other monitoring metrics can be added as needed, including for example the availability of software and hardware resources.

In AQuA_J2EE, anomaly detection policies were implemented as ARL rule sets, accessible from a dedicated `.arl` file. An ARL `Script` inference engine was configured to interpret the detection policies. The `MethodProprietyChangeEvaluator` class was implemented to use detection policies for analysing received monitored data samples. (Figure 4.8 and Figure 4.11). Namely, the detection rules are called to evaluate any changes that may have occurred in the monitoring data collected. The specified policies are executed every time a new value becomes available for one of the considered performance metrics. In the current implementation, monitored performance metrics consist of response time, workload and throughput. These metrics are considered at the EJB method level. Average values are calculated for each performance metric over a certain preconfigured period. For each metric, a new average value is calculated at the end of each period. This triggers the execution of the anomaly detection rules for that metric. The detection rules use collected runtime data on monitored methods and detect possible anomalies.

When triggered, detection policies analyse the data received as input and provide an action result as an output. Input data contains the name of the performance metric which caused the rule set execution. The rule set is started whenever a new value becomes available for that metric. Input data also contains the new calculated value for the respective performance metric. The current implementation of the anomaly detection logic compares the new input value with the previously stored metric value. If the two values indicate that the threshold

set for that metric was crossed, a component evaluation operation is recommended. This is done by returning a `'change'` response value as the result of the anomaly detection process. The possible output results of the anomaly detection process can be `'change'` and `'do_not_change'`. These two actions indicate whether or not a component evaluation is recommended, based on the recently monitored application data.

More complex detection policies can be implemented to recommend changes based on analysing a more extensive set of collected data samples. For example, a detection policy could be specified to recommend a `'change'` action only in case a threshold is being crossed for multiple consecutive periods, rather than for a single period. However, for the experimental tests performed (chapter 5), this behaviour was achieved by tuning the interval over which average metric values were being calculated.

Different detection policies with different configurations are used for analysing each performance metric. In the current implementation, all detection policies for all metrics focus on identifying cases when certain thresholds are being exceeded. The actual threshold values are separately configurable for each different metric.

Additional detection policies were implemented for avoiding false or cascaded alarms. As such, occurrences of small variations across a specified threshold are being ignored and not signalled to the evaluation functionality. This avoids repeated changes from being recommended as a result of fine oscillations around a specified threshold. An insensibility zone was defined for this purpose around each threshold value. Based on this, if currently stored metric values are all on a certain side of a threshold, a change is only recommended in case a new received value crosses the threshold and exceeds it by a certain, configurable amount (Figure 4.6 and Listing 4.2 - lines 50-57).

Further anomaly detection policies can be implemented for detecting additional event types of interest. For example, policies can be specified to detect significant changes in the execution environment, or to analyse data over longer periods [28]. Sets of sequentially monitored data samples could be matched against predefined patterns; each pattern would signify the occurrence of a different anomaly type. When currently monitored data fitted one of the specified patterns, the detection policies would conclude that an anomaly has occurred. As other examples, detection policies can also be specified to trigger the component evaluation process based on the occurrence of exception events, or based on the time of day, week, month, or year.

AQuA_J2EE was designed to allow human system managers to seamlessly add, delete or modify the framework's decision policies. System administrators can specify detection policies as needed to reach their management goals without necessitating a thorough understanding of the underlying framework mechanisms. Detection policies are defined in the dedicated `.arl` file, using the ABLE Rule Language (ARL). Various policies can be devised to analyse any of the available monitoring data and detect diverse performance anomalies in a customised manner. Additional types of monitored data and returned recommended actions can be obtained and respectively defined, as necessary.

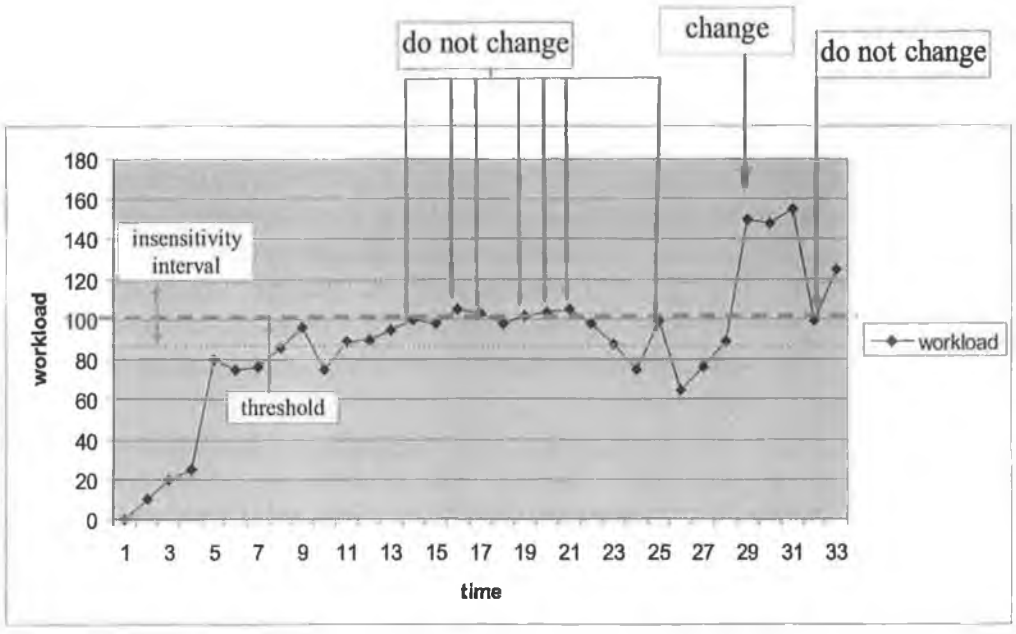

Figure 4.6: workload data example -ignoring oscillations around a threshold when triggering change

Listing 4.2: anomaly detection policies - example

```
%\begin{verbatim}
Categorical action = new Categorical( new String[] {"change", "do_not_change"} );

//other variable declarations
//...

inputs( changedPropertyName, newPropertyValue );
outputs( action );

//the main rule set
void process() using Script {

    //detection rule for changes in the workload property
    change_in_load:
        if( changedPropertyName == LOAD_PROPERTY_NAME )
        then{

            //set the aux variables values
            //  used to evaluate the property change
            auxNewPropertyValue = newPropertyValue;
            auxThreshold = loadThreshold;
            auxPreviousPropertyValue = previous_load;

//invoke another rule set for
//  evaluating the property value change
//this sets the value of the action output result
            invokeRuleBlock("evaluatePropertyChange");

previous_load = auxPreviousPropertyValue;
```

```
30
31          }
32
33      //detection rule for changes in the throughput property
34      //...
35
36      //detection rule for changes in the throughput property
37      //...
38  }//end process
39
40
41  //rule set used by the main process rule set
42  //to evaluate changes in property values
43
44  void evaluatePropertyChange() using Script{
45
46          do_not_change_threshold_not_crossed:
47              if( ((auxNewPropertyValue > auxThreshold) and (auxPreviousPropertyValue >
                     auxThreshold))
48                  or ((auxNewPropertyValue <= auxThreshold) and (
                        auxPreviousPropertyValue <= auxThreshold)) )
49              then{ auxPreviousPropertyValue = auxNewPropertyValue; action = "
                    do_not_change";}
50
51          change_threshold_crossed_downwards_with_more_thanmin_difference:
52              if( (auxPreviousPropertyValue > auxThreshold ) and (auxNewPropertyValue
                    <= auxThreshold )
53                  and (auxThreshold - auxNewPropertyValue > minDifference) )
54              then{ auxPreviousPropertyValue = auxNewPropertyValue; action = "change";
                    }
55          do_not_change_threshold_crossed_downwards_with_less_than_min_difference:
56              if( ( auxPreviousPropertyValue > auxThreshold ) and (auxNewPropertyValue
                    <= auxThreshold )
57                  and (auxThreshold - auxNewPropertyValue <= minDifference) )
58              then{ action = "do_not_change"; }
59
60      //change_threshold_crossed_upwards_with_more_than_min_difference:
61      //...
62
63  //do_not_change_threshold_crossed_upwards_with_less_than_min_difference:
64  //...
65
66    }// evaluatePropertyChange
```

4.6.2 Component Evaluation Policies

Evaluation policies are used to determine the optimal redundant components in a given execution environment. In the current AQuA_J2EE implementation, evaluation policies specify what redundant components are optimal in what execution conditions. For the performed tests, the evaluation policies' triggering conditions were defined based on the incoming load on the components' methods.

For the tested example scenario described in section 5.2, the information on which redundant components were optimal under which workload ranges was obtained based on extensive

testing procedures, which were run on the targeted managed application and execution platform. More precisely, repetitive tests were performed to measure the performance of each redundant component under different incoming loads. Collected data was 'manually' analysed and the optimal redundant component determined for different load ranges. Evaluation policies were accordingly specified to indicate which redundant component to be used under each load range. AQuA's learning mechanism was devised in order to replace this manual process and automatically obtain and update such information instead, during runtime. Thus, the goal of the learning functionality is to enable the management framework to automatically determine optimal redundant components in various execution environments. The automated learning mechanism should be able to perform a similar process to the manual one described.

The evaluation procedure is executed by processing the available evaluation rules, as follows. For each evaluation rule, the current environmental values are compared against the parameter values specified in the rule's conditions. In case of a match the rule is triggered and its action indicates the optimal redundant component. For the tested scenarios, the workload value of the current execution environment is compared against the workload value ranges specified in the rules' conditions. An evaluation rule is triggered if the current load value fits inside the value range specified in the rule's condition. The corresponding rule action sets the name of the optimal redundant component associated with that rule (e.g. Listing 4.3 - lines 61-63).

The component evaluation policies were implemented as an ARL rule set interpretable by a `Script` inference engine. The evaluation rule set is executed whenever the anomaly detection policies (section 4.6.1) signal a potential performance problem or an optimisation opportunity. At the implementation level, the evaluation rules are triggered when the anomaly detection rules return a `'change'` output result. At this point, the evaluation rules consequently determine the optimal change that could be performed considering the available redundant components and the current execution environment.

Whenever the evaluation rules are triggered, they receive a set of input data to process (e.g. Listing 4.3 - line 1). The input data includes the name of the monitored EJB component and the EJB method to analyse. It also contains performance information on the EJB method considered in the anomaly detection and evaluation process. This information provides the latest monitoring values for the performance metrics considered (i.e., response time, throughput and workload). Finally, evaluation rules also receive input information on the current runtime environment status. Method information received as input by the evaluation rules consists of the method signature and the method's runtime monitoring data for the most recent period. Runtime monitoring data includes the method's average response time, workload and throughput over the preceding period. Runtime environment data includes current system CPU, memory and network bandwidth availability. The current AQuA_J2EE implementation assumes that all EJB components run on the same hardware node. Additional environmental information can be added if needed. For example, for an analysed EJB, additional information can be stored on the currently active redundant components used by the neighbouring RGs of that EJB.

The output results of the evaluation rules indicate whether or not an adaptation operation is recommended and if it is then which redundant component would be optimal if activated at the current point (e.g., Listing 4.3 - line 2).

Listing 4.3: component evaluation policies - example

```
inputs{className, methodMonitorJB, environmentDataJB};
outputs{performSwap, optimalVariantName};

void process() using Script {

//calculate time period since last swap operation was performed
: time_since_last_swap_millis = request_time_in_millis - swap_time_in_millis;

//get current performance parameters values
: load = methodMonitorJB.getLoad();
: throughput = methodMonitorJB.getThroughput();
: response_time = methodMonitorJB.getResponseTime();

: methodName = methodMonitorJB.getStoredMethod().getName();

//other rules and conditions
//...

consider_adaptation_enough_time_since_last_swap_and_right_event_source:
//verify that enough time has passed since the last adaptation action
//also verify that the EJB method that triggered
//  the evaluation policies are the ones configured
//  for these particular policies
            if( time_since_last_swap_millis > min_swap_interval_millis
                and className == AccountBean_ClassName
                and methodName == GetAccountID_MethodName )
            then{

//invoke the rule set that finds
//  the optimal redundant component,
//  based on the load performance metric
//  sets the optimalVariantName.byLoad
invokeRuleBlock("processMethodLoad");
//sets the optimalVariantName_byThroughput
invokeRuleBlock("processMethodThroughput");
//sets the optimalVariantName_byResponseTime
invokeRuleBlock("processMethodResponseTime");

//invoke the rule set that determines the
//  final optimal redundant component,
//  considering the optimalVariantName_byLoad,
//  optimalVariantName_byThroughput and
//  optimalVariantName byResponseTime
//sets the optimalVariantName
invokeRuleBlock("determineOptimalVariant");

//invoke the rule set that uses adaptation decisions
//  to conclude whether to adapt the application using the
//  optimal redundant component
invokeRuleBlock("takeAdaptationDecision");
            }
}//end process rule set

//rule set   processMethodLoad
//determines the optimal variant based on the current load
//sets the optimalVariantName_byLoad
```

```
57 void processMethodLoad() using Script{
58 load_over_threshold:
59             if( load >= loadThreshold )
60             then optimalVariantName_byLoad=ThreeHundered_BeanAge_VariantName;
61         load_under_threshold:
62             if( load < loadThreshold )
63             then optimalVariantName_byLoad=Ten_BeanAgeVariantName;
64 }//end processMethodLoad rule set
```

4.6.3 Adaptation Decision Policies

Adaptation decision policies are used to determine whether the managed software application should actually be adapted. The adaptation decision process compares the potential benefits of an adaptation solution with the predicted costs and potential risks of the required adaptation operations. Optimisation solutions are identified and recommended by the evaluation module (subsection 4.6.2). Implementing optimisation solutions into the running application involves one or more component-swapping operations. Adaptation decisions consider the cost of the required component-swapping operations and the potential risk of not actually obtaining the predicted performance benefits or even worsening the overall system performance. In case the adaptation decision policies conclude that a proposed optimisation should be enforced, the system is accordingly reconfigured by activating the optimal redundant components suggested by the evaluation policies.

In the current implementation, the adaptation policies select for activation the redundant component indicated as optimal by the evaluation module. Listing 4.7 shows how two different redundant components can be selected as optimal, based on the rapport between the current incoming load and a specified load threshold. Adaptation decision policies are also responsible for choosing a final optimal redundant component in case more possibilities exist based on evaluating different metrics, or different component methods. For example, a certain redundant component may be optimal under the current environment when only considering the incoming load environmental metric. Nonetheless, another redundant component may be considered optimal with respect to another environmental metric, such as available system resources. Adaptation policies were specified in AQuA_J2EE to resolve any such conflicts and select a unique optimal redundant component for activation. For the tested application scenarios the response time metric was considered to be the most important one for selecting optimal redundant components. The rationale behind this consideration is presented in more detail in subsection 5.2.8. As such, the current adaptation policies in AQuA_J2EE select the optimal redundant component with respect to response times as the final optimal redundant component to be activated (Listing 4.6).

Additional policies were implemented and are used to prevent reactions to false alarms. Conforming to these decision policies, the application will not be adapted if another adaptation operation was executed within a certain preceding interval (Listing 4.4). This avoids cascading adaptation decisions to be triggered based on monitoring data obtained during recent system adaptations. For example, suppose that a significant increase in the incoming workload is detected on a certain managed component. This triggers the evaluation process, which starts searching for potential optimisations in the new execution context. The evaluation module presumably establishes that a redundant component other than the

currently active one would be optimal under the new increased workload. The possible optimisation is sent to the adaptation decision module, which chooses to activate the new optimal redundant component. The component activation mechanism is instructed to perform the corresponding hot-swapping operation. During the component hot-swapping process, all incoming client requests on the adapted component are being delayed (section 4.7). For this reason, the incoming workload measured on the managed component during the adaptation period will be significantly decreased. In turn, this can cause another evaluation and adaptation to be triggered and a redundant component optimal under smaller workloads to be selected for activation. Such management behaviour would be undesirable and should be avoided. The problem that would cause this behaviour is that monitoring data collected during an adaptation process would be interpreted as if it was collected during normal system functioning. A special-purpose adaptation decision was implemented in AQuA_J2EE so as to avoid this situation (Listing 4.4).

For the tested application scenarios, an adaptation decision policy was also implemented to prevent an optimal redundant component from being activated if it was already active at the time the decision was made (Listing 4.4). Such management behaviour would have been undesirable for the particular goals of the tested example scenarios (section 5.2.1). Situations in which this behaviour would have occurred could be generally avoided by correctly specifying detection and evaluation policies. Otherwise, for example, it could happen that the evaluation module finds the same redundant component to be optimal in two different running contexts. Suppose a threshold is specified for a certain performance metric, such as throughput, without two different redundant components being available for the two corresponding execution contexts delimited by that threshold. In such a case, when the threshold is crossed, an evaluation will be triggered. However, the currently active redundant component will be the same one detected as the optimal one in the new execution context. Re-activating the same redundant component under these circumstances would bring no benefits to the system; it should thus be avoided by the adaptation decision functionality. Nonetheless, other scenarios can be envisaged in which re-activating a redundant component would actually be a desirable behaviour. A conclusive example is the case when a mini-rebooting strategy, such as the one proposed in the JAGR project [20], would be employed for providing a certain degree of system fault-tolerance. In the case of EJB applications, such mini-rebooting actions can be implemented by means of EJB hot-deployment operations. AQuA_J2EE can be used to implement the strategy proposed in the JAGR project by re-activating a redundant component in case it was determined to cause erroneous system behaviour. In this case, the adaptation decision policy that did not allow for redundant component to be re-activated would not be used.

In general, AQuA's adaptation logic module was designed to provide the means for human system administrators to seamlessly specify the high-level decision policies governing the desired system management behaviour. Additional policies can also be specified to consider the cost of the required adaptation operations and the outcomes of previous, similar adaptation decisions. Such policies were not used in the tested scenarios (subsection 5.2.6), since the monitored adaptation operations proved to have a limited, sustainable impact on system performance. Decision policies can also be designed to deal with eventual, conflicting optimisation demands.

Listing 4.4: adaptation decision policy to ensure minimum time between subsequent adaptations

```
1  //calculate time period since last swap operation was performed
2  : time_since_last_swap_millis =
3  request_time_in_millis - swap_time_in_millis;
4
5  //...
6
7  do_not_adapt_too_little_time_since_last_swap:
8              if(
9      time_since_last_swap_millis <= min_swap_interval_millis )
10             then {
11     performSwap = new Boolean( false );
12     optimalVariantName = null;
13 }
```

Listing 4.5: adaptation decision policy to not activate an already active component

```
1
2  //decides wheher to adapt the application or not - to perform the component hot
       swapping or not
3    //sets the performSwap (and possibly sets the optimalVariantName to null, e.g., for
         a false performSwap)
4    void takeAdaptationDecision() using Script{
5
6          swap_if_current_and_optimal_variants_are_different:
7              if( currentVariantName != optimalVariantName )
8              then {
9                  performSwap = new Boolean( true );
10                 currentVariantName = optimalVariantName;
11                 date = new Date();
12                 swap_time_in_millis = date.getTime();
13                 }
14             else {
15                 performSwap = new Boolean( false );
16                 optimalVariantName = null;
17                 }
18   }//end takeAdaptationDecision
```

Listing 4.6: adaptation decision policy to determine final optimal component from multiple candidates

```
1    //determines the optimal variant considering the (possibly different) optimal
         variants based on load, throughput, response time..etc
2    //different policy used for each component / method
3    //sets the optimalVariantName
4    void determineOptimalVariant() using Script{
5      decision_for_AccountBean_getAccountId:
6          if( (className == AccountBean_ClassName) and
7      (methodName == GetAccountID_MethodName) )
8          then{
9      optimalVariantName = optimalVariantName_byResponseTime;
10   }//end determineOptimalVariant
```

Listing 4.7: adaptation decision policy - determine optimal redundant component with respect to load only

```
1  //determines the optimal variant based on the current load
2  //sets the optimalVariantName_byLoad
3  void processMethodLoad() using Script{
4        load_over_threshold:
5            if( load >= loadThreshold )
6            then optimalVariantName_byLoad=ThreeHundered_BeanAge_VariantName;
7        load_under_threshold:
8            if( load < loadThreshold )
9            then optimalVariantName_byLoad=Ten_BeanAgeVariantName;
10 }//end processMethodLoad
```

4.7 Component Activation

The *component activation* functionality is used to perform component-swapping operations, during the actual application adaptation process. The application adaptation operations to be performed are dictated by AQuA's adaptation logic (i.e. by the adaptation decision policies).
In AQuA_J2EE, the component activation function was implemented to work on the JBoss application server. Using this implementation, EJB components can be swapped at runtime without breaking client sessions or raising exceptions. The implemented solution is based on the hot-deployment facility provided by JBoss. The hot-deployment function allows packages to be dynamically deployed on JBoss by copying them into a certain deployment directory, without the need to stop and restart the server. JBoss periodically verifies its deployment directory and senses any changes in the existing packages. Redundant components can differ in their EJB implementation class, or in their deployment descriptors. JBoss will detect any change in any of the package files representing a redundant component. A package hot-deployment operation is triggered whenever changes are detected in an existing package, or when new packages are detected. In other words, JBoss performs a hot-deployment operation for all EJBs in a package, whenever that package becomes available for the first time in the deployment directory, or when a new version is loaded for an existing package. In the first scenario, all EJBs in the new package are simply deployed on JBoss, while the server and its applications continue to run. In the second scenario, the hot-deployment process involves two operations. First, all EJBs in the old package version are un-deployed. Second, all the EJBs in the new package version are being deployed instead.
The main difficulty with JBoss' hot-deployment facility is that in most cases it cannot be successfully used to hot-swap EJBs while under heavy workloads. There are two main reasons behind this problem. First, as previously indicated, when JBoss performs a hot-deployment operation, it first un-deploys the old EJB and then deploys the new EJB variant. This creates an availability gap between the two deployment operations, during which neither the old nor the new EJB variants are available. The JBoss application server used to work with AQuA_J2EE was modified so as to solve this problem. Namely, in the modified JBoss version, all incoming requests are being intercepted and delayed for the duration of a hot-deployment operation. This means that client requests for a certain EJB will be delayed for the time the EJB is being replaced by a newer variant. Once the deployment operation is completed, the delayed requests are let through and handled by instances of the new EJB variant.

A second hot-deployment related problem occurs in JBoss when Stateful Session EJBs are used as part of an EJB application. This is because Stateful Session bean instances maintain their state between successive client calls, for the entire duration of a user session (subsection 2.4.2). As such, all client calls belonging to a certain session must be handled by the same Stateful Session bean instance. Thus, problems will occur if a Stateful Session bean is hot-swapped in the middle of a user session; the reason is that subsequent client calls belonging to that session will no longer be able to find the particular Stateful Session instance that used to handle this session. Furthermore, the same problem occurs for bean instances that are used by Stateful Session beans, since they are also being maintained as part of the session state. This problem was solved in the modified JBoss server as follows. Before the hot-deployment operation is started, the container of the EJB to be replaced is instructed to block all requests for the creation of new EJB instances; all other requests are let through. This allows started user sessions to terminate, while not allowing any new session to be initiated. When no more instances of the targeted EJB are available in the container, the hot-deployment operation is executed; after hot-swapping terminates, all incoming requests are unblocked. As a further improvement, the EJB instance cache of the targeted EJB is flushed as soon as no activity is detected on the stored instances for a certain period.

Part of the implemented component activation solution, multiple redundant components are prepared and made available at runtime, from a known location. The component activation module receives information on the system file path where each available redundant component of each managed RG is located. Each redundant component is provided in the form of a JBoss deployable package, such as a `.jar` or an `.ear` archive. File system directories are used to store redundant components. Each Redundancy Group (RG) is represented by a separate directory. This means that all redundant components that belong to a certain Redundancy Group (RG) are stored in a single directory, which was dedicated to that RG. A mapping exists between the unique name of an existing RG and the name of the corresponding directory used to store the redundant components of that RG. In AQuA_J2EE component-swapping operations are performed by copying the package of the redundant component to be activated over the package of the currently active redundant component, in JBoss' deployment directory.

A Graphical User Interface (GUI) was also implemented to allow system administrators to manually manage redundant components. In the current version, the GUI displays all managed RGs and their available redundant components. It also allows redundant components to be individually selected and activated. The GUI can be further extended so as to also support the addition, removal and versioning of RGs and associated redundant components. Thus, redundant components can be activated whether automatically, based on AQuA_J2EE's adaptation logic decisions, or manually, based on a system administrator's decisions (Figure 4.7). Both cases use the implemented component-swapping procedure to activate redundant components.

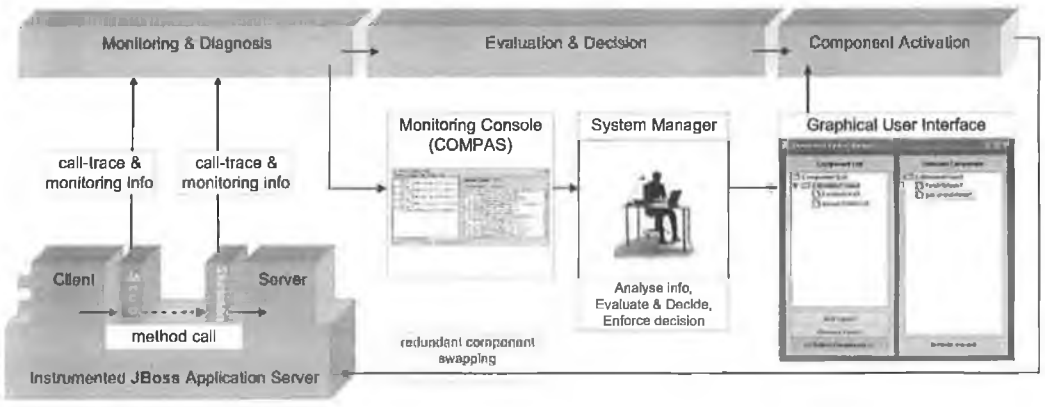

Figure 4.7: activating redundant components - manual and automatic options

4.8 AQuA_J2EE's Implementation Classes Explained

4.8.1 Implementation Classes

This subsection gives an overview of AQuA_J2EE's implementation classes and explains the main roles and functionalities of each class. It also shows how classes work together in order to provide AQuA's main management functionalities. The manner in which the JBoss application server was modified so as to work with the AQuA_J2EE prototype is also described. The UML class diagram in Figure 4.8 shows the main classes used to implement AQuA_J2EE, their interconnections and their main association roles. Each class is briefly described over the following subsections. Appendixes A.2, A.1 and A.3 provide implementation details on the way JBoss classes were modified in order to be integrated with the AQuA framework.

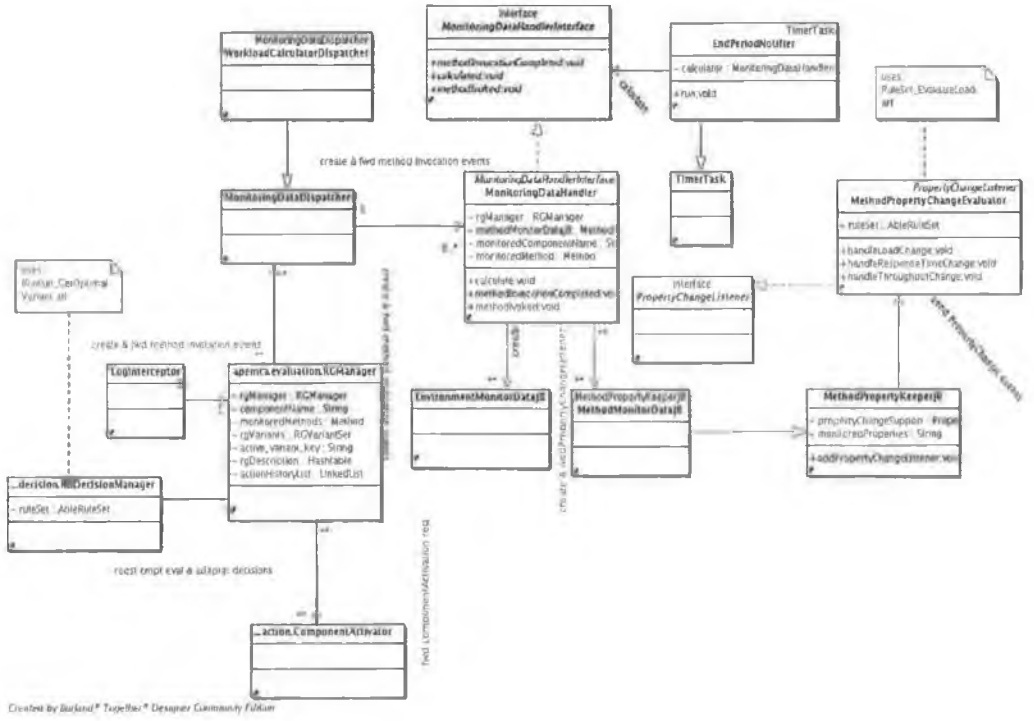

Figure 4.8: UML class diagram for AQuA_J2EE

The LogInterceptor class

The *LogInterceptor* class is part of the JBoss application server's implementation. In AQuA_J2EE, system instrumentation was implemented by modifying JBoss' EJB containers. More precisely, one of the interceptor classes in the container interceptor chain was modified. Namely, the LogInterceptor class was instrumented to capture messages to and from the EJB instances managed by the container. The UML diagram in Figure 4.9 shows how the modified LogInterceptor is integrated and used in a typical JBoss container for managing Entity EJBs. As indicated in the figure, an instance of the modified LogInterceptor class intercepts all incoming client requests and corresponding outgoing responses. These events, along with some associated data, are forwarded to the management framework instance that was given to the LogInterceptor instance when it was created. The modified JBoss server was integrated with two management frameworks. These were the COMPAS monitoring framework and the AQuA_J2EE management framework. When integrated with COMPAS, the LogInterceptor was modified to send its intercepted events to the COMPAS Client [28]. As indicated in the figure, the JBoss EJB Instance Pool was also modified to send relevant events to the COMPAS client. Such events included the creation and removal of instances for the managed EJB class. The goal was to enable COMPAS to compute the current number of available instances for each monitored EJB class at all times. When integrated with AQuA_J2EE, JBoss' LogInterceptor was modified to send relevant events to an instance of the RGManager class. In AQuA_J2EE, the RGManager is the main class responsible for managing a Redundancy Group. In the adopted decentralised approach, Redundancy Groups were designed at the EJB granularity, and one management framework instance was created for each managed RG. Implicitly, as there is one JBoss container per deployed EJB class and one RGManager instance for

each management framework instance, there will be one RGManager instance associated with each JBoss EJB container. Each JBoss container contains in turn one LogInterceptor instance.
As an alternative solution to modifying the existing JBoss LogInterceptor class, a new interceptor could have been implemented. In this case, JBoss EJB containers would have been correspondingly configured to use the new interceptor in their respective interceptor chains. Nonetheless, this approach would not bring a significant contribution with respect to the targeted research goal, which was to test the management capabilities of the AQuA framework prototype.
The way the LogInterceptor class was modified for sending monitored data to AQuA_J2EE is described in more detail next. The design and implementation of the JBoss server, including its plugin-in based interceptor chain, were described in more detail in section 4.1.1. The relevant instrumentation code and explanatory comments inserted in the LogInterceptor class are listed in appendixes A.2 and A.1.
In JBoss, an instance of the LogInterceptor class is created whenever a new EJB container is instantiated to manage a newly deployed EJB component. Part of the JBoss server modification, the LogInterceptor's instantiation process was augmented to create an instance of the RGManager class. In turn, the RGManager will instantiate the rest of the AQuA_J2EE framework's classes. There is one AQuA_J2EE instance for managing the performance of each deployed EJB, given that the EJB was previously configured to be managed by AQuA. For instantiating the framework, the LogInterceptor initially retrieves the managed EJB's metadata, including the EJB's JNDI name and class name. It then establishes whether or not an RGManager instance already exists for this managed EJB component. An RGManager instance may already exist in case the managed EJB is not a new EJB component (or RG) in the system. This can happen if an EJB with the same JNDI name has previously been deployed on JBoss, and at the present time a new EJB class is deployed in place of the old one. This is equivalent to a redundant component implementing the EJB component with that JNDI name being swapped in place of the old redundant component. In this case, the RG for the managed EJB component is not new in the system. However, the EJB class providing the RG's interface was changed via a hot-deployment operation. The old EJB class is un-deployed and the EJB container managing it destroyed, along with its LogInterceptor instance. When the new EJB class is deployed, a new EJB container and LogInterceptor instances are being created. However, it is important to note that one AQuA_J2EE instance is created and tied to a Redundancy Group, as a whole managed entity. It is not tied in particular to any of the redundant components (i.e. or EJB classes) providing the RG's external methods. Thus, the AQuA_J2EE instance should not be destroyed and recreated in case different redundant components are activated in the RG. If this happened, all data accumulated for that RG would be lost whenever a different redundant component was activated, by hot-deploying a different EJB class. This would not be the desired behaviour. For this reason, an AQuA_J2EE instance, including the RGManager instance, is maintained for the entire duration a RG exists in the system. This is done by maintaining a list of all instantiated RGManagers in a single RGManagerAdministrator instance. The RGManagerAdministrator instance is a singleton entity created the first time the JBoss server is started. Each RGManager instance for each RG is uniquely identified by its RG name. A unique RG name is created based on the unique JNDI name of the managed EJB component. The JNDI name of an EJB does not change when different EJB classes are swapped and activated in a RG. During its instantiation, the LogInterceptor class enquires the RGManagerAdmintsrator about the existence of an RGManager with the JNDI name of the EJB it

needs to manage. If an RGManager instance does not already exist, the LogInterceptor creates one and registers it with the RGManagerAdmnistrator. The JNDI name of the managed EJB component is associated with the newly registered RGManager instance in the RGManager-Admministrator's list. The name of the EJB class being deployed is also used to retrieve the EJB's provided methods and compare them with the list of methods that should be monitored, conforming to AQuA_J2EE's configuration. During runtime, the LogInterceptor only uses the created framework instance in case its managed EJB is indicated as a component that should be managed by AQuA_J2EE. This is specified in the management framework's configuration. This implies that framework instances are only created and used for those EJB components that AQuA_J2EE was previously configured to manage. In case an RGManager instance does already exist, the LogInterceptor instance will simply store and use this instance as the contact point with the AQuA_J2EE framework.

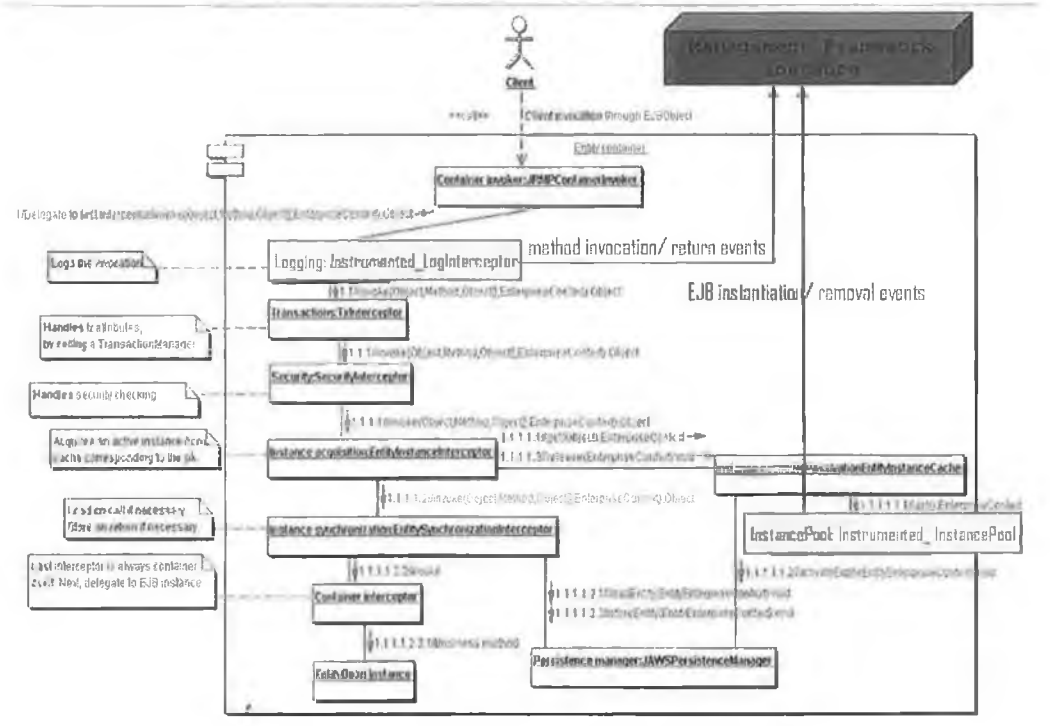

Figure 4.9: instrumenting JBoss's EJB container

The RGManager class

The *RGManager* class is a central entity in the AQuA_J2EE implementation. It coordinates the core management operations for a Redundancy Group (RG) and is the main point of contact between AQuA_J2EE and the underlying managed system. In the current implementation, all monitoring data extracted from the JBoss server platform is sent to the RGManager for further processing. An RGManager instance is created for each managed RG. This operation is performed the first time an EJB class is being deployed for that RG. The RGManager instantiation process is controlled by a LogInterceptor instance. This LogInterceptor instance is part of the JBoss container that manages the deployed EJB. Figure 4.10 shows how a LogInterceptor belonging to a JBoss container is related to an RGManager belonging to an AQuA_J2EE instance,

for managing a certain deployed EJB component, as part of a RG.
At its construction, the RGManager class receives the JNDI name and provided methods of the EJB it has to manage. The JNDI name is used to uniquely identify the RG. From the list of EJB provided methods, the RGManager identifies the ones that should be monitored as indicated by AQuA_J2EE's current management configuration. The RGManager also retrieves the list of the available redundant components for the RG it represents. This is done by reading the contents of the system directory allocated to this RG, which is uniquely identified using the RG's name. In turn, the RG's name is formed based on the unique JNDI name of the managed EJB component. In the RG directory, there is one separate directory for each available redundant component. Each such directory contains all sources to be deployed for the corresponding redundant component (e.g., an .ear or .jar archive).
For each of the identified redundant components, the RGManager creates an instance of the VariantDescriptionManager class, to manage the performance description (section 3.11.2) of that redundant component. The RGManager maintains a list of all VariantDescriptionManager instances, identified by the corresponding redundant component name. This way, the RGManager can always access performance information about any of the available redundant components in the managed RG. Part of its instantiation process the RGManager subsequently registers itself with the RGManagerAdministrator, so that it can be located and utilised when needed. For example, JBoss' LogInterceptor instances need to obtain references to RGManager instances for sending them monitoring events from the running application. RGDecisionManager instances also need to access RGManager instances, in order to send them adaptation decisions. The RGManager also creates one RGDecisionManager, one MonitoringDataDispatcher and one ComponentActivator instances. The main roles and functionalities of these classes are described in the following paragraphs.
Once created, an RGManager instance can receive method invocation events from the system monitoring functionality, which is implemented in AQuA_J2EE through JBoss' LogInterceptor. Method invocation events are forwarded to the corresponding MonitoringDataDispatcher instance for further processing. The LogInterceptor also sends method completion events to RGManagers, which are forwarded in a similar manner to the MonitoringDataDispatcher. In case the RGDecisionManager takes an adaptation decision, based on current monitoring data, available information and decision policies, it instructs the RGManager to enforce that decision into the running application. The redundant component to be activated as part of the adaptation operation is indicated. The RGManager subsequently instructs the ComponentActivator to activate the specified redundant component. When the new redundant component has been successfully activated, the RGManager updates its information on the current application configuration (i.e., which redundant component in the managed RG is currently active).
When AQuA_J2EE is configured to infer performance information from monitoring data during runtime (section 4.5), the RGManager receives monitoring data samples as soon as they become available. Whenever it receives a new data sample, the RGManager identifies the currently active redundant component and forwards the data sample to the corresponding VariantDescriptionManager, for further processing.

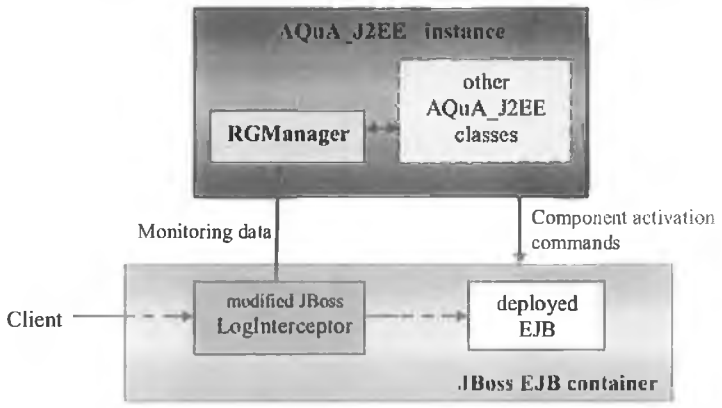

Figure 4.10: AQuA_J2EE's integration with JBoss

The MonitoringDataDispatcher class

The *MonitoringDataDispatcher* class is responsible for dispatching method invocation events to the entities responsible of handling such events. The MonitoringDataDispatcher receives method invocation events from the corresponding RGManager, part of the same AQuA_J2EE framework instance. Received events are being dispatched to the corresponding monitoring data handlers, based on the respective methods for which each event was issued. One monitoring data handler is available for managing each method. All events generated by monitoring a certain method are being dispatched to the monitoring data handler responsible for that method. Instances of the MonitoringDataHandler class are used for handling monitoring data and events. Thus, a MonitoringDataDispatcher dispatches method invocation events to MonitoringDataHandler instances. One MonitoringDataDispatcher instance is created and associated with each RGManager. Multiple MonitoringDataHandler instances are created for each RGManager, each instance being responsible for a separate monitored method.

The MonitoringDataHandler class

The *MonitoringDataHandler* class is responsible for managing and processing collected monitoring data for a certain component method. As previously explained, there is one AQuA_J2EE instance managing each RG. This AQuA_J2EE instance is used to manage the RG irrespective of the currently active redundant component in that RG. In conformance with this, a single MonitoringDataHandler instance is created and allocated for managing a certain (RG) method, regardless of the currently active redundant component handling requests for that method. As such, one MonitoringDataHandler instance is created for each external method a RG provides. The MonitoringDataDispatcher associated with the RG creates the required MonitoringDataHandler instances, during its own instantiation process.

During runtime, a MonitoringDataHandler instance receives monitoring data from the MonitoringDataDispatcher. This includes method invocation and execution completion events, as well as method response time data. As explained in subsection 4.8.3, the MonitoringDataHandler uses incoming monitoring data to periodically calculate performance metric values, including method response times, workload and throughput. An EndPeriodNotifier timer task is created and scheduled during the instantiation process of the MonitoringDataHandler. The role of this timer task is to notify the MonitoringDataHandler instance when to calculate

method-related performance values. Such notifications are regularly sent at a fixed timed interval, configured when the timer task is created.
The performance values calculated for the most recent completed interval are stored in a MethodMonitorDataJB JavaBean instance. Changing the values stored in this JavaBean instance at the end of each interval causes a MethodPropertyChangeEvaluator to evaluate the change and detect or predict any potential performance anomalies (Figure 4.11). In its constructor, the MonitoringDataHandler creates and instance of the MethodPropertyChangeEvaluator class and registers it as a property change listener for the MethodMonitorDataJB JavaBean. This way, any change in the considered metric values is detected and can be analysed.
The MethodMonitorDataJB JavaBean instance associated with each MonitoringDataHandler instance is used to store monitoring performance data on the monitored method. Besides this, an EnvironmentMonitorDataJB JavaBean instance is also created and associated with the MonitoringDataHandler, in order to store environmental related monitoring data. Changes in the environmental condition related metrics can be detected and analysed using the same mechanism as the one used to spot changes at the method level.

The EndPeriodNotifier class

The *EndPeriodNotifier* class is used to schedule repeated time intervals for a monitoring data handler, so the handler can regularly calculate its performance values. An EndPeriodNotifier object is associated with each MonitoringDataHandler object that is created as part of an AQuA_J2EE instance.
The EndPeriodNotifier class is an extension of the java.util.TimerTask class. Thus, a java.util.Timer object can be scheduled to repeatedly execute this task at regular intervals. Each EndPeriodNotifier instance is created and scheduled with such a timer by the MonitoringDataHandler instance that uses it. The actions performed as part of an EndPeriodNotifier's task involve calling the associated MonitoringDataHandler and notifying it to calculate and update its performance metric values.

The MethodPropertyChangeEvaluator class

The *MethodPropertyChangeEvaluator* is used to analyse value changes that occur in the considered performance metrics, during runtime. The MethodPropertyChangeEvaluator class implements the java.beans.PropertyChangeListener interface. Thus, it can be registered as a property change listener for Java Beans. As such, each MethodPropertyChangeEvaluator object is registered with a MethodMonitorDataJB Java Bean, so as to dynamically detect changes in the Java Bean's stored values. Detected value changes are analysed using decision policies available from an ARL rule set file (i.e., the `RuleSet_EvaluateLoad.arl` file). If a potential performance anomaly is detected or predicted based on these policies, the associated RGDecisionManager is notified to search for a possible optimisation solution. In the current implementation, value change patterns are analysed based on the two most recently monitored values, for any of the considered metrics. Namely, the current and the previous monitored values are being compared. Another possible approach which was developed for the management framework analyses a configurable number of recently monitored values instead of only analysing the two most recent ones. Nonetheless, this approach was not needed in the tested example scenarios and thus was not integrated in the

current AQuA_J2EE implementation.

The RGDecisionManager class
The *RGDecisionManager* class is used to take component evaluation and adaptation decisions. It uses decision policies, which it can access from an ARL rule set file (i.e., the `GetOptimalVariant.arl` file). When a PropertyChangeEvaluator detects a possible performance anomaly, it notifies the associated RGDecisionManager, asking it to search for possible optimisation solutions. Consequently, the RGDecisionManager uses the available ARL policies for component evaluation and determines the optimal redundant component, under the current execution conditions. Adaptation decision policies are subsequently used to determine whether or not the optimal redundant component should actually be activated in the running system. If a decision to adapt the application is taken, the RGDecisionManager notifies the associated RGManager to activate the selected redundant component. The RGManager handles such application adaptation requests by forwarding them to the associated ComponentActivator instance.
The RGDecisionManager is created by the RGManager during its own instantiation process.

The VariantDescriptionManager class
The *VariantDescriptionManager* class is used to manage the performance descriptions of redundant components. A separate component description is available for the performance characteristics of each separate redundant component in a RG. Thus, a VariantDescriptionManager instance is created for each available redundant component in a RG. All component descriptions in a RG can be accessed from the RGManager instance managing that RG. Each VariantDescriptionManager instance can be uniquely identified by the name of the redundant component it describes.
As presented in section 4.5, component descriptions consist of histories of monitored performance values and of inferred performance information obtained from analysing those values. System files are used in AQuA_J2EE as storage support for component descriptions. The information stored in these files is updated with each new collected monitoring data sample. This process can be performed whether repeatedly during runtime, or statically upon request. When component descriptions are being updated during runtime, an instance of the VariantDescriptionManager class is used for this purpose. Namely, the VariantDescriptionManager instance associated with the currently active redundant component constantly receives monitored data samples for further processing. The description manager uses these data samples to repeatedly update the redundant component's description. The data history and inferred information files used to store the component's description are accordingly updated in effect. The strategy and algorithm used for this purpose are described in more details in section 3.13.

4.8.2 Design Details for AQuA_J2EE's Adaptation Logic

AQuA_J2EE's adaptation logic is executed every time new data samples become available from the monitoring process. Part of the adaptation logic, the anomaly detection policies are initially evaluated. These policies are loaded and processed from the `RuleSet_EvaluateLoad.arl` ARL rule set file. In case a potential performance anomaly

is detected or predicted, the component evaluation policies are subsequently run. Finally, in case a possible optimisation solution is found for the current execution context, the adaptation decision policies are processed in order to evaluate the viability of the proposed adaptation solution. Component evaluation and adaptation decision policies are loaded and processed from the `RuleSet_GetOptimalVariant.arl` rule set file. A positive adaptation decision outcome triggers the corresponding application adaptation process, which involves the activation of the optimal redundant components.

The UML sequence diagram in Figure 4.11 illustrates the way the anomaly detection, evaluation and adaptation process is implemented in AQuA_J2EE. The adaptation logic execution is triggered when a MonitoringDataHandler calculates one of its managed performance metrics, thus updating the metric's value. Figure 4.11 illustrates the case when the considered performance metric is the incoming workload on a managed component method. The calculated workload values are set in the associated MethodMonitorDataJB JavaBean. This JavaBean was implemented so as to send a PropertyChangeEvent each time one of its property values was being modified. Thus, the MethodMonitorDataJB sends a property-change event whenever its workload property is changed. It also sends such events when the values of its response time and throughput properties are being modified. This paragraph discusses an example where the incoming load is the updated property. Nonetheless, a similar process is performed for the other metrics.

Considering the workload metric example, each PropertyChangeEvent object contains the new calculated load value and the previous load value for the managed component method. It also contains the JavaBean instance which was the source of the property change event. This JavaBean contains information on the managed component and method, as well as on the other metric values, as needed. Property-change events raised by a certain MethodMonitorDataJB JavaBean are sent to a corresponding MethodPropertyChangeEvaluator for further processing. The particular evaluator object that receives the events was previously registered as a listener for the particular JavaBean instance raising those events. The MethodPropertyChangeEvaluator analyses a received event and determines the performance metric that caused the event to be raised. In the described example the evaluator determines that the received event was caused by a change in the incoming load metric. Based on this, the property change evaluator triggers its logic for handling modifications in the incoming load metric. For this purpose, the change evaluator uses the values it receives for the current and previous incoming loads and the available anomaly detection policies.

When a performance anomaly is detected, the MethodPropertyChangeEvaluator notifies the associated RGDecisionManager to search for an adaptation solution to the potential problem. When this situation occurs, the RGDecisionManager uses its component evaluation and adaptation decision policies to find a viable optimisation solution. If successful, it forwards the adaptation decision result to the RGManager, indicating the redundant component to be activated. In turn, the RGManager requests the associated ComponentActivator object to perform the adaptation operation.

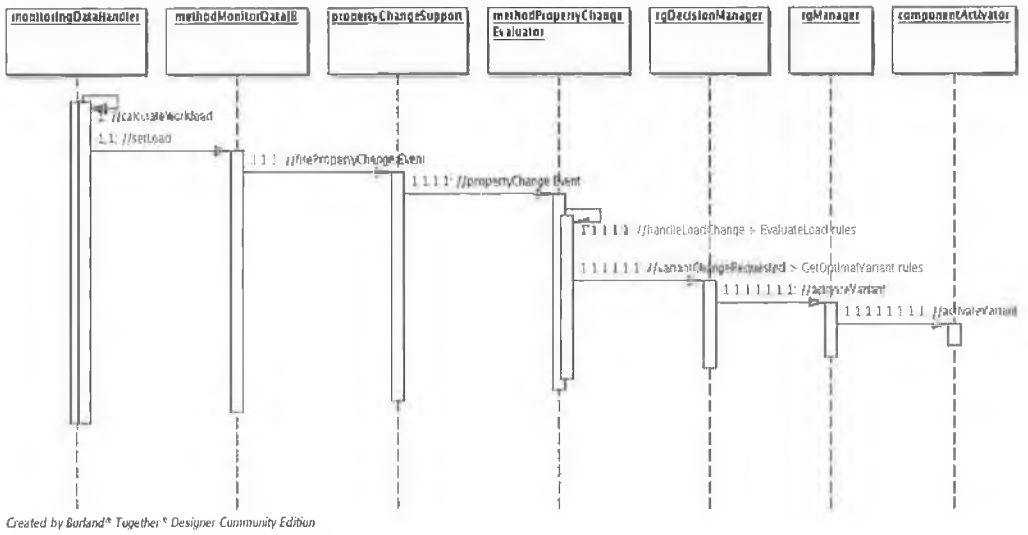

Figure 4.11: UML sequence diagram for AQuA_J2EE's adaptation logic

4.8.3 Design Details for AQuA_J2EE's Data Dispatching Process

This section describes the way client method invocations are being intercepted and processed by AQuA_J2EE, in order to extract monitoring data from the running system. The UML sequence diagram in Figure 4.12 shows the main classes involved in this procedure, as well as the way these classes interact in order to process method invocation events.

When a client invokes a method on an EJB method, the JBoss container that manages that EJB intercepts the method request. The request is intercepted by the modified LogInterceptor in the container. The LogInterceptor first verifies whether AQuA_J2EE was configured to manage this EJB component. If yes, the interceptor signals the method invocation event to the RGManager instance associated with this EJB container. JBoss containers were modified so as to only communicate with the AQuA_J2EE framework via RGManager objects. This approach simplifies the instrumentation procedure for other middleware servers, as a single AQuA_J2EE class, the RGManager, needs to be integrated with the server. The RGManager instance associated with an EJB container is notified whenever a method invocation event is being intercepted in that container. The invoked Method and the unique JNDI name of the invoked EJB are sent as parameters to the RGManager instance. The RGManager subsequently forwards the method invocation notification and associated parameters to the associated MonitoringDataDispatcher instance. The MonitoringDataDispatcher class is responsible for dispatching method invocation events to the respective MonitoringDataHandler instances, which are in charge of managing each component method. Thus, upon receiving a method invocation event, a MonitoringDataDispatcher instance first determines the particular MonitoringDataHandler instance responsible for managing the particular method invoked. It subsequently forwards the method invocation event and associated parameters to the identified MonitoringDataHandler instance, for further processing.

In the current implementation, MonitoringDataHandler instances are responsible for using

method invocation and method return events for calculating incoming workloads, average response times and throughputs for their respective managed method. Thus, each time a MonitoringDataHandler receives a method invocation event, it increments a method invocation counter. This counter is used to calculate the workload, average response time and throughput of the managed method, over a certain period. The end of each such period is signalled to the MonitoringDataHandler by an associated EndPeriodNotifier instance. The EndPeriodNotifier class is an extension of the java.util.TimerTask class, which can be scheduled to repeatedly perform certain specified tasks at regular time intervals. In AQuA_J2EE, an EndPeriodNotifier is scheduled to regularly notify an associated MonitoringDataHandler of when to calculate the performance values for its managed method.

Once the method invocation event is processed by AQuA_J2EE, the JBoss LogInterceptor forwards the method invocation through the container interceptor chain. At the end of this chain, an instance of the targeted EJB component handles the method request and then returns, possibly also sending a response value. Method return event are handled similarly to method invocation events. Namely, the JBoss LogInterceptor intercepts the method response and signals the event to the associated RGManager instance. The response event is forwarded in a similar manner through the management framework's instances, as in the case of method invocation events. An additional piece of information, the method response time, is sent in this case as an extra parameter along with the method completion event. This value is used by the MonitoringDataHandler instance to calculate the average response time of the managed method, over the current time interval. Once the method completion event is processed, the JBoss container returns the method response to the initiating client.

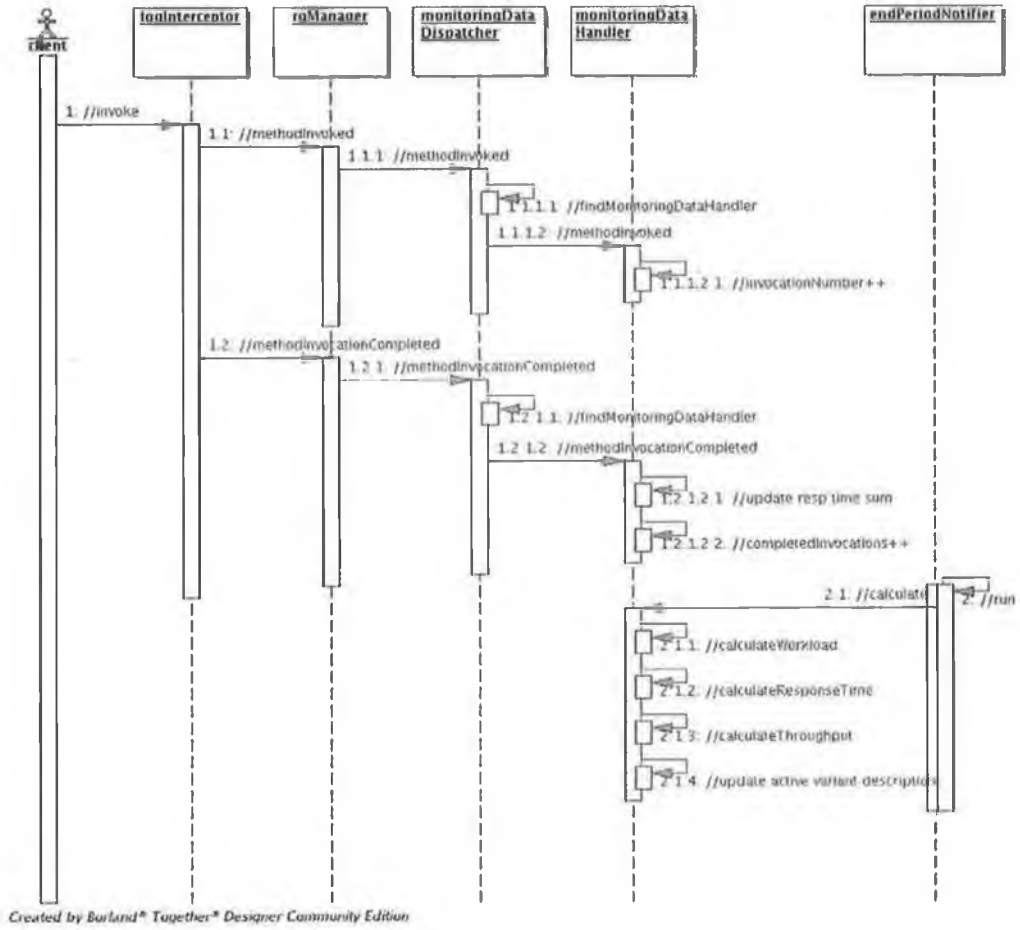

Figure 4.12: UML sequence diagram for monitoring data dispatching in AQuA_J2EE

CHAPTER

FIVE

Experimental Work: Tests and Results

Chapter Summary

This chapter presents the thesis experimental work. Two main goals were pursued for validating the thesis contributions. First was to exemplify cases in which the redundancy-based management solution would benefit application performance. The second goal was to show how the AQuA management framework can be employed to implement the redundancy-based solution and automatically optimise system performance.

Two practical example applications were tested towards attaining the first goal. Tests were performed on the two examples under various dynamically changing execution environments. Obtained results indicated how different redundant components were optimal in different execution contexts. Performance values measured during the tests clearly showed the demand for dynamic system adaptations. This proved the applicability and potential benefits of the redundancy-based optimisation solution in the tested scenarios.

The AQuA_J2EE framework prototype was employed for accomplishing the second experimental goal. AQuA_J2EE's management capabilities were tested for automatically optimising one of the example applications. In this scenario, the managed application was automatically adapted to dynamic workload variations in its execution environment. AQuA_J2EE dynamically collected monitoring data, detected workload variations and decided to adapt the application, based on a clearly specified set of decision policies. Redundant components were automatically swapped in effect, without interrupting system execution. The measured performance of the automatically optimised application proved superior to the results obtained when the application was not adapted. In addition, AQuA_J2EE's learning mechanism was tested on collected monitoring data. Preliminary test results validated the learning algorithm's ability to group and merge similar data samples into clusters of inferred information.

Goals of this chapter:

- Two practical example scenarios were identified and proved to benefit from the redundancy-based optimisation solution. In the first example, two redundant components alternately yielded optimal response times depending on the available network bandwidth. The second example sowed how two redundant components were optimal with respect to memory usage depending on the incoming workload.
- The AQuA_J2EE prototype was successfully tested to automatically adapt and optimise a sample software system, without requiring human intervention
- Preliminary test results proved the ability of AQuA_J2EE's learning function to group and merge similar monitoring data into clusters of information

5.1 Response Time Variations with Network Load Scenario

An example application was implemented for the EJB technology[1] to show the potential benefits of a redundancy-based optimisation solution. The purpose of this example is to show how different redundant components, differing in their implementation strategies, can alternately yield optimal performance under different execution contexts. Tests performed on this example application showed how the performance of two different redundant components changes with variations in their execution environment [30]. Namely, obtained results indicate how the redundant components' response times vary with changes in their available network resources. Conforming to these results, each redundant component is optimal in a different range of environmental conditions. Therefore, optimal performance will be obtained if the two redundant components were alternately used as the boundary between the two execution contexts was dynamically crossed. Further examples are available from related work, in the area of component hot-swapping for performance optimisation purposes (e.g. [3], [93], or [101]).

The sample application was implemented so as to make use of the component redundancy concept. As such, two different component implementations were provided to supply the same functionality, while being optimised for different execution contexts. The functionality provided was to repeatedly retrieve information from a remote database (DB). The two redundant components differed at the design level, as depicted in Figure 5.1. The first redundant component design involves a single Stateless Session EJB, which implements the entire required functionality. This is achieved by using SQL code for directly accessing the DB. Thus, a separate SQL call is made to the DB each time information is requested from this redundant component. The first redundant component is referred to as the *session only* variant. In the second redundant component design, a session facade strategy is used to implement the requested functionality. Namely, a Stateless Session EJB uses an Entity EJB as means of interacting with the DB. In this case, the Entity EJB acts as a local cache for data in the remote DB. Consequently, data is only retrieved from the remote DB once, at the beginning of each client session, and is then readily available from the local Entity EJB instance. The second redundant

[1]Sun Microsystem's J2EE-Enterprise JavaBeans Technology: java.sun.com/products/ejb

component is referred to as the *session facade* variant. A client Stateless Session EJB is used for calling the currently active redundant component, repeatedly requesting information.
The testing platform for the example application is depicted in Figure 5.1. Three distributed stations were involved, connected by an Ethernet LAN, as follows. The EJB sample application was deployed on an IBM WebSphere application server, running Windows2000 on an Intel Pentium4, with 1.6GHz CPU and 512 MB RAM. The DB2 relational DB was used, installed on a platform with Windows2000, Intel Pentium 4 processor, 1.6 GHz CPU and 256 MB RAM. A third machine was used for generating traffic and loading the network link to the remote DB, to various degrees. The Tfgen traffic generator tool[2] was used for this purpose. The three machines were connected through a switched 100 Mbps Ethernet LAN, completely separated from other traffic.

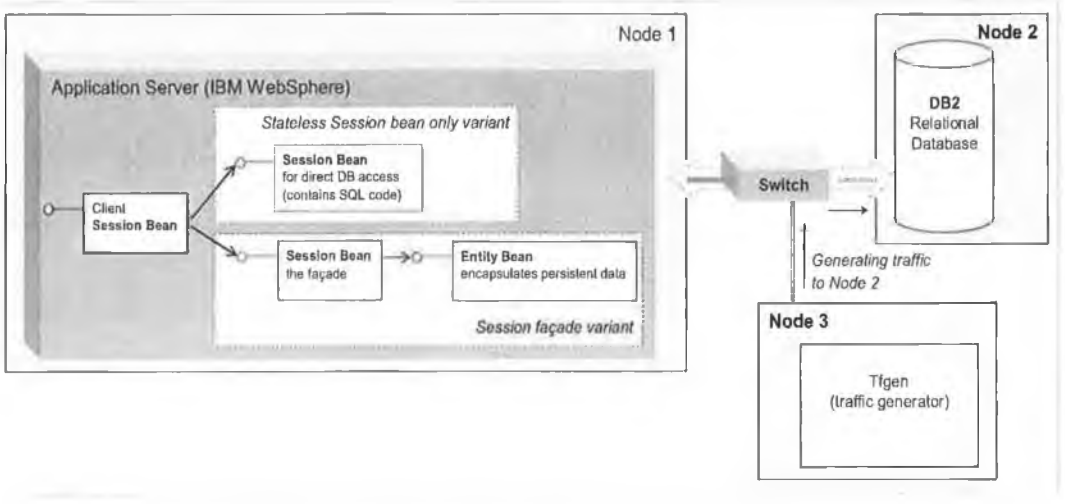

Figure 5.1: testing platform for example EJB application with two redundant components

The response delays of each redundant component were repeatedly measured, in different environmental conditions and for various usage patterns. Variations in the environmental conditions were caused by changes in the amount of available bandwidth resources on the network link to the remote DB. Usage patterns variations were caused by changes in the number of information-retrieval requests per client transaction. Measured performance values indicated certain performance characteristics for the two redundant components, as follows. When the network link to the DB was lightly loaded, smaller delays were experienced in the *session only* variant than in the *session facade* variant. This situation occurred regardless of the number of repetitive information-retrieval requests per client transaction. Namely, cases for 1, 10, 100 and 1000 read requests per client transaction were tested, with similar outcomes. This result is accounted for by the overheads incurred by the *session facade* variant, because of its extra Entity EJB management and inter-EJB communication costs. Nonetheless, increasing the load on the network link to the remote DB had a significant impact on the performance of the *session only* approach. On the contrary, increased network loads hardly affected the *session facade* variant. The reason is that the *session only* variant needs to access the DB for each individual

[2]TFGen traffic generator: www.st.rim.or.jp/ yumo/pub/tfgen.html

information-retrieval request. In contrast, the *session facade* variant requires a single DB access per client transaction, for the first information-retrieval request in each transaction. The reason is that data is cached in the local Entity Bean instance after the first DB read operation. Locally cached data is subsequently retrieved from the Entity EJB instance for succeeding requests. Based on these facts, under certain environmental conditions, the *session only* design choice produces higher delays than the *session facade* variant does. Such environmental conditions are characterised by increased numbers of information-retrieval requests and significant network loads, such as over 90% loads. Using an Entity EJB to locally represent remote DB information becomes in these circumstances the optimal choice. The optimal redundant component cross-point between the two implementations is reached when the inter-component communication and CPU overhead in the *session facade* variant is exceeded by the repeated remote DB access overhead in the *session only* variant.
Figure 5.2 shows the response time curves corresponding to the two redundant components, for various network loads, when 1000 information-retrieval requests were made per client transaction. For obtaining these curves, request response times were repeatedly measured for different network loads. The average response time value for each network load level was then calculated and represented on the response-time graph, as shown in Figure 5.2.
The obtained test results indicate that an informed alternation of the two redundant components, each one optimised for a different execution context, would provide better overall performance than either component alone could provide.

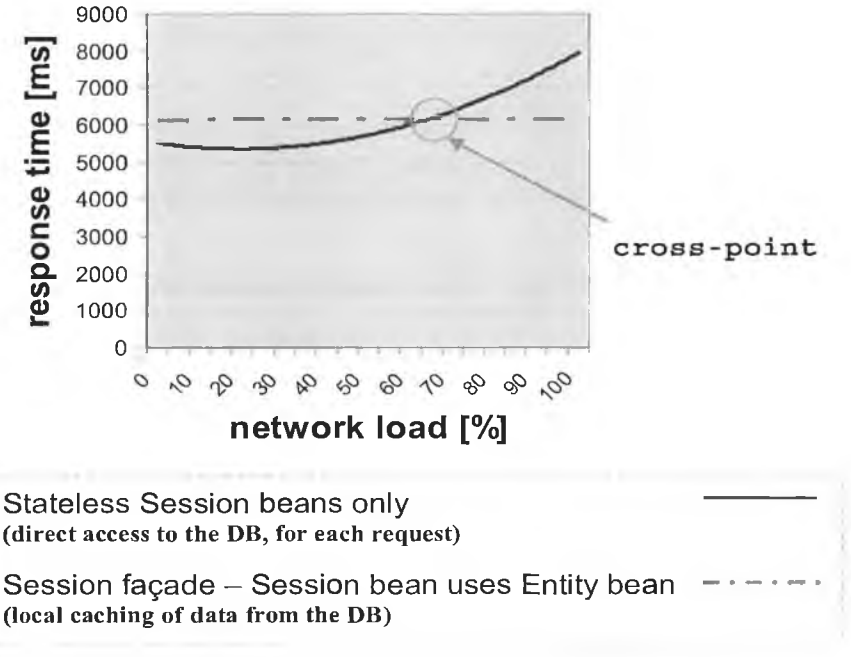

Figure 5.2: response-time variation with network load
for two redundant components - example scenario

Even though simple, this example provides a valid demonstration of how the redundancy-

based approach can be used to dynamically adapt and optimise systems.
The problem addressed concerns the difficulty of devising a single component that exhibits optimal characteristics in all possible running contexts. The optimal component implementation and configuration highly depends on the component's execution environment, which can frequently change. Therefore, the redundancy-based optimisation approach provides a flexible means of utilising the optimal component implementation and configuration under each distinctive execution environment. The redundant component behaviours optimal in different execution contexts are pre-coded and made readily available for the system's use, during runtime. Each redundant component is used in the specific execution context it was optimised for. Due to significant system complexity, it is highly desirable that the redundant component selection process is performed by an automatic system manager, without the need for human intervention. Consequently, an important factor in this solution becomes the adaptation logic for automatically determining the optimal redundant component and optimal combinations of redundant components in each running context.

5.2 Memory Consumption Variations with Incoming Workloads Scenario

5.2.1 Duke's Bank Sample J2EE Application

A second example application was obtained and tested to validate the thesis research. The purpose of this second example is twofold. First, the application used provides a second example in which different redundant components alternately provide optimal performance under different execution contexts. Redundant components in this example differ at their configuration level. This is different from the application presented in the previous section, in which redundant components differed at the implementation level. Second, the example was used to test and prove the applicability of the AQuA framework and its prototype implementation for dynamically optimising system performance.
An enterprise banking application, Duke's Bank [3], was used to demonstrate AQuA_J2EE's performance management capabilities. Duke's Bank is a sample J2EE application from Sun Microsystems. It provides functionality that allows customers to perform banking operations online. Such operations include accessing account histories and performing banking transactions. Administrators can also use Duke's Bank application to manage customer records and accounts.
Duke's Bank is designed as a typical three-tier enterprise application, with web, application and DB tiers. Figure 5.3 provides a high-level view of Duke's Bank's architecture, depicting the main J2EE components involved and the way they interact with external clients and with the DB. Duke's Bank comprises three main business entities: the customer, the account and the banking transaction. Each of these business entities is represented by a separate Entity EJB in the application tier and by a corresponding table in the relational DB. Namely, information on customers, accounts and banking transactions is persisted in a DB and accessed via Entity

[3]Duke's Bank sample J2EE application from Sun Microsystems: java.sun.com/j2ee/tutorial/1_3-fcs/doc/Ebank.html

EJBs with Bean-Managed Persistence (BMP). Stateful Session EJBs are used to handle client sessions in Duke's Bank. The EJB components in the application tier are in turn accessed via web components in the presentation tier. Web components employed include JSP files and servlets.
Duke's Bank is designed so that all Entity EJBs are accessed via Stateful Session EJBs (Figure 5.3). Conforming to the EJB specification, instances of Stateful Session EJBs maintain their state for the entire duration of the client session accessing them. Consequently, in Duke's Bank, the Entity bean instances must also be maintained available for the entire duration of the client sessions using them. This implementation detail had an important role in the outcome of the performed test cases and presented adaptation scenarios.
Duke's Bank application and the way it was modified for the performed tests are presented in more detail in Appendix C.

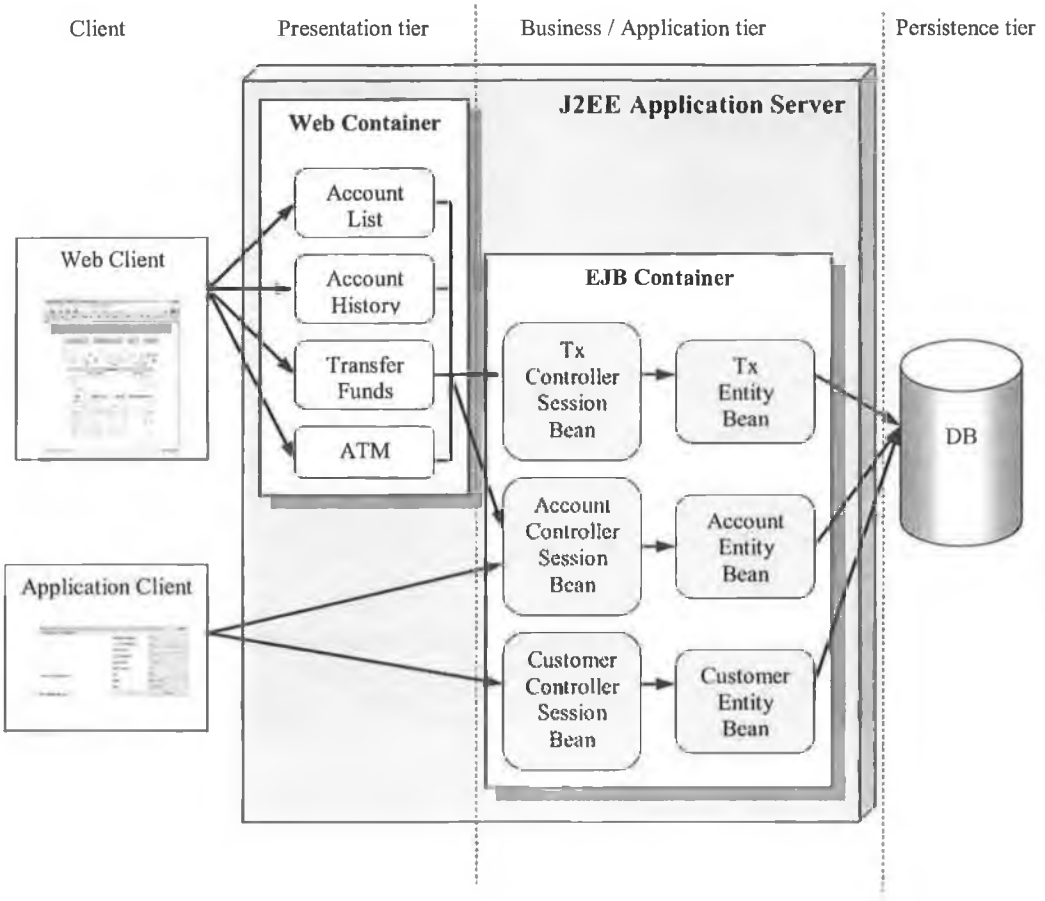

Figure 5.3: high-level architecture of Duke's Bank application

In the original Duke's Bank distribution, Entity beans were implemented so as to use Bean Managed Persistence (BMP). BMP means that the bean implementation is responsible for managing the bean's state, during runtime. This is usually achieved by introducing SQL statements in the bean code, for managing connections to the DB and performing the necessary reading and writing DB operations. An alternative possibility for handling bean persistence is to use Container Managed Persistence (CMP). When CMP is used, the EJB container is re-

sponsible for managing the Entity bean's state and its corresponding persistent representation in the DB. In this case, the EJB provider needs to specify the bean attributes that are part of the EJB's persistent state. Subsequently, the EJB deployer must map the EJB attributes to the corresponding fields in the persistence storage used. Relational DBs are commonly used to provide persistence support for Entity beans. In such cases, an Entity bean is typically mapped to a DB table and the bean's attributes to corresponding fields in that table.
The Entity beans in the original Duke's Bank distribution were modified so as to use CMP instead of the original BMP option. The CMP option grants the container more control over the managed Entity bean instances. Consequently, using CMP simplifies the container-level implementation of the dynamic component-swapping functionality. Undoubtedly, it is also possible to implement the component-swapping facility so as to support runtime replacement of Entity beans with BMP. However, a fully-functional component replacement function that worked for all EJB types and configurations was out of the thesis scope. In addition, the latest J2EE implementation releases from most providers (e.g. starting with JBoss 3.2.x) supply CMP capabilities that are typically more robust and provide better performance than most custom BMP implementations could. In other words, building a BMP Entity bean that is better in certain respects than its equivalent CMP version would prove in most cases a highly-expensive and risky task. As such, BMP usage is generally no longer encouraged.

5.2.2 Database Settings for Duke's Bank

The MySQL[4] relational DB was selected as the persistence storage for Duke's Bank. Separate tables were created in the DB to store data for each of the three main business entities in the bank: customer, account and banking transaction (Appendix C). In turn, Duke's Bank uses three Entity beans to represent these business elements at the application level. The DB tables were populated with initial testing data as follows. The `customers` table was populated with the details of 1000 different customers. Each customer had one banking account opened and stored in the `accounts` table. For each opened banking account, 100 transactions were recorded in the `transactions` table.
Considering this data, it can be deducted that for each user that runs the testing scenario, the server needs to use 1 Customer, 1 Account and 100 Transaction Entity bean instances. These instances are stored in the corresponding container caches of the Customer, Account and Transaction Entity EJBs, respectively. For example, the Account Entity bean instances are placed in the cache of the container that manages the Account Entity EJB. New instances need to be put in the cache for each new user accessing the application. However, for a certain period, instances employed by previous users remain in the cache even after no longer used or needed. The `maximum-bean-age` configuration of a cache dictates how fast unused instances are to be removed from the cache (Appendix B.1). Thus, this configuration critically influences the number of EJB instances stored in a cache at any moment in time.

[4]MySQL opensource database: www.mysql.com

5.2.3 JBoss Caching Configurations

JBoss[5] was used as the J2EE application server for deploying and testing Duke's Bank. Various caching configurations were used during the performed tests to optimise the application performance under different execution environments. This subsection describes the way the EJB instance caching works in JBoss. In addition, some of the possible caching configurations relevant for the presented testing scenarios are also explained. More details on this topic are available from Appendix B.

In JBoss, an instance cache is a repository that stores instances of stateful EJBs between subsequent client requests, as well as whilst handling client requests. Stateful EJB instances maintain their state for longer than one client call and can be Stateful Session beans or Entity beans (section 2.4.2). In short, Stateful Session beans maintain their state between subsequent client calls, for the entire duration of a client session. Once the client session terminates, the instance state is deleted. Entity beans additionally maintain state between subsequent client sessions. Any stateful EJB instance has a unique identity associated with it. Clients use this identity for pointing out the precise instance they want to use. Stateful Session bean instances are identified by their client session ID. Entity bean instances are identified by a unique key attribute value.

When certain stateful EJB instances are being regularly used, the JBoss container maintains the instances in a cache for resource-saving considerations. In addition, JBoss also requires instances of stateful beans to be present in the cache for the entire duration of their use. This means that a JBoss instance cache must be large enough to accommodate all EJB statefull instances required for use at any one time. It also means that a statefull EJB instance cannot be removed from the cache while in use. This is a particular caching strategy adopted by JBoss and may not be valid for other J2EE servers.

The EJB caching configurations relevant for the experimental work are summarised below.

- *min-capacity*: the minimum number of EJB instances that should be available in the cache
- *max-capacity*: the maximum number of EJB instances that can be stored in the cache. If the cache is full and yet more instances need to be added for use, the system will suffer performance degradations, or no longer function
- *max-bean-age*: the maximum period for which a bean can be inactive before being passivated by the overager process and removed from the cache. When an EJB instance is passivated, its state is saved into the persistence storage used (e.g., a relational DB, or a file system). The freed instance is subsequently returned to the JBoss container instance pool.
- *overager-period*: the period between subsequent runs of the overager task. The purpose of the overager task is to see if the cache contains bean instances with an age greater than the max-bean-age element value. Any beans meeting this criterion will be passivated.

In order to call the EJB components' functionalities, clients must first obtain an instance of the EJB component they want to use (subsection 2.4.3). In JBoss, clients obtain instances of an EJB component by sending a request to the JBoss container that manages that component. When a

[5]JBoss opensource J2EE application server: www.jboss.org

statefull EJB instance is requested, the container performs the following operations in order to acquire and return the instance. If the required EJB instance is available from the cache, then it can be directly used from there. Otherwise, an EJB instance is obtained from the instance pool first. If the pool is empty, a new EJB instance is created. Instances stored in the instance pool have no identity associated with them and thus can be used for any client. Once obtained, the EJB instance is placed in the cache and a unique identity is associated with it. Additionally, in the case of Entity beans or passivated Stateful Session beans, the current state of the required EJB instance is brought from the persistence storage and associated with the instance. A reference to the statefull EJB is finally returned to the requesting client. The EJB reference can be used to call on the EJB's methods. The EJB instance is subsequently maintained in the cache, so as to conform to the JBoss container caching configurations. As such, if a statefull EJB instance is inactive for a certain period, equalling the maximum-bean-age cache setting, then the container passivates the instance. This involves removing the instance from the cache and saving its state in a temporary storage location.
A statefull EJB configured with a maximum bean age that is insufficient for its running conditions will not function properly on JBoss. This is because the container will try to passivate the instance while individual clients might still need to use it as part of their sessions. This problem occurs for example for Stateful Session beans, or Entity beans used by Stateful Session beans. The reason is that in such cases, EJB instances may be locked as part of a client session. Thus, they cannot be passivated and deleted from the instance cache. Heavy workloads or limited resources can cause EJB instances to stay idle for long intervals. The EJB instances can be locked waiting for responses from other EJBs to be returned, or for needed resources to become available. Consequently, the EJB container will detect the instances' inactivity, consider the instances are no longer needed and try to passivate them. However, as EJB instances cannot be passivated when they are locked in a running session, resources are wasted trying to perform an illegal activity. This in turn induces further delays in processing client requests, finally causing transactions to expire and be rolled-back. Performance can drop dramatically, sometimes even causing the application to stop functioning properly. These considerations may determine deployers to configure EJB caches with long maximum bean-age values. However, under light workloads, client sessions may take much shorter times to execute. In such cases, the EJB instances' lack of activity would correctly indicate that the instances are no longer needed and could be safely passivated to save memory resources. At application deployment time, the exact runtime loads and application usage scenarios are not known. Furthermore, such running conditions can repeatedly and significantly change over time, meaning that no single deployment configuration can be optimal at all times. Considering this, a number of redundant components were developed for the thesis experimental work. Each redundant component was optimised for different system loading conditions.

5.2.4 Redundant Components for Duke's Bank

Several redundant components were built and used for enabling Duke's Bank to adapt to changes in its running environment. The redundant components differed in their deployment configurations, which instructed JBoss containers on how to manage instances of the EJB components at runtime. More precisely, the `max-bean-age` parameter was tuned for each instance cache of each redundant EJB, so as to be optimal in certain system-load condi-

tions. The `max-bean-age` parameter dictates the amount of time an inactive EJB instance is kept in a cache before being *passivated*. Passivating an EJB instance involves saving its state in a persistent storage and removing the instance form the instance cache. This consequently frees the caching resources used to manage the passivated EJB instances.
The `max-bean-age` setting is used to indicate the time to wait before concluding that an inactive component is no longer being used. At this point, the EJB instance can safely be passivated so as to save system resources. Nonetheless, in a highly loaded system, an EJB instance can remain inactive for long periods, even while actually handling client requests. This can happen as the EJB instance may be blocked waiting for responses from other EJB instances, or for needed resources to become available. In such cases, if the EJB instance remains inactive for longer than its `max-bean-age`, JBoss will rightly attempt to passivate it. However, as the EJB instance is being locked in a client transaction or session, the passivation operation will fail. Additional resources are consumed while JBoss attempts to perform illegal operations, further increasing delays and worsening resource contention. Performance consequently deteriorates until transactions start to expire and roll-back. Exceptions are consequently thrown causing system availability to degrade. To avoid such undesirable situations, application deployers commonly configure EJB instance caches with extended maximum-bean-ages, which will most certainly suffice in eventual heavy-load scenarios. As an example, the standard JBoss configuration for the `max-bean-age` parameter is 600 seconds. However, when the system is lightly loaded, extended `max-bean-age` configurations mean that EJB instances are kept in the cache for long periods, even if typically no longer reused or needed by the application. Memory resources are being inefficiently used in effect. This is a clear example scenario where an application's optimal configuration directly depends on the application's execution context.
Ideally, the managing application server (e.g. JBoss) would simply 'know' how to differentiate between the two scenarios and be capable of deciding to passivate EJB instances only when 'really' inactive and no longer needed. For achieving this goal, the application should be dynamically reconfigured when its execution environment changed. AQuA_J2EE provides the means to automatically execute such adaptive management operations, at runtime. Redundant components with different caching configurations are used to support the application's dynamic optimisation. The policy-based detection, evaluation and decision functionalities allow system managers to specify, in a platform-independent manner, the difference between the various execution scenarios and the possible corrective actions to be taken in each situation.
Based on these considerations, two redundant components were employed in the presented tests on Duke's Bank. Namely, redundant components were used for the Entity EJBs employed in the tested usage scenarios. Each redundant component was configured so as to be optimal under a different system load range. The redundant components were built to differ in their instance caching configurations. Namely, the caches were configured with 10 second and 500 second maximum-bean-ages. These redundant components are referred to as the *10-bean-age* component and the *500-bean-age* component, respectively.
More than the two presented redundant components were actually developed and tested as part of Duke's Bank. Nonetheless, for the targeted range of running conditions the two presented component variants were determined to suffice, for providing close to optimal performance at all times. Conforming to the performed tests, using more than these two variants would have produced insignificant additional gains in performance, at the cost of inducing

unnecessary adaptation overheads. Redundant components such as the ones presented are created at deployment time, when the deployment platform and expected running conditions are known. Further redundant components can also be created and added during system runtime, in order to handle unexpected execution conditions. For example, additional redundant components can be created in the same manner, to be optimal under system loads larger than the ones used in the tests presented.

The adaptation strategy used to optimise Duke's Bank in the presented test case was based on dynamically tuning the caching configurations of the application's Entity beans, according to changes in the system load. The maximum bean-age of the cache was the principal tuning parameter for the performance optimisation. The used adaptation strategy was derived from knowledge on the operation of the JBoss containers and their provided services (e.g. EJB lifecycle management, instance pooling and caching). Assumptions were made on the way the various JBoss container configurations could be tuned, so as to optimise system performance under certain execution contexts. These assumptions were then tested and verified based on the obtained results. Test results were subsequently used to determine and validate the particular caching configurations that were optimal under each system loading conditions.

The optimum maximum bean-age value is influenced by several factors. First, the system load highly influences the EJBs' response times and thus the correct interpretation of idle EJB periods. Namely, the incoming load and the amounts of available resources dictate the length of client sessions. They also determine possible bean inactivity periods within these sessions. Bean instances are being locked for the entire duration of the session that uses them. Hence, the maximum bean-age of a cache should be larger than the apparent inactivity periods during which bean instances are being locked. Otherwise, the container will unsuccessfully try to passivate and remove locked instances.

Another important aspect to consider when tuning redundant component configurations is the application's usage patterns. Usage patterns, or work mixes, indicate the frequency at which the same clients return to access an application, in a way that requires the same bean instances to be used. This aspect dictates whether it is worth maintaining EJB instances in the cache, after the session that uses them terminates. The application's business logic may have a high influence on this aspect. Thus, in a banking application, such as Duke's Bank, users rarely return to manage their bank accounts repeatedly, at short time intervals. In addition, each individual user generally has their own banking accounts and transactions, which should never really be accessed by other users. This implies that the Entity EJB instances required by one individual user will not be needed for handling requests from other users. Hence, in this case, Entity bean instances should be removed from the cache as soon as possible, once the client session they were involved in terminates. However, no attempts should be made to remove bean instances from the cache while still in use by a client session. Based on these considerations, the redundant components built for the presented Duke Bank's tests were tuned so as to be optimal under their targeted ranges of system loads. In contrast, in an e-commerce application for example, multiple users are commonly accessing the same products, whether for viewing or purchasing them. Consequently, the Entity bean instances representing these products at the EJB application level are being reused across multiple users. In such cases, it may indeed be beneficial to maintain certain Entity bean instances in the cache for more than the duration of a single client session. The particular characteristics of each managed application are typically known to system administrators. Thus, administrators can accordingly configure the redundant components used and the associated application adaptation strategies.

The optimal JBoss configuration values also depend on the general characteristics of the business data stored in the persistent DB. Such characteristics include the amounts of data to be processed, as well as the relations between the various data items. In the performed tests on Duke's Bank application example, the average number of banking accounts per customer and the common number of banking transactions per customer account had an important influence on the optimal redundant component configurations. Configurations impacted by these aspects included workload cross-points (or thresholds) and caching parameter values.

5.2.5 Test Platform

Three stations were employed for installing the J2EE application and performing the tests. One station was used for running Duke's Bank application on JBoss, a second for running a relational DB and a third for simulating client activity on the tested system. JBoss 3.2.5 was used as the J2EE application server, running on a Microsoft Windows Server 2003 Enterprise Edition platform, with Intel Pentium III at 860MHz and 512 MB of RAM. The MySQL relational DB was selected as persistence support for Duke's Bank application. The DB was run on Microsoft Windows Server 2003 Enterprise Edition, on an Intel Pentium III processor at 866MHz and 128 MB of RAM. The OpenSTA [6] load-generating tool was used to simulate clients for the tested application. OpenSTA was installed on a Windows Server 2003 Enterprise Edition station, Intel Pentium III at 701 MHz and 1 GB of RAM. The three stations were connected via an Ethernet LAN at 100Mbps. The GCViewer[7] tool was used to visualise JVM-level memory consumption data, for the Java process which ran the J2EE application and JBoss server.

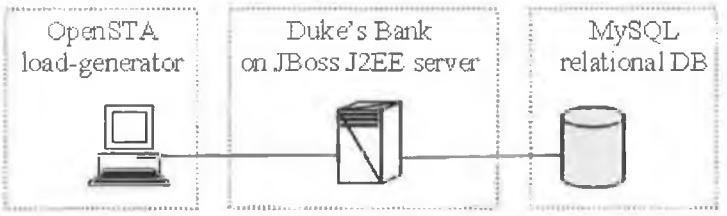

Figure 5.4: testing platform for Duke's Bank

5.2.6 Test Scenarios and Procedures

Duke's Bank was tested under two different workload conditions, starting with a low workload of 15 concurrent users and continuing with an increased workload of 60 concurrent users. After a certain period, the workload was decreased back to the initial lower workload of 15 users. The OpenSTA load generator was used to simulate the varying incoming workloads and user behaviour on the tested J2EE application. Thus, OpenSTA was configured to generate different numbers of concurrent users accessing Duke's Bank application. Each simulated user interacted with Duke's Bank following a certain well-defined usage scenario. An OpenSTA script was used for specifying the client behaviour to simulate (Appendix D). The usage

[6]OpenSTA - the Open, Systems Testing Architecture: www.opensta.org
[7]GCViever - data visualisation tool for JVM-level measurements: www.tagtraum.com/gcviewer.html

pattern run by each client involved several operations, as follows. First, the client logs in, lists all the banking transactions of a selected account and finally logs out, terminating the client session.
The OpenSTA script defining clients' behaviour was configured so that each user received a unique identity (Appendix D). This means that each generated client logged in as a separate bank customer, with different personal details and banking accounts. In turn, this configuration implies that once a client's session was completed, the EJB instances used to serve that client's requests were no longer needed. At that point, these EJB instances should be promptly removed from the cache, to free system memory.
Duke's Bank application was tested in two different scenarios. In one scenario, the automatic application adaptation was *not* used during runtime. This means that AQuA_J2EE was not integrated with JBoss for the performed test, during this first scenario. In the second scenario, the automatic application adaptation capabilities of AQuA_J2EE were employed to optimise Duke's Bank to changes in its running environment.

5.2.7 Test Results

In the testing scenario that used AQuA_J2EE, the application was initially configured to run the 10-bean-age variant, as optimal for the initial low workload. When the workload increased, the management framework automatically detected the load variation, determined that the 500-bean-age variant was the optimal one in the current environment and decided to activate it. Consequently, the initial 10-bean-age variant was dynamically swapped with the new 500-bean-age variant. The system adaptation was triggered and carried out automatically, without any human intervention.
During the interval immediately following this adaptation operation, the special-purpose policies in the adaptation decision module prevented further adaptations from being performed, so as to avoid oscillating adaptations (subsection 4.6.3). Such a situation would have occurred in this case due to the decreased workload detected on the adapted components, during the actual component-swapping process. The workload decrease was caused in this case by the component activation process, which blocked requests on Redundancy Groups (RGs) while swapping their redundant components. Nonetheless, the detection module is unaware of this aspect when interpreting monitored data. It consequently alerts the evaluation module, which determines the optimal redundant component in the apparent low workload context. Activating this recommended redundant component at this point would constitute undesirable management behaviour. The reason is that the application adaptation would actually be based on monitoring data collected during another adaptation process which is optimising the application for the real, increased workloads. The current implementation of the adaptation decision policies prevents this incorrect behaviour by not allowing adaptation operations to be performed within a certain interval after a system adaptation was completed.
During the AQuA_J2EE-enabled testing scenario, when the workload later decreased back to a low level, the application was automatically adapted again, in a similar manner, so as to reuse the initial 10-bean-age variant.
In the scenario in which the application was *not* adapted, the 500-bean-age variant was used throughout the test, workload fluctuations ignored. Results obtained during the two tested scenarios are presented in Figure 5.5.

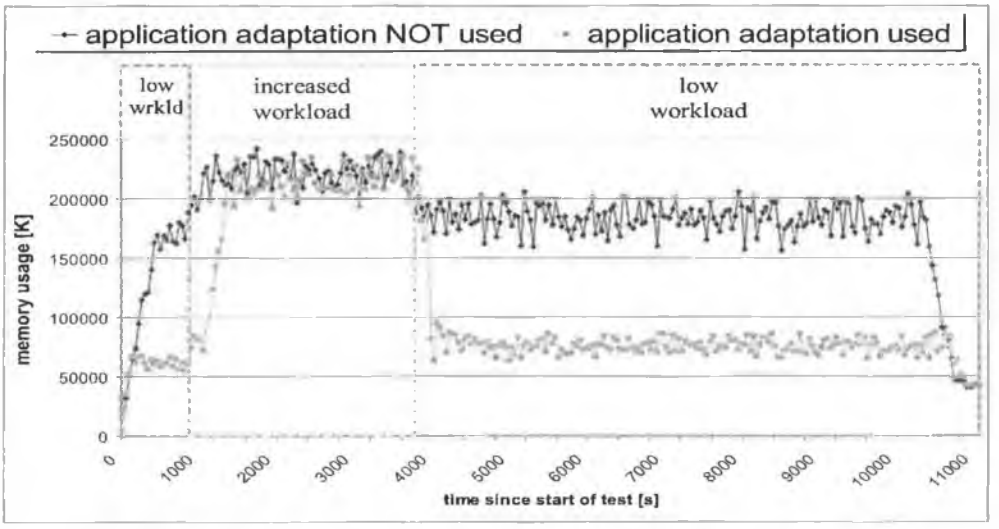

Figure 5.5: adaptation impact on memory usage

Figure 5.6 summarizes the memory-usage characteristics of the two redundant components under the low and increased workload conditions. The results indicate that the 10-bean-age component consumes much less memory than the 500-bean-age component, when executed in low workload conditions. Nonetheless, the 10-bean-age component cannot be used under high workloads. On the contrary, the 500-bean-age variant can be used under both workload ranges, but is suboptimal when executed in low workload conditions. The impact the two redundant components had on the system memory usage is discussed in more detail below.

redundant component / workload	10-bean-age	500-bean-age
low (15 users)	60,000 – 80,000 [K]	160,000 – 200,000 [K]
increased (60 users)	exceptions	200,000 – 240,000 [K]

Figure 5.6: different redundant components are optimal under different workloads: the 10-bean-age component is optimal under low workloads but cannot be used under increased workloads; the 500-bean-age component can be used under both workload ranges but is sub-optimal under low workloads

When using the 10-bean-age component and testing the application under the low 15 user load, the maximum number of instances reached in each cache was very close to the actual number of required instances, which is 15 Customer, 15 Account and 1500 Transaction Entity EJB instances. However, this was no longer the case in the non-optimised application, when

the 500-bean-age component was used under the same low 15 user workload. In this case, the maximum number of EJB instances in each cache was considerably increased, reaching up to five or six times the number of needed Customer, Account and Transaction Entity EJB instances. The number of EJB instances in the JBoss' container caches directly impacted system memory usage, as confirmed by the obtained test results.

Figure 5.5 shows the memory usage levels recorded during the two separate test scenarios. Results clearly indicate the benefits of the application adaptation on the system memory usage. Under low system loads, the non-adapted application consumed more than 100% more memory (i.e. 100MB more memory) than the adapted application. The measured memory consumption represents the memory usage of the entire web and application tiers of the enterprise system. This includes both Duke's Bank application and the JBoss server. Thus, the memory usage gains are reported with respect to the memory consumption of the entire enterprise system (DB excluded).

As previously explained, the reported gains were obtained based on the way the different caching configurations work on JBoss. As such, when using the 10-bean-age component and testing the application under a low load, the maximum number of instances reached in each EJB cache was close to the number actually needed for handling client demands. However, when using the 500-bean-age component under the same load, the maximum number of instances in each cache was significantly increased, as EJB instances were being maintained in the cache for long periods, even after no longer used. In a real banking application, each customer has their own banking account, which they normally manage once a day, at most. Thus, instances cached for a certain customer are never actually used again before passivation. As a result, under low system loads, the memory consumption caused by keeping the caches at increased levels, as in case of the 500-bean-age variant, do not bring any visible performance benefits.

In order for the system to be able to run under the two tested environments, sufficient system resources needed to be available to accommodate both low and increased workloads. For this reason, the memory gains obtained by adapting the application to low incoming workloads would not directly improve the application's performance characteristics (i.e. response time and throughput). However, a realistic scenario in which such gains would be beneficial is that of a cluster of servers on which multiple applications are being run; applications are dynamically being ported between the available servers in the cluster, so as to cope with fluctuations in the incoming workloads, optimise cluster resource usage, or mask server crashes (e.g. [96]). In this scenario, saving memory on one of the cluster servers would allow for a memory-consuming application to be ported on that server. This scenario was simulated in the executed tests by starting a memory-consuming application whenever sufficient memory became available. The memory-consuming application was simulated by starting a java application and setting its JVM memory allocation parameters to the desired memory consumption value (e.g. `java resource_consumers.MemoryConsumerApp -Xms150m -Xmx150m`). This application was set to consume about 150MB of memory. In this scenario, the memory saving benefits could be observed at the performance level of Duke's Bank. Namely, when Duke's Bank was adapted to use the 10-bean-age variant under low user loads, running the memory-consuming application in parallel did not impact on Duke's Bank performance, as sufficient memory was available. However, attempting to start the memory-consuming application when the 500-bean-age variant was used resulted in out-of-memory exceptions being raised, causing the JBoss server to crash and thus dramatically

affecting system availability. This shows how adapting Duke's Bank can optimise memory usage in a clustered system with limited available memory. When the tested system was upgraded so as to avail of sufficient memory for the correct functioning of both the non-optimised Duke's Bank (i.e. using the 500-bean-age variant) as well as of the memory-consuming application, availability issues were solved, but response times of Duke's Bank were impacted. Figure 5.7-a shows the response times measured in this final scenario for the two redundant components running under low loads. Results indicate that when the 500-bean-age variant is run, certain users experience response times of up to 20% (i.e. 4 seconds) bigger than when the 10-bean-age variant was used.

Response times measured during the entire duration of the two testing scenarios are shown in Figure 5.7-b. The original JBoss distribution was used when the application adaptation was not used. Thus, the presented results indicate that during normal system execution AQuA_J2EE induces no visible overheads on application performance. Performance overheads occur only during the actual application adaptation process. This is reflected in the two spikes that appear in the response time values, at the points where the two swapping operations occurred. As previously discussed, the response time overheads caused by the component-swapping process are critically dependent on the actual swapping implementation used and on the particular characteristics of the managed application. These overheads must generally be considered when evaluating an adaptation operation, to ensure the potential benefits would outweigh the induced overheads. The impact that the presented solution has on other system quality attributes, such as reliability, should also be considered.

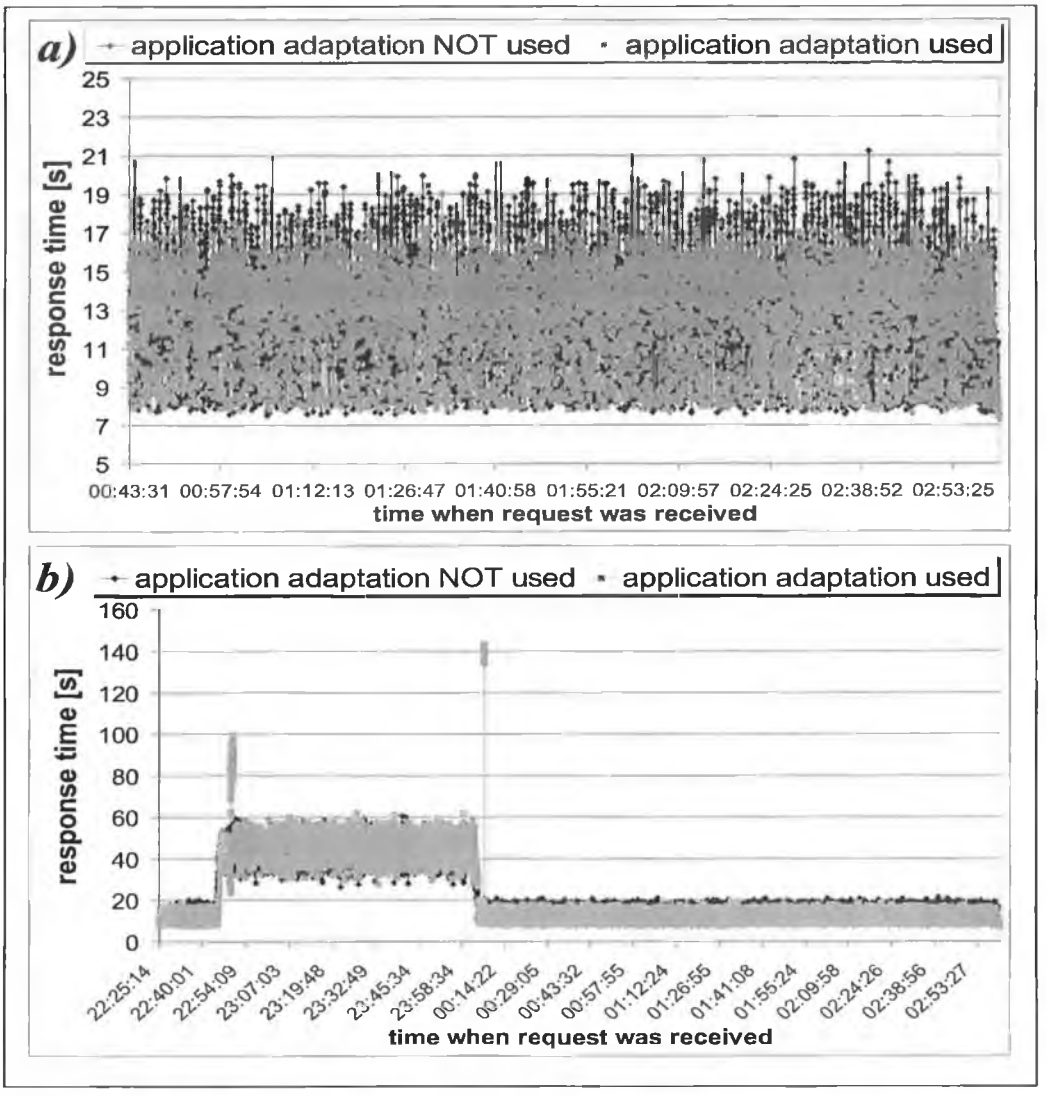

Figure 5.7: application adaptation impact on response times - limited memory availability: a) low workloads b) low and increased workloads

Duke's Bank example offers a valid case in which no single optimal configuration exists for all possible system execution environments. The 10-bean-age component is optimal under low workloads, but cannot be used under heavy loads. The 500-bean-age variant is needed for increased workloads, but is sub-optimal under low workloads.

5.2.8 Discussion

A few notes are in order for showing the way monitored parameter values should be interpreted in the tested scenarios. The discussion is tied to the fact that values of a single parameter often cannot be interpreted completely independently from the values of other parameters. The reason is that certain parameters may be strongly related, their values influencing each other. Monitored values of a certain parameter may have different meanings depending on

the readings obtained from different, related parameters. For example, monitored component-level workloads can directly influence, as well as be directly influenced by system-level resource availability.

In the exemplified Duke's Bank application, the parameter that ultimately dictates which redundant component is optimal at anyone time is the component response time. This is the time required for a component to reply to client requests. A component method's response time is the key parameter in the tested Duke's Bank example. The reason is that the response time provides in this case a decisive indication on the execution context characteristic that critically influence the component's evaluation. More precisely, response time provides a direct indication of the current system load and consequently of how a component's inactivity should be interpreted. Namely, the inactivity of a cached EJB instance can be interpreted differently, depending on the system load (subsections 5.2.3 and 5.2.4). In this context, the 'system load' is used to indicate the system's resource utilisation for handling client requests. Thus, the system load directly depends on the available system resources, as well as on the incoming workload. The system load significantly impacts on the applications' performance characteristics, such as response times and throughputs. Specifically, a component generally yields lower response times and higher throughputs under low system loads than under high system loads. On the contrary, high component response times typically indicate an increased system load. This means that a component may be locked in client transactions waiting for needed software and/or hardware resources to become available. As such, under low system loads, an EJB instance's inactivity typically means that the instance is no longer needed and can be discarded. On the contrary, under high system loads, the same EJB inactivity period can be caused by the instance being blocked waiting for required resources to become available. A component's response times can be used to differentiate between the two cases, as a direct indication of the current system load. Namely, the response times of the various components involved provide a clear indication of the amount of time EJB instances may remain idle in a cache, waiting for the arrival of responses they need to complete their tasks. More precisely, higher response times indicate a loaded system, requiring the 500-bean-age variant to be used. Similarly, decreased response times will indicate a lightly loaded system, allowing for the optimised 10-bean-age variant to be activated. Hence, analysing current response time values for deciding which redundant component to use is a viable option.

Alternatively, it is also possible to analyse those system parameters that directly impact response time values, such as the incoming workloads and the resource availability. For example, if the amount of physical system resources remains constant, and resources are sufficient so as not to become saturated at any point, then the incoming workload can be successfully used to evaluate and predict overall system loads and response time fluctuations. This option was selected for specifying the detection and evaluation policies in the presented tests. Nonetheless, if resource contention occurred, individual component workloads would cease to increase with the actual incoming client load on the system. This is because client requests would be queued (at lower middleware, JVM or OS levels) waiting for resources to be freed. Consequently, queued client requests would not influence the monitored component workloads. In such cases, the combined variations of workloads and system resource usage need to be considered.

Overall, the purpose of collecting and correctly interpreting runtime monitoring data is the ability to accurately identify the exact cause of a detected performance problem. Performance problems are typically caused by the depletion of a limited resource, required for the system's

correct functioning. Bottlenecks are consequently formed causing incoming requests to be queued, thus increasing delays and diminishing throughputs. A correct bottleneck diagnosis often requires the correlated interpretation of multiple metric values. As previously shown, workloads have a direct impact on a system's resource consumption. Thus, measured workloads can be used as a direct indication of the system load and a predictor of the system's performance fluctuations. Nonetheless, if a bottleneck is formed at the system resource level, workload variations may no longer reflect the system load, as incoming requests would be queued and not counted at the monitored component level. A combination of the monitored workload and available hardware resources, such as CPU, memory and network bandwidth should be used in this case to get a clearer picture of the current system state. Nonetheless, monitoring a system's hardware resources may not always suffice. Software-level configurations can also limit the amounts of software platform resources available to a certain application. Such software resources may include the maximum number of accepted transactions on a web server, the total number of allocated threads or processes, or the permitted number of DB connections. A bottleneck at this software resource level can not be detected by solely monitoring workloads and hardware resources. The reason is that as before, measured workloads would stop increasing as a result of incoming requests being queued at lower system levels, waiting for the limited software resources to become available. For this reason, the software resource limitation would not be sensed at the hardware resource level either. The reason is that the queued requests would not consume any processing or communication resources at the hardware level. Thus, in this case, more system metrics should be monitored in order to detect this type of software resource bottleneck. For example, monitoring samples may be collected from the underlying JVM or OS layers, in order to keep track of the software resources available at these levels, including the numbers of free threads, processes, or connections.

The presented Duke's Bank example shows the paramount importance of clearly understanding the way a system's state is reflected, at different levels, in its various parameter values. Being able to correctly interpret the available monitoring data and provide viable system diagnosis is crucial for specifying and obtaining the desired system adaptation behaviour. Further monitored parameters can be added as necessary to provide a more accurate view of the system state. It is important to note that the devised adaptation strategy and associated redundant components were specifically designed for Duke's Bank and the JBoss application server. Thus, an identical performance management strategy and configuration may not produce the exact same performance benefits for different applications and/or on different platforms.

Another important aspect to discuss in the context of Duke's Bank example is the way in which the managerial component-swapping operations impacted on system performance. An important management factor influencing this aspect is the component-swapping operation. The current component-swapping implementation is based on a modified version of the component hot-deployment facility, provided by the application server (subsection 4.7). The particular characteristics of this component-swapping solution have a direct and important impact on AQuA_J2EE's management behaviour, as presented over the following paragraphs. The prototype component-swapping implementation can be updated and improved so as to minimise induced performance overheads. Nonetheless, an optimised fully-functional component hot-swapping implementation was out of the thesis scope.

The current component-swapping implementation in AQuA_J2EE imposes blocking new incoming requests for the entire duration of the swapping operation. This includes the time needed to complete executing client requests that were already being handled when the swap-

ping operation was initiated. In more detail, when an EJB-swapping operation is started, all requests for creating new EJB instances must be blocked. These requests are delayed for the entire duration of the EJB-swapping operation. The delayed requests are only allowed to go through after all the currently running EJB instances have terminated their execution and have been removed from memory, and only after the actual EJB-swapping operation terminates installing the new EJB variants.

The delays caused by this approach directly depend on the running application's characteristics and execution environment. For instance, the incoming workload and the available system resources directly dictate how long currently handled requests take to execute. This in turn directly impacts on the time the new incoming requests have to be blocked, waiting for the EJB-swapping operation to complete. Also important are the type and complexity of the EJBs used to implement the managed application's business logic. For example, using the current component-swapping implementation to replace Stateless Session EJBs will take considerably less time than when swapping Stateful Session beans. This is because Stateless Session EJB instances are only used for handling one client request and can be safely deleted afterwards. Once all stateless instances are deleted, the EJB can safely be swapped for its new redundant variant. In contrast, Stateful Session bean instances are maintained during the entire duration of a client session. They are then additionally maintained in the instance cache for a certain period. Thus, when statefull beans are used, the delays imposed to new incoming calls (during a hot-swapping operation) directly depend on the duration of the currently running client sessions. In addition, the delays critically depend on the EJB's caching configurations. Delays can consequently range from a few seconds to tens of minutes. Additionally, such delays may not only occur while swapping Stateful Session EJBs, but also when swapping Stateless Session or Entity EJBs that are being used by Stateful Session beans. The reason is that these EJBs, even if not tied to client sessions themselves, are nonetheless locked in running client sessions by the Stateful Session EJBs using them. This case occurred when swapping Duke's Bank Entity beans, which were being used by Stateful Session beans. As such, the use of Stateful Session beans in an application may induce increased management delays when using AQuA_J2EE. Nonetheless, the use of Stateful Session beans for implementing business logic is not recommended, especially when performance is an important factor[8].

Considerable improvements to the current component-swapping technique would be achieved if new incoming requests were handled by the new redundant components, in parallel with older requests being handled by the old redundant components. Once the older requests finished executing, deprecated redundant components could be removed (e.g. The JSR 88 from Sun Microsystems: "J2EE Deployment API Specification", November 2003, or [65]).

5.3 Testing the Learning Mechanism

AQuA's learning mechanism (section 3.13) enables the management framework to infer performance information from the collected monitoring data samples. Inferred information on the components' performance characteristics is used to improve AQuA's management be-

[8]The BEA Documentation Source, "Scaling EJB Applications", 1999: edocs.bea.com/wle/wle50/tuning/tsejb.htm

haviour and decision accuracy. The learning function can be executed whether periodically during runtime (as new monitoring data samples become available), or upon request (processing stored monitoring data samples) (section 4.5).
Several initial tests were performed to investigate the expected capabilities of the proposed learning algorithm. Namely, initial procedures were carried out to test the learning algorithm's ability to group and merge similar monitored data samples into clusters of information. The monitored parameters considered were the incoming workload, memory consumption and response times. Though, memory usage monitoring capabilities are not currently integrated with AQuA_J2EE. The performed memory consumption measurements were performed using a special-purpose monitoring tool, the GCViewer. The monitored data samples recorder by AQuA_J2EE consisted of pairs of workload and response time parameter values. Nonetheless, as the response time did not play an important role in the tested scenario on Duke's Bank application, this parameter was no further considered in the presented tests for the learning process. The incoming workload was the unique parameter used to represent environmental conditions in each monitored data sample.
The testing scenario considered was similar to the one used for testing the AQuA_J2EE prototype on Duke's Bank application. More specifically, monitoring data was collected while Duke's Bank was running under two different workloads, corresponding to a low user load (i.e. 1 concurrent user) and a high user load (i.e. 60 concurrent users). The incoming workload used on Duke's Bank for testing AQuA's information inference function is shown in Figure 5.8. This workload was measured on one of the Duke's Bank's component methods (i.e. one of the `MyAccount` EJB's methods). The `500-bean-age` redundant component was used during the performed tests. The recorded workload values ranged between 700 and 800 request per time interval when under 1 concurrent user load and between 7,000 and 9,000 requests per time interval for 60 concurrent users.

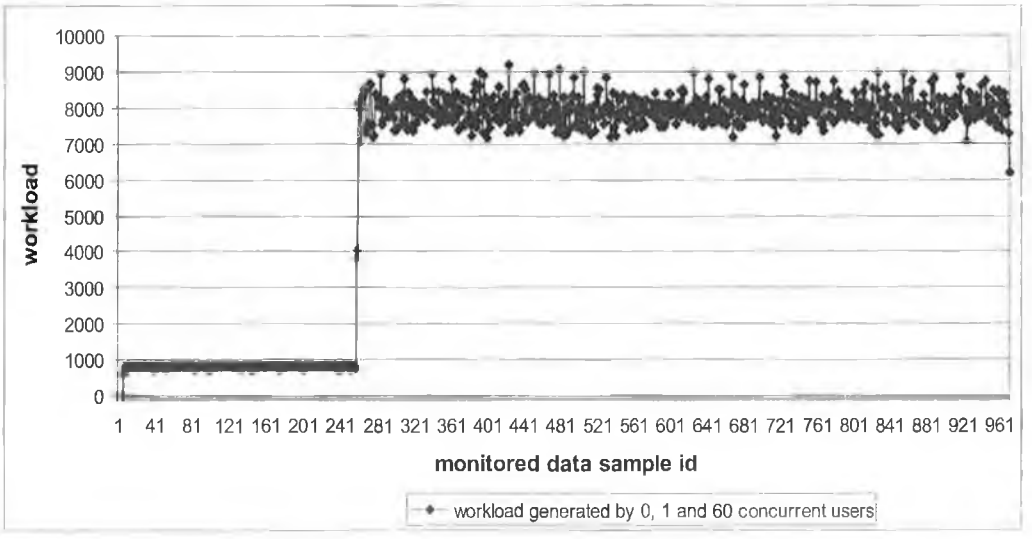

Figure 5.8: workload generated on the MyAccount EJB by 0, 1, and 60 concurrent users, for testing AQuA's learning function

The procedure used to merge similar data samples into clusters of information was described

in sections 3.13 and 4.5. In short, data samples are grouped based on their overall similarity factor (o_SimF), which is calculated based on the samples' respective parameter values. The overall similarity factor is used to assign data samples to clusters. It is also used to calculate the effect the values of a new data sample has on the current information of the existing clusters. The performed tests show how both these two procedures critically depend on the `no similarity interval` parameter. This configuration parameter represents the maximum difference between two data samples for which the samples are considered at least somewhat similar (i.e. $o\_SimFgreaterthan0\%$). If the values of two data samples differ with more than the `no similarity interval` value, then the overall similarity factor between the samples is zero. The performed tests confirmed the learning algorithm's capability of generating information clusters based on a series of monitoring data samples. It also indicated the way cluster formation is influenced by the configuration setting of the `no similarity interval` parameter. This section examines this topic starting from the obtained test results and further discusses the subsequent steps the learning process will perform based on additional monitoring data and implemented capabilities.

The initial value of the `no similarity interval` parameter was set to 100. This value represented the difference between two measured workloads, or numbers of incoming requests per time interval. The overall similarity factor threshold (i.e. o_SimF threshold) was set to 50%. This meant that a new cluster was created if the overall similarity between a new data sample and any existing cluster is smaller than the 50% threshold (i.e. o_SimF smaller than 50%). In the tested scenario, this meant that a new cluster was created if a new workload value differed from the existing cluster centres values with a value greater than 50. The graph in Figure 5.9 indicates the way information was progressively inferred from available data samples, in this initial configuration scenario. It displays the inferred workload values in the order in which they were calculated and the clusters generated based on these values. A total of 36 clusters were created in this case. The final inferred value of each cluster is the last value that occurs in the graph for a certain cluster ID, before a new cluster ID is shown. Similarly, the centre of each generated cluster is the first value that appears on the graph for a certain cluster ID. The end-results of the information-inference process for the aforementioned settings are shown in Figure 5.10. This graph shows the centre values of the generated clusters, the corresponding reliability value of each cluster and the associated cluster information value. A cluster's reliability factor is determined by the number of data samples that were allocated to that cluster and that were used to calculate the cluster's inferred information. For clarity, the generated clusters are shown in the graph in ascending order of their centre values. In practice, the order in which clusters are created, or stored has no effect upon the learning or evaluation processes. Similarly, Figures 5.9 to 5.18 present the analogous test results obtained for various values of the `no similarity interval` parameter, namely for 400, 1000, 4000 and 10000 values respectively.

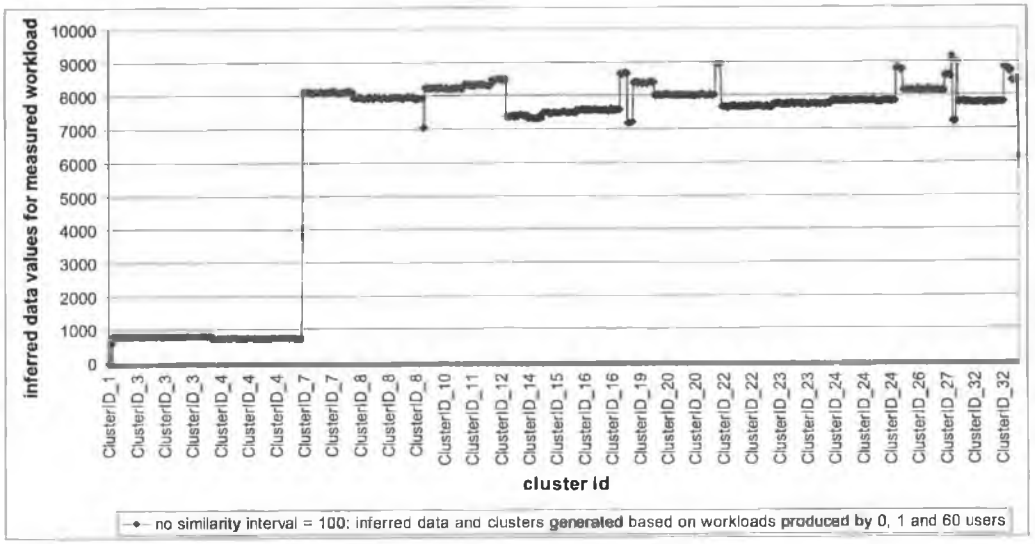

Figure 5.9: inferred workload data values and formed clusters for a no similarity interval of 100

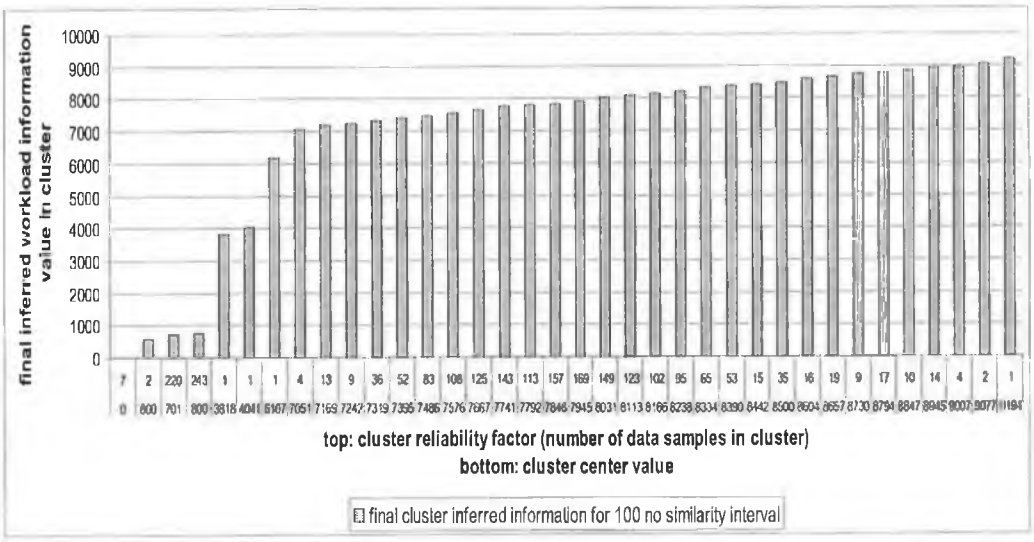

Figure 5.10: generated clusters - centre values, reliability factors and final inferred information, for 100 no similarity interval

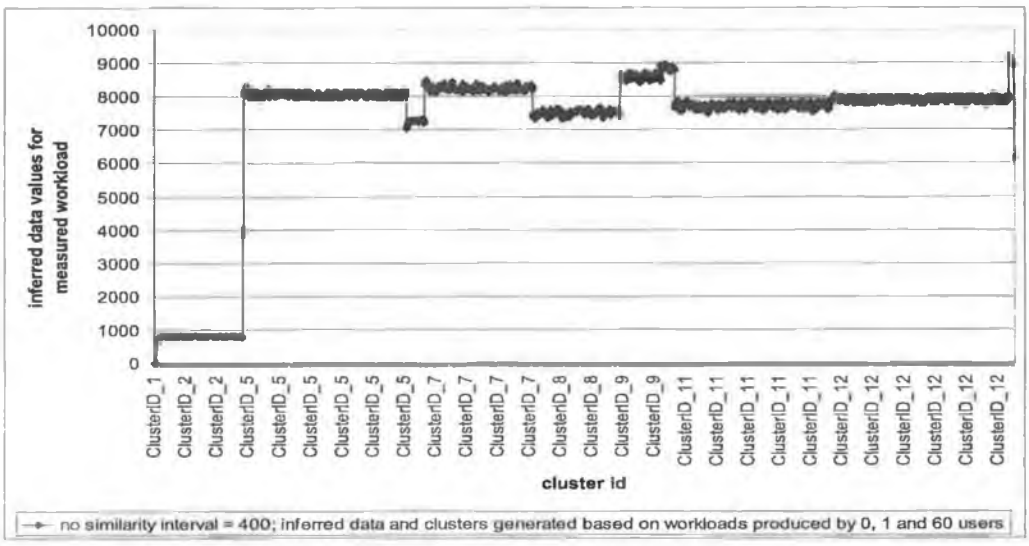

Figure 5.11: inferred workload data values and formed clusters for a no similarity interval of 400

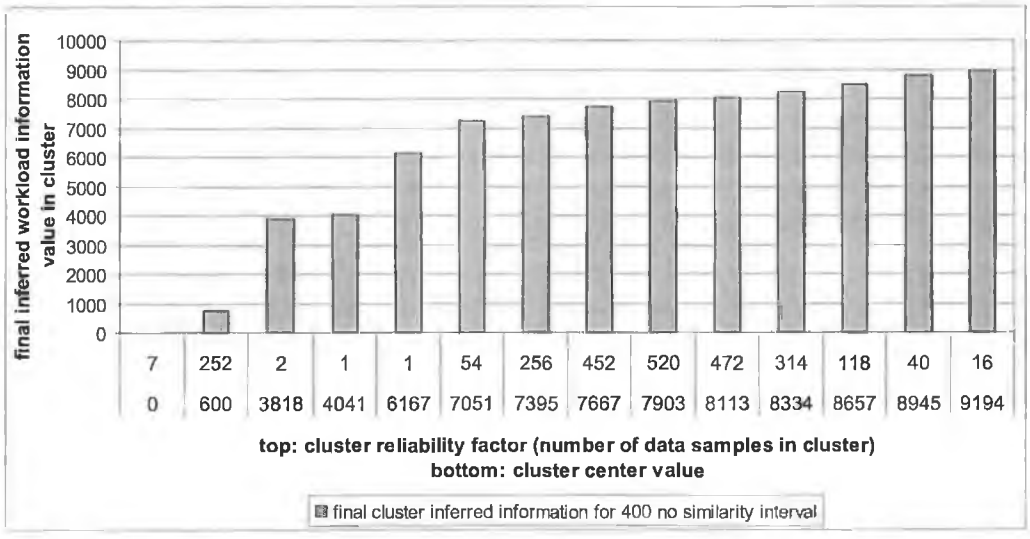

Figure 5.12: generated clusters - centre values, reliability factors and final inferred information, for 400 no similarity interval

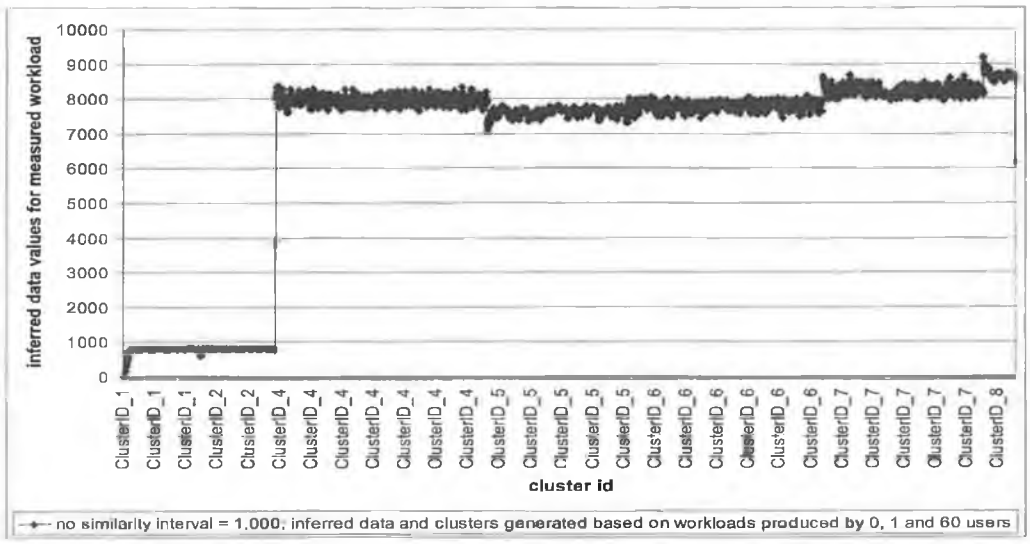

Figure 5.13: inferred workload data values and formed clusters for a no similarity interval of 1000

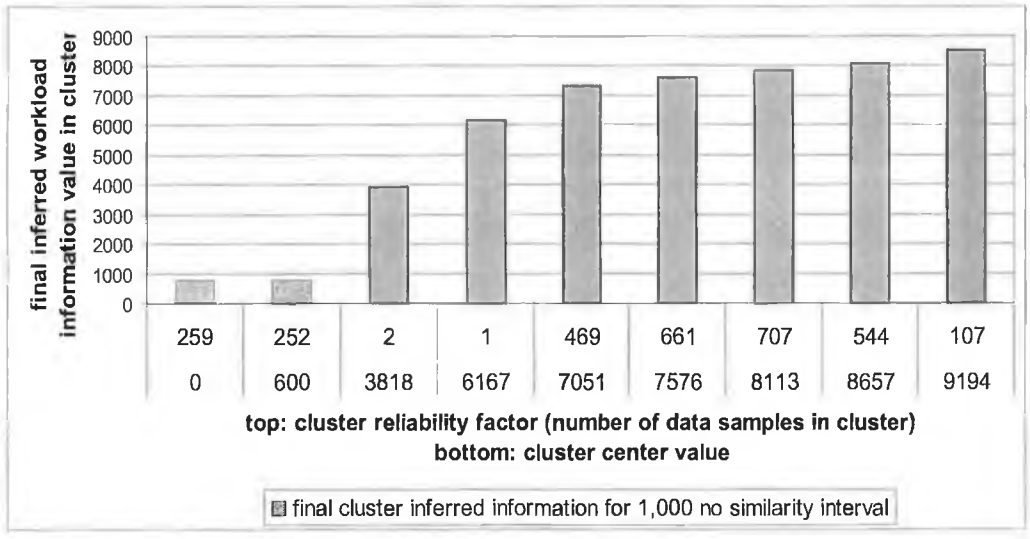

Figure 5.14: generated clusters - centre values, reliability factors and final inferred information, for 1000 no similarity interval

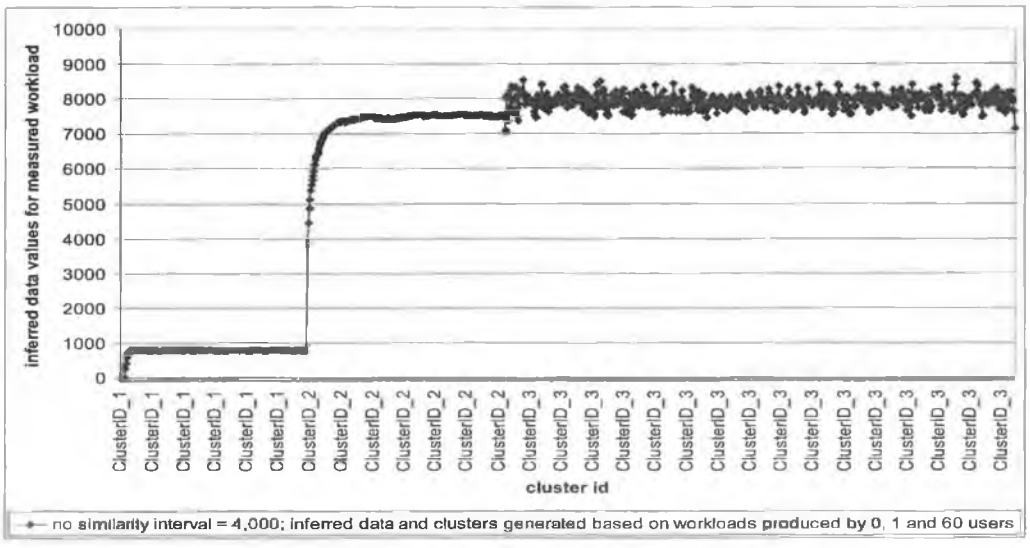

Figure 5.15: inferred workload data values and formed clusters for a no similarity interval of 4000

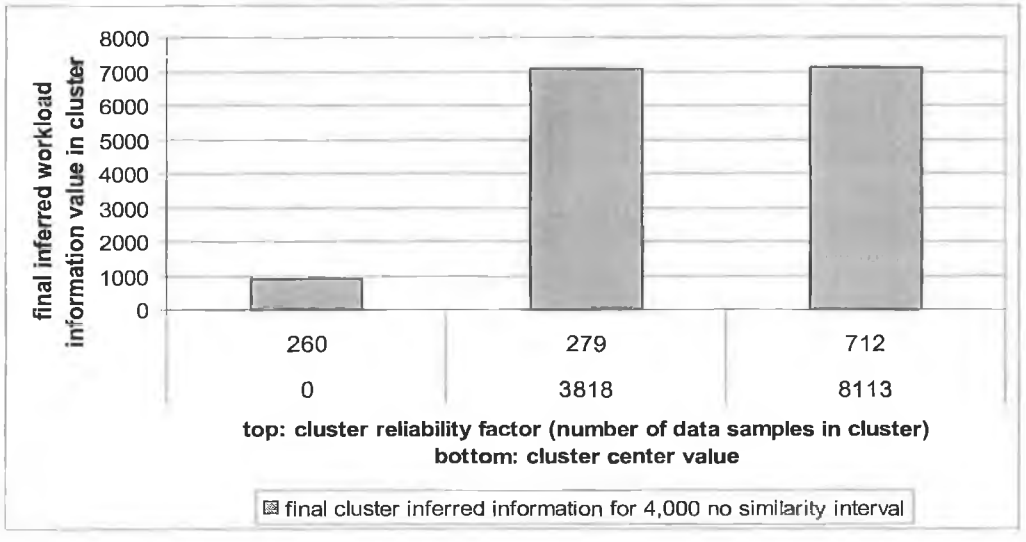

Figure 5.16: generated clusters - centre values, reliability factors and final inferred information, for 4000 no similarity interval

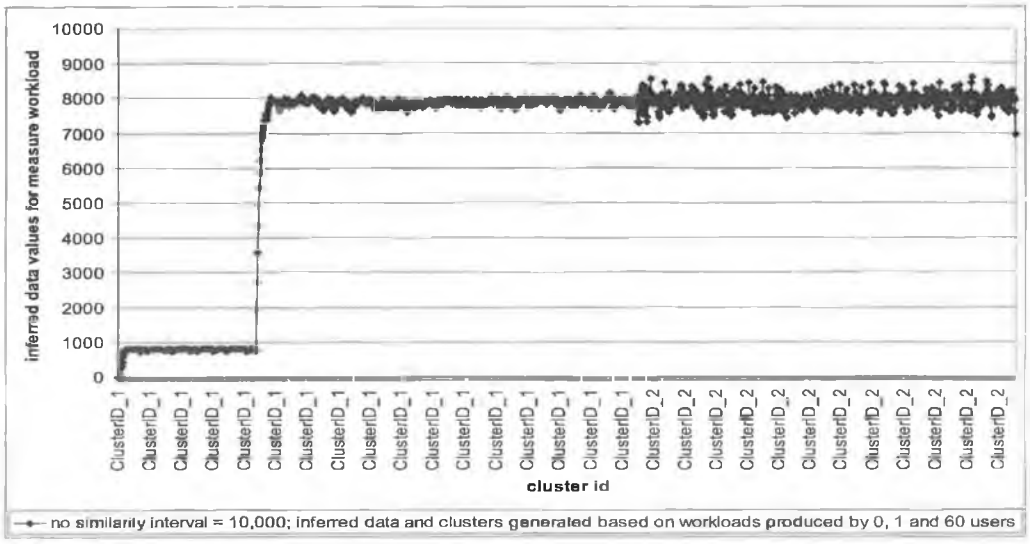

Figure 5.17: inferred workload data values and formed clusters for a no similarity interval of 10000

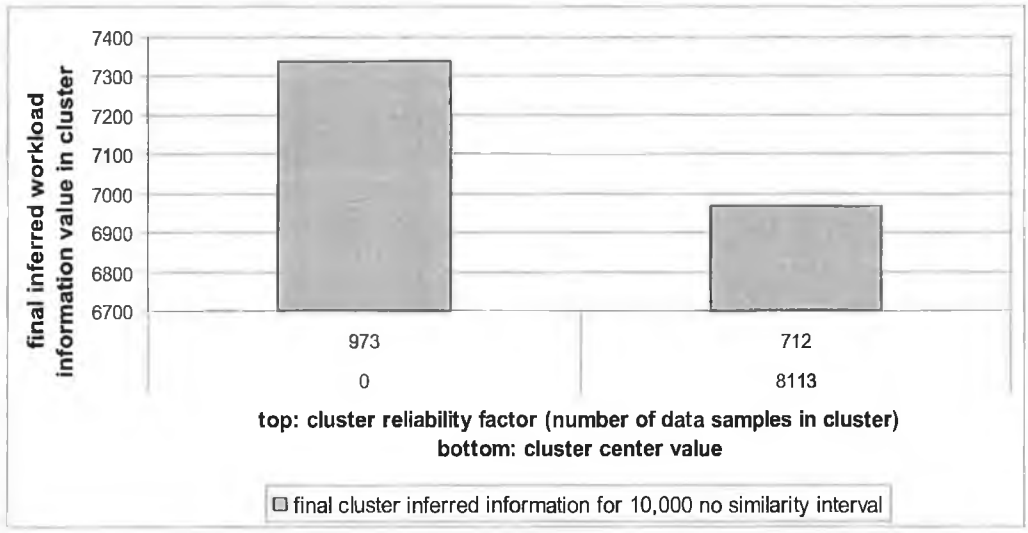

Figure 5.18: generated clusters - centre values, reliability factors and final inferred information, for 10000 no similarity interval

The obtained test results indicate the importance of the `no similarity interval` value configuration on the efficiency of the obtained performance information. More precisely, if the value of this parameter is too small, an unnecessary amount of fine-grained clusters is created. For example, when a `no similarity interval` of 100 was set in the presented test case, multiple clusters were created to represent workload data generated by the same number of concurrent users. Namely, 60 users generated an incoming workload on the monitored component method of between 7,000 and 9,000 user requests per time interval. More than 10 clusters were generated in this first configuration case to represent the environmental

conditions characterised by these workloads. This might be too fine a granularity, since the recorded workload differences were too small to make a difference in the selected redundant component. Nonetheless, even though inefficient, using this setting would not have impeded the evaluation algorithm from selecting the correct optimal redundant component. The reason is that all the fine-grained clusters covering the increased workload values would indicate the same optimal redundant component (i.e. the 500-bean-age variant). Thus, for any fine-grained cluster that the evaluation mechanism would find in the high workload interval, the same redundant component would be detected as optimal. Nonetheless, this case would be less efficient, as more clusters would need to be searched for in order to find a similarity match. In addition, each fine-grained cluster would contain less data samples. Consequently, the reliability of the inferred information element in each cluster would be decreased. Also, a possible danger when evaluating a current workload condition would be that the particular interval of the high workload range would not be covered by existing clusters. For this reason, accurate clusters would need to collect and process more data samples than clusters with larger spans, in order to achieve the same information reliability and environmental conditions coverage.

On the contrary, too large settings of the `no similarity interval` value may cause important information to be lost. For example, a step, or an important variation in the measured data values may be completely lost by the inferred information in case the selected cluster span covered too large a workload interval. For example, Figures 5.17 and 5.18 show the results of setting the `no similarity interval` to 10,000. This value was greater than the general workload difference recorded when the user load increased from 1 user to 60 users. More specifically, the recorded workload difference between the low system load and the increased system load was about 6,000, which was smaller than the selected cluster span of 10,000. This caused the generated clustered information to disregard the differences between the two system loads. Thus, in the particular test case in which the workload increased, the data collected during the low system load conditions was lost. This behaviour was correct in principle, except the coarse grain granularity used caused the inferred information to miss the important workload threshold between the two distinct environments. An optimisation opportunity given by this environmental conditions change could not be automatically detected in this case.

Test results also show that the inferred information element value can deviate from the cluster centre with up to the `no similarity interval` value. For example, when the 4,000 value was used for the `no similarity interval`, the cluster centred on a workload of 3818 requests per time interval had an associated information element with a value of 7091 requests per time interval (Figure 5.16). If an important variation occurred in the associated performance characteristics between these two workloads (i.e. 3,000 and 7,000 requests/interval) then the resulting clusters would not provide the correct inferred information. Thus, the cluster span should be selected so as to avoid such potential deviations from causing the loss of important information.

To conclude, the `no similarity interval` parameter has an important influence on the outcome of the cluster creation algorithm. Small values of this parameter generate many, more focused clusters, while increased values form less clusters with larger spans, or scopes. The trade-off is between accuracy and efficiency. Creating more focused clusters may increase accuracy, but may be less efficient in case the fine granularity obtained was not necessary. On the contrary, larger clusters may be more efficient and reliable, as more data samples are used

to infer the cluster information. However, relevant information may be lost if too extensive scopes are selected (e.g. for 4,000 no similarity interval, in Figure 5.16, and for 10,000 no similarity interval, in Figure 5.18, `no similarity interval` values in the presented test case).

The tested example scenario on Duke's Bank application (section 5.2) showed how component redundancy can be used to optimise system memory usage with variations in the system load. System load variations were obtained by maintaining the underlying platform resource availability constant and modifying the incoming workloads on the application. The current AQuA_J2EE prototype did not measure environmental parameters such as hardware and software resource availability (e.g. CPU load, memory occupancy, bandwidth consumption and processes, threads or connections available). The system memory consumption was measured separately at the JVM level and visualised using the GCViewer monitoring tool. For this reason, the memory consumption data was not collected by AQuA_J2EE and was thus not available for the learning algorithm. The way this data would be used to detect the optimal redundant component in the performed test case scenario is discussed next. The tested example scenario did not directly optimise application response times. Thus, analysing this parameter as part of the learning procedure would have brought no significant difference in the distinct execution contexts discovered and their associated optimal redundant components.

The `500-bean-age` redundant component was used during the performed tests, for the two different system loads (i.e. generated by 1 and 60 concurrent users respectively). Clusters of information were created for the `500-bean-age` variant, based on the collected workload data. The memory consumption measured in the corresponding workload conditions would be associated with the inferred information element of each generated cluster. During the learning process, further measurements would be collected while the `10-bean-age` redundant component was running under a low user load (e.g. 1 user). The clusters generated based on the collected data samples would be associated with the `10-bean-age` redundant component's performance description. This component's description would also indicate the occurrence of exceptions thrown in case this variant was run under increased system loads (e.g. generated by more than 20 users, on a certain platform). Based on the performance descriptions available for the two redundant components, the evaluation algorithm would automatically determine that the `10-bean-age` variant was optimal under low system loads, while the `500-bean-age` variant was optimal under increased system loads.

Two important configurations must be set on the information-inference learning procedure. First, system administrators have to decide which environmental parameters to consider for calculating the overall similarity factors between data samples (e.g. incoming workloads, available hardware and software resources). Second, administrators must specify the parameters to be considered when evaluating the components' performance characteristics and determining the optimal redundant variants (e.g. response times, throughputs, consumed hardware and software resources). Certain parameters in the two categories may overlap. For example, memory consumption may be used both to characterise an application's execution context as well as the application's performance. However, the way a parameter is interpreted in the two cases is conceptually different. In the previous example, when describing an execution context the memory parameter would indicate the memory availability, while from a performance perspective it would indicate the memory consumption of an application configuration. Using the same monitored parameters to group

data samples into clusters and to evaluate performance characteristics of components has an important advantage. Namely, it allows the detection of cases in which the parameters used to describe the components' execution contexts are insufficient. In this case, different performance characteristics (e.g. response times, throughputs, or memory availability) would occur for apparently similar execution contexts (e.g. incoming workloads, CPU and memory usage). Using the performance metrics to describe execution contexts would generate in this case different clusters of information, with different occurrence probabilities of the associated performance characteristics. These probabilities would be taken into consideration by a more sophisticated evaluation algorithm for determining optimal redundant components and their corresponding risks. This situation would also alert system administrators to try and determine the extra execution context parameters that may be relevant in predicting the measured differences.

CHAPTER

SIX

Conclusions

Goals of this chapter:

- The proposed component redundancy-based adaptation approach addresses real performance optimisation needs and system management requirements.
- The AQuA framework supports redundant components and automates the redundancy-based performance optimisation and system adaptation processes.
- AQuA's modular design enables the reuse of its individual management functions and encourages the adoption of separate custom solutions from representative research areas.
- The AQuA_J2EE prototype provides a valid instance of the AQuA management framework, for the J2EE component technology and the JBoss application server
- Test case scenarios and experimental results support the feasibility and applicability of the component redundancy-based solution and the AQuA framework for automatically optimising system performance.
- Related work on redundancy usage and automatic system management is complementary to the thesis. Existing approaches can be adopted to further the capabilities of the presented AQuA framework towards providing a complete autonomic management solution for enterprise systems.

6.1 Problems Addressed

Software systems are increasingly being implemented to control, manage and provide access to information and real world processes. The growing complexity of computer systems and their integration into everyday life places important demands on software management.
Enterprises have largely adopted component technologies for building large-scale, distributed applications. Component Based Systems (CBS) are generally developed by acquiring multiple components, possible off-the-shelf (COTS), and assembling them together into a coherent application, which supports the enterprise's business processes. Software application servers are used to provide common middleware services to enterprise application components, such

as security, transactions, connectivity and lifecycle management. This clearly separates the system's business logic from its middleware support, allowing programmers to concentrate on implementing the application's functionality while being able to reuse the middleware infrastructure provided by the application server. This approach significantly enhances system modularity, flexibility and reusability. System implementation and management costs and risks are consequently reduced. Nonetheless, the emerging behaviour of enterprise systems built using such component technologies is considerably complex. Complexity arises from the complexity of the implemented business processes, the complexity of the underlying middleware used and the dynamic nature of enterprise systems and their execution environments. In consequence, managing such complex enterprise systems and ensuring their quality requirements becomes a difficult process.

While successfully addressing system manageability and reusability requirements, current component technologies provide little support for performance management tasks. Namely, they supply little or no means of predicting and controlling the emerging performance of software systems assembled from distinct components. Static component testing and tuning procedures are undeniably important, but they provide insufficient performance guarantees for components that are to be run in diverse component assemblies, under unpredictable workloads and on different platforms. The environmental conditions in which a component may run as part of a software application can periodically change during the component's lifetime. Often, no single component implementation or deployment configuration exist that can yield optimal performance in all possible conditions under which a component may run. Consequently, system optimisation and tuning processes must be repeatedly performed during system runtime, rather than only once before system deployment. Nonetheless, manually optimising complex applications and adapting them to changes in their running environments is a costly and error-prone task. In consequence, the essential necessity for automating system management procedures has been identified.

6.2 Solution Overview

The thesis proposed a solution for automating the performance optimisation of component-based enterprise systems. The solution is based on using functionally-equivalent redundant components, optimised for distinct execution environments (e.g. incoming workloads). Knowledgeably alternating the use of redundant components allows applications to adapt to variations in their execution environments and yield optimal performance at all times. As such, systems are dynamically configured so as to use the optimal redundant components in each execution context.

To support this process, the thesis additionally proposes an automatic management framework called AQuA (Automatic Quality Assurance). The framework's functional goal is to be able to perform the necessary monitoring, decision and adaptation operations without the need for human intervention. AQuA manages redundant components and enables software systems to fluently mould to their changing execution environments. The thesis focus is on optimising system performance. Nonetheless, other system qualities, such as reliability and availability, can also be managed based on the proposed component redundancy principles and automatic management approach.

At runtime, AQuA monitors application components and their execution environment. Collected data is analysed so as to detect performance anomalies or significant variations in the system's performance characteristics or running conditions. AQuA subsequently evaluates available redundant components and determines the optimal system configurations. System adaptation decisions are subsequently taken, considering the predicted benefits of the selected optimisation solutions as well as other related factors such as potential costs and risks. Finally, system adaptation decisions are automatically enforced into the running system. Part of the framework's control feed-back loop, system adaptation results are subsequently monitored and the actual performance gains are compared with the intended benefits. Collected monitoring data is analysed and used to automatically learn and improve AQuA's management behaviour over time.

While the proposed redundancy-based optimisation solution is conceptually simple, it is by no means straightforward to implement. Global performance optimisation issues need to be considered in addition to local optimisations. The impact that performance optimisations have on other system qualities, such as correctness, or dependability has to be considered. Thus, the solution's complexity is precisely the reason for which the management processes required to support it should be automated, as specified in the thesis. Finally, the proposed approach must be integrated with other system management mechanisms in order to provide a complete autonomic management solution.

6.3 Review of Contributions

6.3.1 Using Component Redundancy for Optimising Performance

The thesis proposes a performance optimisation solution based on redundant components (chapter 3). Conforming to this approach, a number of redundant components are prepared to provide certain application functionality. While functionally equivalent, each redundant component is optimised for a different range of running conditions, such as different incoming workloads or available system resources. The redundant components are knowledgeably alternated during runtime, so as the system uses the optimal redundant components at all times. This enables applications to initially optimise their performance for the execution context in which they are deployed and run. Most importantly, it enables applications to dynamically adapt to subsequent variations in their running environments. As a component's execution environment changes, the component is dynamically replaced with an equivalent redundant variant optimised for the new running conditions. The proposed solution completely separates the available redundant behaviours from each other and from the adaptation logic used to select their alternate usage. The solution's modularity allows redundant components and adaptation logic policies to be independently added, deleted, or modified, thus improving system flexibility and manageability.

The solution specification takes into account the possible side effects a local redundant component-swapping operation can have on the overall system performance. As such, it is proposed that local component optimisation processes are combined with a global system

control facility, which would facilitate a general system view and an overall optimal system. The manner in which a system's performance optimisation may affect other system qualities, such as security or reliability, must also be considered.

6.3.2 Automated Performance Optimisation Framework

An automatic management framework was proposed to perform runtime monitoring, decision and adaptation operations on administered systems. The implemented tasks were necessary for supporting the proposed redundancy-based optimisation solution. Using component redundancy, the AQuA (Automatic Quality Assurance) framework enables applications to fluidly shape their behaviours so as to optimally fit in their varying execution environment.

Modular, Extensible Design

The main functionalities the AQuA framework features include runtime monitoring, anomaly detection, component evaluation, adaptation decision, component activation and learning. Each of these functionalities can be independently extended as part of separate research efforts, or by adopting more complex solutions from the existing specialised research areas (e.g. system monitoring, policy-based adaptation logic, data mining, knowledge management, machine learning, component hot-swapping, or component versioning). The AQuA framework specification shows the required management functionalities and the way they should interoperate in order to provide a complete performance optimisation solution. It provides a way of integrating separate research efforts on monitoring, policy-based management and dynamic adaptation subjects into a single, complete, autonomic administration solution. The modular framework design allows for any of its functionalities to be independently updated without affecting the other functionality implementations.

Learning Capabilities

Learning capabilities were specified for the AQuA framework in order to enable it to improve its management behaviour over time, with minimum human intervention. A learning mechanism was designed to analyse collected monitoring data and infer high-level information on the components' performance characteristics. The goal of this automated learning mechanism is to simulate the processes that human administrators or testers would perform, in order to acquire information on the components' performance behaviours in various environmental conditions. As such, monitoring data collected in certain execution environments is merged into clusters of information and used to predict the components' performance in similar execution contexts. The more data samples are available in a cluster, the higher the confidence level associated with the performance prediction associated with that cluster.

The proposed learning capability avoids imposing extra requirements on component providers or deployers. More specifically, precise information on the components' performance characteristics is not compulsory to be provided at component deployment time. This information can be automatically inferred over time from monitoring data collected while components are integrated in the targeted managed system. The learning process is executed

during the system's runtime. Additionally, it can be initially run while the system is running off-line, under simulated or predicted workloads and data.
The learning mechanism is also used to avoid completely relying on initial performance information obtained from static component testing procedures. Such procedures are typically performed while components are executing on various platforms, different from the targeted running environment. Thus, they often provide insufficiently accurate or reliable information on the components' performance characteristics, when run in dissimilar environments. Initial information can indeed be provided at components' deployment time, from performed tests and previous experience with the targeted components. In this case, the learning process is subsequently used to validate and accordingly update initial performance information, with accurate monitoring data obtained while the components are running in the targeted system and execution environment.

Specialised Design for Managing
Component-Based Enterprise Applications
The AQuA framework was specifically designed to manage enterprise applications built using component technologies based on contextual composition frameworks [91] (subsection 2.3.4), such as EJB, or CCM. For this reason, AQuA's design was devised to meet the particular characteristics and requirements of such enterprise systems. A decentralised management control topology was specified for scalability reasons to perform local management operations at the component level. In addition, two global management solutions were proposed to provide general control over the decentralised framework instances and ensure overall system optimisation. As such, global management can be achieved by using a centralised framework entity to manage component-level framework instances, in a hierarchical topology. A second global management solution can be achieved by specifying and implementing a communication protocol between the decentralised framework instances. This approach is possible when managing application performance because of the manner in which the local performance of individual components influences the overall performance of a client transaction. As such, local component managers involved in a client transaction can intercommunicate to signal the influence local optimisations on other components had on their managed components. The communication protocol should be specified and tuned so as the emerging behaviour of local component managers and the communication protocol led to an overall optimal configuration at system level. A decentralised management control topology was specified, implemented and tested as part of the AQuA_J2EE prototype (sections 3.11 and 4.2). The hierarchical and communication protocol-based approaches for global system optimisations were specified and discussed in section 3.11.

6.3.3 Relevant Examples of Component Redundancy Applicability for Performance Optimisation

Several example scenarios were shown to indicate the applicability of the redundancy-based solution for performance optimisation. The examples show how the alternate usage of redundant components can benefit system performance, or other quality attributes, such as reliability and manageability. Two such examples were implemented and tested, as described in

sections 5.1 and 5.2. The AQuA_J2EE framework prototype was tested for one of the implemented example scenarios, which involved the Duke's Bank sample J2EE application. Test results from the experimented example scenarios clearly indicated the potential benefits of the component redundancy-based solution.

6.3.4 Framework Prototype for J2EE

A fully-automated prototype of the AQuA framework - AQuA_J2EE - was implemented for the J2EE component technology. The JBoss application server was selected as the J2EE application server for AQuA_J2EE. Nonetheless, due to the way it was designed, AQuA_J2EE can be modified to work on any J2EE-compliant application server (section 4.3). The server-independent parts of the AQuA prototype can be seamlessly reused to create a management framework for various application servers, or different component technologies. This can be achieved by integrating AQuA_J2EE with different monitoring and system adaptation solutions, as appropriate for the targeted management platforms. For example, a server-independent version of AQuA can be built by integrating AQuA_J2EE with the COMPAS monitoring tool[1] and a platform-independent proxy-based component-swapping solution.
The current AQuA_J2EE implementation uses custom monitoring and component-swapping mechanisms for the JBoss application server. JBoss EJB containers were instrumented to extract runtime monitoring data and send it to AQuA's adaptation and learning logic, for further processing. A modified version of JBoss's hot-deployment functionality was used to implement AQuA_J2EE's component-swapping capability. AQuA_J2EE's adaptation logic was specified based on a decision policy-based approach. The ABLE Rule Language (ARL)[2] was used to declare decision policies, in scripting files completely separated from the underlying framework mechanisms (section 4.6). This allows human system administrators to seamlessly state their high-level management goals and strategies, without the need to understand or modify the underlying framework mechanisms. In addition, administrators can use their management expertise to express management policies in a formal manner, which can subsequently be interpreted and executed by an automatic management framework.
A learning mechanism was implemented for AQuA_J2EE, in order to automatically analyse collected monitoring data and infer higher-level information on the components' performance characteristics. The algorithm used to implement the learning function is based on grouping and merging data samples based on the similarity of the execution environments in which the samples were collected. Merged data samples form clusters of information which are used to predict components' performance in reoccurring execution contexts. The more data samples are merged into a cluster, the higher the reliance that can be placed on performance predictions that are based on the information in that cluster. The information inference learning procedure was described in section 3.13. The learning process can be triggered whether repeatedly during runtime, as new monitoring data samples become available, or upon request, to analyse an entire set of collected monitoring data samples.
The automatic management capabilities of AQuA_J2EE have been successfully tested on a

[1]COMPAS J2EE monitoring and analysis tool: compas.sourceforge.net
[2]ABLE Rule Language (ARL), from IBM: www.research.ibm.com/able

sample J2EE application, the Duke's Bank[3]. AQuA_J2EE automatically detected changes in the execution environment (i.e. incoming workloads), decided to adapt the managed application (i.e. the Duke's Bank) and performed the corresponding EJB-swapping operations, while the application was continuously running [28] and [33]. Obtained test results showed that system performance and availability were visibly improved when AQuA_J2EE was used to adapt the application, compared to the case when no adaptation was used.
The modular design of AQuA_J2EE allows any of its management functionalities to be independently modified, without affecting the rest of the framework implementation. As such, any of the framework's management functions can be separately extended or replaced with custom solutions from the respective research areas. The same framework infrastructure can be used to manage other system quality attributes, such as reliability or availability. AQuA_J2EE can be customised in this case so as to serve the new management goals, as follows. The monitoring functionality can be extended in order to collect additional system parameters, as relevant for the new managed quality attributes. Decision policies can be specified to state the new system management goals, which include the additional quality attributes. Finally, additional system adaptation mechanisms can be implemented to allow the dynamic replacement of components with various granularities or of different types, such as middleware services and application servers.

6.4 Comparison with Related Work

This thesis is related to work from two main research areas. Namely, the presented work is mostly similar to research on using redundancy for enhancing system quality characteristics and to research on autonomic management frameworks. Additionally, relevant related work also exists in certain sub areas of the aforementioned research domains. Such sub areas include work on system monitoring, policy-based management, component hot-swapping, data mining, machine learning, statistics, knowledge management, system evolution and emergence. Relevant results from these research areas can be adopted and integrated with the proposed component-redundancy based optimisation solution and with the AQuA framework. AQuA provides an integration point for the outcome of multiple autonomic computing related areas. Related work from these two research domains was discussed in chapter 2. The rest of this section discusses the most significant aspects differentiating the thesis from related work in the two aforementioned domains.

6.4.1 Using Redundancy for Improving Performance and Dependability

The necessity for multiple implementation variants for achieving optimal performance in dynamically changing execution environments has been indicated in other software research domains, including the management of scientific applications [93], operating systems [3],

[3]The Duke's Bank sample J2EE application from Sun Microsystems: java.sun.com/j2ee/tutorial/1_3-fcs/doc/Ebank.html

Web services and Service-Oriented applications [64], data management applications [101], distributed real-time embedded (DRE) systems [8], reflective middleware and component technologies (e.g., K-Components [37] and Arctic Beans[4]), or network-centric combat systems [83]. The general concepts in these approaches are similar to the proposed redundancy-based optimisation solution. Nonetheless, these various research efforts aim at managing significantly dissimilar system types. This causes considerable differences in the way the respective solutions are designed and implemented. The main differences between the thesis solution and some of the most similar research projects are discussed next.

Performance optimisation techniques such as presented in [101] are conceptually similar to the presented component redundancy-based optimisation approach. The main features differentiating AQuA's adaptation solution from these approaches are the lack of requirements on component providers to supply accurate initial performance information on individual redundant components, or replacement mechanisms for each separate pair of redundant variants. In addition, the actual exploration work carried out in the two approaches is significantly different. The research presented in [101] focuses on an adaptation decision algorithm, called Delta. The goal of the Delta algorithm is to determine the most favourable cross-points for hot-swapping redundant components, so as to attain optimal performance benefits. Targeted scenarios involve component implementations that take a significant amount of time to hot-swap, in comparison with client request response times. An example of such a scenario involves the hot-swapping of two different persistence-support structures, a relational database and an LDAP style Directory. AQuA is different in that it targets the management of enterprise application business logic, and does not attempt to transfer state between hot-swapped component implementations. Thus, in general, the delays induced by the actual hot-swapping operations are not significant when compared to client request response times. If they were for some cases, the Delta algorithm can be adopted by AQuA's decision process.

The two research approaches also differ in their requirements for the source of redundant components and their hot-swapping mechanism. In the framework proposed in [101], the component developer is responsible for providing all component implementations, as well as the code that controls the hot-swapping between these implementations. This is not a requirement in the AQuA framework. In AQuA, different redundant components can be acquired from different providers. The hot-swapping between redundant components is performed by the AQuA framework, uniformly for all redundant component pairs. In other words, in AQuA, the component variant hot-swapping operation does not depend on the particular component variants involved. AQuA provides an entire management framework specification, with provided monitoring, adaptation logic and hot-swapping operations. The targeted applications that AQuA was devised for are enterprise applications built using contextual composition frameworks [91]. The adaptation logic in AQuA is based on performance information acquired at runtime, based on monitored data and a provided learning process. In [101], the decision process is based on static component performance information provided at component deployment time. By design, AQuA considers multiple execution context parameters, not only workload. This is because in certain cases, application or component-level workload measurements are not sufficient to indicate significant changes in a component's execution environment. Considering and analysing multiple environmental parameters, such as software and hardware resource usage, provides a more complete view of a component's run-

[4]Arctic beans project: http://abean.cs.uit.no

ning context, which has a significant influence on the component's performance. In short, the focus of the two research efforts renders them complementary rather than conflicting. Namely, the decision algorithm presented in [101] can be adopted and used as part of AQuA's decision policies, for managing systems in which the adaptation operations take considerable periods to execute.

The Active Harmony project[5] proposes an automated performance management framework for scientific applications and grid computing. The taken approach is based on automatic library swapping and parameter configuration during runtime. More precisely, the Active Harmony system allows the dynamic switching of algorithm implementations and the dynamic configuration of library parameters, in order to automatically tune the applications' performance during runtime. The associated research focuses on the specification of an algorithm that is able to find the optimal parameters configuration for various execution contexts. Performance parameters considered include CPU time and memory. Unlike the AQuA framework, in the Active Harmony approach applications need to meet several requirements in order to work with the Active Harmony server. These requirements include providing information on the application's tuneable parameters and needed resources. These are specified in a special-purpose Resource Specification Language (RSL) and accessible via a specific API. These requirements may be achievable in the targeted scientific application domain, where the swappable components consist of software libraries implementing scientific algorithms, or data storage structures [93]. As an additional requirement, in the Active Harmony approach the internal implementation of managed applications needs to be modified. Specifically, applications must register with the Active Harmony server upon start-up and to periodically require parameter tuning updates from the Active Harmony server during runtime. The AQuA framework imposes no such requirements, as the application instrumentation and adaptation mechanisms are implemented outside the actual application components. More specifically, AQuA's monitoring sensors and adaptation actuators may be implemented whether in the supporting system middleware, or in system-independent component proxies.

Redundancy as a means of achieving dependability and performance for Service-based Internet systems, such as Web services systems, is proposed in the RAIC (Redundant Arrays of Interchangeable Components) project [64]. The addressed problem domain in this case however, is different in scope from the presented thesis. This is because RAIC was devised to manage systems that are composed of various Internet services, offered by different providers, from different geographic locations. No single authority owns, or has complete control over the entire system. The developer of one system service has no knowledge of, or access to the implementation, deployment platform, or supporting resources of the other services it needs to rely upon. Redundancy support cannot be implemented in this case at the execution platform or middleware level. Instead, redundancy support for the Internet services used is implemented at the (client) software application level. The client application dynamically selects the service provider that is most reliable and that supplies the optimal performance in the current execution context. AQuA can be used in combination with the RAIC approach, to allow a certain service provider to supply optimal performance to its clients, at all times; this may increase the provider's chances of being selected by interested clients.

The use of multiple implementation strategies for performance optimisation purposes, as in the Open Implementation approach [57], is similar to the intent of this thesis. The essential

[5]The Active Harmony project: www.dyninst.org/harmony

difference between the two approaches is in the manner in which the optimal module implementation is selected from the available implementation strategies. The Open Implementation initiative allows clients to decide which implementation variant to instantiate and use for optimal performance, in a specific context. The proposed AQuA framework automatically takes such decisions. The rationale behind the proposed approach is that often there is no single strategy that is optimal under all possible execution contexts in which a component will run. Therefore, clients may not be able to statically decide the optimal strategy to select, at application design and implementation time. The strategy used needs to dynamically change in order to accommodate runtime changes in the component's execution context. Nonetheless, manually determining and employing the optimal implementation strategies, in due time, would be prohibitively expensive for human system managers. This would be especially the case for complex systems and frequently changing environments. Therefore, AQuA automates the management processes needed for selecting and using the optimal implementation strategies at all times.

Another important difference between the Open Implementation approach and the AQuA management framework is in the manner in which the optimal implementation strategy is selected. Conforming to the Open Implementation, the client decides on the optimal implementation strategy to use. The decision is based on information on the way the client will use the module implementation. In contrast, in the AQuA approach, the decision is taken at the server, or provider side, rather than at the client side. That is, each component (or module) decides which implementation strategy to use for handling incoming client request. The decision is based on information on the current component workload, usage patterns and available system resources. For this reason, the Open Implementation and AQuA framework can be used together as complementary approaches.

In addition to performance management, component redundancy has previously been used to provide fault-tolerance and improve system reliability. These approaches are fundamentally different from this thesis' work, in both their targeted objectives and their employed strategies. More precisely, improved reliability was typically achieved by running multiple redundant components (sequentially or in parallel), comparing the obtained results and deciding on the correct response to return. The presented performance optimisation solution is based on selecting a single redundant component to execute at any one time, so as to attain optimal performance under varying running conditions. The Recovery Blocks (RB) and the N-Version Programming (NVP) techniques use redundant software variants to provide fault-tolerance for software systems. Testing and decision mechanisms are employed in both approaches, for managing the multiple variants and obtain the correct results. These aspects constitute the main similarities with the presented dissertation research. Nonetheless, important differences stem from the fact that fault-tolerance related approaches target functionality-specific faults, particular to each application, whereas the thesis aims at solving performance-related problems, functionality-independent and thus common to all applications. More precisely, in order to detect functional errors, fault-tolerance schemes require knowledge on the correct system behaviour, as well as methods for assessing system behaviour correctness during runtime. These requirements need to be separately provided for each particular system, as they directly depend on the specific system functionalities. On the contrary, the thesis focuses on performance-related problems, common to all applications and independent of functionality semantics specific to each system. The AQuA framework can also be used to handle certain types of faults that can be detected independently of system functional semantics. Such faults

include thrown exceptions and non-functional integration faults, such as deadlocks. Performance problems, exceptions and integration faults can generally be detected without requiring application semantics information. Application semantics specify the correct behaviour of an application. Being independent of the application's semantics, the presented framework does not need to be re-implemented for each particular application it manages. The framework can be implemented once, as part of the component platform, or middleware layer, for the benefit of all applications deployed on such platforms. The framework implementation will evidently need to be configured according to the specific performance requirements of each particular application managed.

With respect to performance overheads induced by fault-tolerance schemes, the RB technique introduces execution-time overheads, because of the acceptance test and sequential execution of variants. NVP introduces resource usage overheads, as all variants execute in parallel, even when no faults are detected. In the AQuA framework a single redundant component is run at any one time for handling a certain client request. This means that in AQuA, no processing overheads are introduced since multiple redundant components are absent. Also, no acceptance tests are performed in AQuA for component output assessments. However, certain framework functionalities for supporting component redundancy and automatic system management in AQuA will certainly have some impact on resource usage. Specifically, monitoring functions will constantly impact system performance as they are performed continuously during runtime, in order to collect information and detect performance anomalies. Component activation operations, involving the hot-swapping of redundant components, will also impact system performance, but only during application adaptation periods. The adaptation logic-related processes, including the analysis of monitoring data, anomaly detection and learning, component evaluation and adaptation decision operations continually need system resources. Nonetheless, these operations can be run remotely from a separate station and hence would not impact the managed system performance. In addition, AQuA's specification allows for its functionalities to be configured for performing optimally in different running contexts.

In the presented fault-tolerance schemes, neither the variants nor the decision algorithm (or adjudicator) can be changed during system execution. In the RB scheme, it is because the acceptance test is mingled with the functional variants. In contrast, AQuA's design allows for both redundant components and adaptation logic policies to be dynamically added or deleted at runtime. This is a consequence of the fact that component variants are separated from each other, as well as from the evaluation and decision logic policies. This allows redundant component variants and decision policies to be modified separately and independently of each other. In the RB approach, the adaptation logic for deciding which variant to use is implicit and cannot be modified. It is dictated by the order in which variants are listed in a recovery block. In the case where the primary variant is faulty and fails the acceptance test, it will still be used as the primary option for handling subsequent client requests. The AQuA framework is different in that it is able to *learn* from its previous experience with a managed system and accordingly modify its adaptation logic. Thus, AQuA always aims at using the optimal redundant component(s) in each situation, and avoids reusing undesired component configurations.

Research in the area of dynamic component versioning bears certain similarities to this thesis. However, the main intent and goals of dynamic component versioning and redundancy-based performance optimisation are notably different and focus on different system management aspects. In some component versioning approaches [77], a number of component versions can coexist, in order to continue providing deprecated functionalities and accommodate already

existing clients. This is different from the presented redundancy-based approach. The difference is that in component versioning, different versions can provide different functionalities, or services. For example, a new component version can support new functionalities compared to the old one, whereas the old version can support functionalities that have been deprecated in the new version. In these approaches, new versions, with new functionalities and improved performance, are meant to replace the old versions, in time. Old versions are only temporarily maintained in the system, for version compliance-related reasons. They are not intended as alternatives to new versions, or as part of the system's adaptation facilities. In the AQuA approach, all component variants provide the same, or equivalent, functionalities. All redundant variants advertise the same provided functions or services. AQuA can also be used for cases in which component variants would trade the quality of their service responses, including (for example) result accuracy, the security method employed, better service performance, or reduced resources demand. In other words, service degradation can be employed in order to improve the system's non-functional quality characteristics. Nonetheless, the functionalities advertised by the redundant component variants to external clients, through their public interfaces, are always identical.

6.4.2 Autonomic Performance and Dependability Management

In the area of autonomic system management, there are no performance optimisation frameworks that completely overlap with the present work on AQuA_J2EE. More precisely, no similar frameworks exist that employ monitoring, learning, decision and adaptation facilities for applications based on contextual composition frameworks [91], at the application component level. General frameworks for self-adaptive systems are presented in [72] and [43], featuring inter-related monitoring, analysis and adaptation tiers. AQuA_J2EE aligns with these solutions, while specifically targeting the performance of enterprise applications based on contextual composition middleware [91]. Management solutions have been devised for other component technology types (e.g. [43] or [93]) or Web services based systems [64]. These frameworks differ from the thesis by their management requirements and subsequent applicable solutions, as discussed in subsection 2.4.3. For example, the Rainbow project [43] proposes an automatic framework for managing the quality attributes of distributed component-based systems. The proposed AQuA framework complies with the general Rainbow architecture and goals, while focusing on the performance management of Internet-enabled enterprise systems. In addition, while Rainbow is based on a centralised model-based approach, AQuA was designed towards decentralised and hierarchical control solutions.

Research efforts for automating system management for other quality attributes complement the presented performance optimisation solution. Several projects, such as JAGR [20] and JADE [73], propose automatic frameworks for managing the availability and dependability characteristics of component-based applications. Similarly to AQuA, these frameworks were also devised to manage Internet-enabled enterprise systems, focusing on the J2EE component technology. JAGR [20] uses component level micro-reboots as the repair mechanism for transient faults. A hot-deployment based solution was adopted for micro-rebooting faulty

EJB components on JBoss. AQuA_J2EE can be used for the same purpose, by specifying management policies that detect and dynamically replace a faulty redundant component with the same redundant component. This procedure would be equivalent with re-deploying or re-booting a component. JADE [73] focuses on automating the deployment and re-configuration of J2EE systems. Managed entities can be entire servers (e.g. Tomcat), server-provided services (e.g. security, or transactions), or individual application components (e.g. EJBs). The proposed redundancy-based solution and AQuA framework is complementary with this work. For example, the platform-independent part of AQuA_J2EE can be integrated with the proprietary monitoring and re-deployment mechanisms implemented in JADE. This would extend the quality attributes that can be automatically managed in J2EE systems and leverage the presented policy-based problem-detection and adaptation decision mechanisms.

Monitoring and data analysis tools such as COMPAS[6] are compatible with the monitoring and diagnosis module of AQuA_J2EE. For example, the modified version of JBoss used for AQuA_J2EE has initially been configured to send monitoring data to the COMPAS monitoring tool, which further analysed collected data, displayed it into performance graphs and signalled performance alerts [28].

Several researchers in the area of model-based adaptable software use system architecture as a basis for constructing, evaluating and re-factoring system models [23, 72, 37]. Architectural system models are represented in a graph-like manner. System components represent the nodes of the graph. Component interconnections are represented as directed arcs in the graph, correspondingly connecting the graph nodes. System adaptation consists of changes in the system components and component interconnections, by means of graph reconfiguration operations. System evaluation and optimisation operations are performed in a centralised manner in these schemes. Monitoring information is centralised, the overall system is evaluated and globally optimised. For large-scale systems, possibly consisting of tens or hundreds of components, globally evaluating and optimising the system whenever a local problem is being detected might introduce significant overheads and not scale well. This thesis proposes propose a framework where the system adaptation operations are decentralised. In the presented approach, when a problem is detected locally, by component-level adaptation mechanisms, attempts are initially made to locally solve the problem, without affecting the rest of the system and without involving higher-level adaptation mechanisms. In addition, two possible approaches are proposed for handling problems at a higher or global level. One approach involves a hierarchical organisation of the adaptation mechanisms. In this approach, local, component-level adaptation mechanisms are controlled by higher-level mechanisms. A single highest-level adaptation mechanism is always available to supervise the entire adaptation framework from a global level. When this approach is used, unsolved local problems are signalled upwards in the hierarchical adaptation tree and addressed at higher adaptation levels. Global system optimisations can also be triggered periodically or upon request.

This thesis' aim was to provide automatic performance optimisation capabilities for managing Internet-based enterprise systems. This goal complies with the general objectives of the self-adaptive and autonomic computing initiatives. The provided functions represent a subset of the functionalities specified as part of these initiatives. The AQuA framework was devised to support the goals of the presented performance optimisation solution. Its

[6]The COMPAS monitoring and analysis tool: compas.sourceforge.net

design conforms to the general frameworks proposed in several research efforts from the self-adaptive and autonomic management area (e.g., [72], [43], and [54]). The main similarities to these frameworks are related to the functional modules involved and their connection in a closed control loop. Monitoring, decision logic and system adaptation functions are specified in most self-management frameworks. Therefore, most self-adaptive software solutions present certain similarities to this thesis. Nonetheless, the differences in the targeted domains and system types raise different problems, imposing different requirements on the performed management processes. Therefore, different approaches have to be adopted when specifying, designing and implementing adaptation mechanisms, depending on the general characteristics of the targeted system type.

The term 'component' is used with different meanings in different research initiatives, to refer to various system parts, such as servers, clients, software modules or entire software applications. This is an essential aspect when discussing the main differences between the thesis and related work in the self-adaptive software area. The unique nature of Internet-based enterprise applications built using component technologies such as EJB might make general approaches devised for dissimilar component-based system types difficult to apply (subsection 2.4.3). Some of the main system type characteristics that AQuA was customised to consider include soft inter-component bindings and unpredictable, frequent fluctuations in the number of component instances. In systems in which the managed 'components' represent servers, software modules, or embedded devices such issues may not be as stringent.

AQuA uses component hot-swapping operations in order to switch between the available redundant components. The same mechanism can also be used to dynamically replace or update AQuA's adaptation logic, though this functionality is not supported in the AQuA_J2EE prototype. In AQuA_J2EE, the hot-deployment facility provided by the application server used (i.e. JBoss) was modified to support component-swapping operations. Efforts towards standardizing and implementing component hot-deployment functionalities in J2EE applications are being made in research initiatives such as Sun Microsystem's JSR 88[7], or [65]. Available standards and existing solutions for component hot-swapping can be adopted and integrated into AQuA_J2EE.

With respect to monitoring data analysis and learning procedures, several research projects adopted similar approaches for processing extensive data in complex systems [34, 25, 24, 102]. The direction taken is to correlate low-level system configurations and monitored data with higher-level observed events of interest, such as the system's performance characteristics (e.g. response times, or throughputs).

In [34], such an approach is taken to automatically optimise the CPU utilisation of an Apache Web server, by tuning the MaxClients and KeepAlive configurable parameters exposed by the server. The AutoTune agent framework proposed by this research associates the server's various tuning configurations with the resulting performance characteristics, in order to automatically determine the optimal server configurations.

The approach proposed in [25] uses system history information to predict and argue about future performance anomalies. The proposed method associates system state signatures with observed performance problems in order to assist operators in diagnosing and comparing detected performance anomalies. Raw monitoring data is merged into clusters of informa-

[7]JSR 88 on deployment from Sun Microsystems: java.sun.com/j2ee/tools/deployment

tion, where each cluster associates a system's state signature with the related performance problems observed while the system was observed in similar states.
While the research in [25] focuses on predicting and arguing about system performance anomalies, the presented thesis work applies a similar approach for arguing about the performance characteristics of various design and configuration variants, with the goal of identifying the optimal system implementation in each execution context.

6.5 Validation of Redundancy-Based Optimisation Solution and Framework

Several examples indicating the applicability of redundant components for performance optimisation were described in section 3.5. Some of these examples were implemented and tested, as described in sections 5.1 and 5.2. Obtained results validated the assumption on the potential benefits of component redundancy on system performance. More specifically, one implemented example showed how different redundant components provided optimal response times under different network loads (section 5.1). A second example, based on the Duke's Bank sample J2EE application, showed how different redundant components can provide optimal memory usage under different incoming workloads (section 5.2).
The AQuA_J2EE prototype was implemented and tested to validate the AQuA framework specification, for automatic performance optimisation based on redundant components. AQuA J2EE was tested on the Duke's Bank sample J2EE application, for which several redundant components were prepared. Obtained test results showed that system performance and availability were visibly improved when AQuA_J2EE was used to adapt the application, compared to the case when no adaptation was used (subsection 5.2).
In the executed testing scenarios, performance overheads induced by the management operations during normal system execution were insignificant. This is due to the fact that AQuA_J2EE was configured to only manage a certain set of components, rather than the entire application. In addition, monitoring delays were minimised as a result of implementing system instrumentation at the application server level. This instrumentation approach inserted no extra proxy layers between clients and targeted EJB components, thus avoiding an additional level of indirection. Though, noticeable delays were recorded during the actual application adaptation operations. This was caused by client requests being postponed until the required EJB-swapping procedures were completed. Nonetheless, no client requests were refused and no client transactions expired during the tests. It is important to note that the goal of testing AQuA_J2EE on the sample Duke's Bank application was to prove its automatic management and optimisation potential. The tests showed how the monitoring, adaptation logic and execution functionalities worked and how they could be used in the tested scenarios. Performed tests were not intended to exhaustively prove that the current AQuA_J2EE prototype was able to optimise the performance of any J2EE application in the exact manner.
The proposed monitoring data analysis and learning algorithm was tested on real monitoring samples collected during Duke's Bank execution. Obtained test results indicated the potential of the proposed learning approach to categorise available data into distinctive information

clusters that could subsequently be used to predict component performance. (section 5.3).

6.6 Limitations and Research Opportunities

This thesis presents a dynamic performance optimisation solution based on the automatic selection and swapping of redundant components, in varying environmental conditions. The AQuA framework and AQuA_J2EE prototype were devised as part of this work to automatically perform the necessary management operations associated with the proposed optimisation solution. The goal of the thesis was to present the component-redundancy based optimisation approach, exemplify execution scenarios in which the approach would prove beneficial and show how an automatic framework can be employed to perform the required management operations for implementing this approach. The thesis attained these goals as presented in chapters 3, 4 and 5. The completed work can be advanced by extending its constituent parts, or integrating existing solutions from relevant research areas.

Further studies can advance the analysis on the impact that various designs, implementation and configuration choices have on system performance. This work would help identify the most common situations where redundant components can be implemented and used. The most common or significant cases identified can subsequently be documented in a comprehensive specification, similar to performance design patterns, or anti-patterns.

The goal of devising the AQuA framework was to indicate the main functionalities required to provide automatic management support for the proposed component-redundancy based solution. The AQuA_J2EE prototype was implemented in order to exemplify how the AQuA framework can be implemented and used to automatically manage system performance at runtime. Each of the framework's functionalities can be further extended so as to feature increasingly complicated behaviours and gradually be able to handle more complex management scenarios. As such, possible research efforts can be directed towards extending each of the AQuA functional capabilities, including the monitoring, anomaly detection, learning, component evaluation, adaptation decision and component hot-swapping functions. For example, the applicability of data mining algorithms, machine learning techniques, statistical approaches and knowledge management solutions to AQuA's learning capability can be investigated. Another important research direction is concerned with the manner in which values of different monitored metrics, at various system levels, can be correlated and interpreted so as to identify and pin-point performance bottlenecks and their exact causes. The efficiency of the component-swapping mechanism can be further optimised, by allowing multiple redundant components to run in parallel. This would avoid the situation in which new incoming requests received during swapping operations are delayed until the termination of the current client sessions that execute on the old redundant components. The way adaptation logic based on other types of decision policies, such as goal-oriented policies, can be applied in the context of presented solution can be investigated. AQuA_J2EE can be enhanced so as to allow decision policies to be written, deleted, or configured at runtime, after the managed system has started executing.

BIBLIOGRAPHY

[1] Takoua Abdellatif. Enhancing the management of a j2ee application server using a component-based architecture. In *Proceedings of the 31th IEEE/Euromicro Conference, CBSE Track:Component-based software engineering*, Porto, Portugal, 2005.

[2] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns - Best Practices and Design Strategies*. Sun Microsystems Press, Prentice Hall, 2001.

[3] J. Appavoo and et.al. Enabling autonomic behavior in systems software with hot swapping. *IBM SYSTEMS JOURNAL*, 42(1):60–76, 2003.

[4] F. Aquilani, S. Balsamo, and P. Inverardi. Performance analysis at the software architectural design level. *Performance Evaluation*, 45(2-3), July 2001.

[5] A. Avizienis. *The Methodology of N-Version Programming, Software Fault Tolerance*. John Wiley & Sons Ltd., 1995.

[6] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *COMPSAC*, pages 149–155, 1977.

[7] J. Bailey, A. Poulovassilis, and P. Wood. An event-condition-action language for xml. In *11th International World Wide Web Conference (WWW2002)*, Honolulu, Hawaii, 2002.

[8] J. Balasubramanian, B. Natarajan, J. Parsons, D. C. Schmidt, and A. Gokhale. Middleware support for dynamic component updating. In *International Symposium on Distributed Objects and Applications (DOA 2005)*, October 2005. Agia Napa, Cyprus.

[9] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Software performance: state of the art and perspectives. Technical report, Dipartimento di Informatica, Universita dell'Aquila, December 2002.

[10] Arosha K Bandara, Emil C Lupu, Jonathan Moffett, and Alessandra Russo. A goal-based approach to policy refinement. In *IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2004)*, 2004.

[11] Mario Barbacci, Mark H. Klein, Thomas H. Longstaff, and Charles B. Weinstock. Quality attributes. Technical Report CMU/SEI-95-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1995.

[12] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering, Addison Wesley Inc., December 1997.

[13] Len Bass, Paul Clements, and Rick Kazman. Software architecture in practice. In *SEI Series in Software Engineering*. Addison Wesley Inc., December 1997.

[14] Don Batory. On the reusability of query optimization algorithms. *Information Science*, 49:177–202, 1989.

[15] M. Bearden, S. Garg, and W. Lee. Integrating goal specification in policy-based management. In *2nd International Workshop on Policies for Distributed Systems and Networks*, 2001.

[16] Philippe Boinot, Renaud Marlet, Gilles Muller, and Charles Consel. A declarative approach for designing and developing adaptive components. In *International Conference on Automatic Software Engineering (ASE2000)*, 2000.

[17] Andrea Bondavalli, Felicita Di Giandomenico, and Jie Xu. A cost-effective and flexible scheme for software fault tolerance. *Journal of Computer Systems Science and Engineering*, 8:234–244, 1993.

[18] J. Bosch and P. Molin. Software architecture design: Evaluation and transformation. In *IEEE Conference and Workshop on Engineering of Computer-Based Systems*, Nashville, Tennessee, USA, March 1999.

[19] S. Bouchenak, F. Boyer, D. Hagimont, S. Krakowiak, A. Mos, N. de Palma, V. Quema, and J. B. Stefani. Architecture-based autonomous repair management: An application to j2ee clusters. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS)*, Orlando, Florida, October 2005.

[20] G. Candea, E. Kiciman, S. Kawamoto, and Armando Fox. Autonomous recovery in componentized internet applications. *Cluster Computing Journal*, 2005. Kluwer Academic Publishers.

[21] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of ejb applications. In *Proceedings of 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 246–261, Seattle, Washington, USA, 2002.

[22] Hoi Chan, Alla Segal, Bill Arnold, and Ian Whalley. How can we trust a policy system to make the best decision? In *2nd International Conference on Autonomic Computing*, Seattle, Washington, USA, June 13-16 2005.

[23] S.-W. Cheng, D. Garlan, B. Schmerl, J. P. Sousa, B. Spitznagel, and P. Steenkiste. Using architectural style as a basis for self-repair. In *Software Architecture: System Design, Development, and Maintenance (Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture)*, pages 45–59, August 2002.

[24] I. Cohen, M. Goldszmit, T. Kelly, J. Symons, and J. S. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *6th USENIX OSDI*, San Francisco, CA, December 2004.

[25] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP*, Brighton, United Kingdom, October 2005.

[26] V. Cortellessa and R. Mirandola. Deriving a queueing network based performance model from uml diagrams. In *In WOSP2000 Conference Proceedings*. ACM, September 2000.

[27] Rob Cutlip. Self-managing systems. the optimization challenge. IBM developerWorks, Autonomic computing, online article series: http://www-128.ibm.com/developerworks/library/ac-selfo/?ca=dnt-634, 23 August 2005.

[28] A. Diaconescu, A. Mos, and J. Murphy. Automatic performance management in component-based software systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC04)*, pages 214–221, New York, USA, May 2004.

[29] A. Diaconescu and J. Murphy. A framework for using component redundancy for self-adapting and self-optimising component-based enterprise systems. In *The ACM SIGPLAN Student Research Competition (3rd place), International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2003)*, pages 390–391, Anaheim, California, USA, October 2003.

[30] A. Diaconescu and J. Murphy. A framework for using component redundancy for self-optimising and self-healing component based systems. In *Workshop on Software Architectures for Dependable Systems (WADS), International Conference on Software Engineering (ICSE2003)*, Hilton Portland, Oregon USA, May 2003.

[31] A. Diaconescu and J. Murphy. A framework for automatic performance monitoring, analysis and optimization of component based software systems. In *Workshop on Remote Analysis and Measurement of Software Systems (RAMSS), International Conference on Software Engineering (ICSE 2004)*, Edinburgh, Scotland, UK, 2004.

[32] Ada Diaconescu and John Murphy. Quality of service in wide area distributed systems. In *Proceedings of the Information Technology and Telecommunications conference*, Waterford Institute of Technology, Ireland, 30-31 October 2002.

[33] Ada Diaconescu and John Murphy. Automating the performance management of component-based enterprise systems through the use of redundancy. In *the 20th IEEE/ACM International Conference on Automated Software Engineering*, Long Beach, California, USA, November 2005.

[34] Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus. Managing web server performance with autotune agents. *Autonomic Computing, IBM Systems Journal*, 42(1), 2003.

[35] M.S. Dias and D.J. Richardson. Issues in analyzing dynamic system evolution. In *Software Engineering and Applications*, November 2003.

[36] Minh B. Do and Subbarao Kambhampati. Sapa: A scalable multi-objective metric temporal planner. In *Journal of Artificial Intelligent Research (JAIR 2003)*, 2003.

[37] Jim Dowling and Vinny Cahill. The k-component architecture meta-model for self-adaptive software. In *Reflection 2001*. LNCS 2192, 2001.

[38] P. Ebraert, T. D'Hondt, and T. Mens. Enabling dynamic software evolution through automatic refactoring. In Ying Zhou and James R. Cordy, editors, *Proc. 1st Int'l Workshop on Software Evolution Transformations*, pages 3–6, November 2004.

[39] M. P. J. Fromherz, L. S. Crawford, C. Guettier, and Y. Shang. Distributed adaptive constrained optimization for smart matter systems. *Intelligent Distributed and Embedded Systems*, pages 34–39, 2002. Papers from the 2002 AAAI Spring Symposium; 2002 March 25-27, Palo Alto, CA. Menlo Park, CA: AAAI Press;.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object Oriented Software*. Addison-Wesley, NY, 1995.

[41] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object Oriented Software*. Addison-Wesley, NY, 1995.

[42] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1), 2003.

[43] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-bases self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), October 2004.

[44] D. Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)*, pages 18–19, November 2002.

[45] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. ISBN-0-13-820432-2. Prentice Hall Inc., 1991.

[46] C. Hrischuk, J. Rolia, and C. M. Woodside. Automatic generation of a software performance model using an object-oriented prototype. In *Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 399–409, Durham, NC, January 1995.

[47] M. N. Huhns and V. T. Holderfield. Robust software. *Agents on the Web, IEEE Internet Computing*, March/April 2002.

[48] IEEE. *IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology. New York: Institute of Electrical and Electronics Engineers*, 1990.

[49] IEEE. *IEEE Standard 1061-1992 . Standard for a Software Quality Metrics Methodology. New York: Institute of Electrical and Electronics Engineers*, 1992.

[50] J K. Hui, Appavoo, R. Wisniewski, M. Auslander, D. Edelsohna, B. Gamsa, O. Krieger, B. Rosenburg, and M. Stumm. Position summary: Supporting hot-swappable components for system software. In *HOTOS*, 2001.

[51] L. P. Kaelbling, M.l L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[52] S. Kambhampati and B. Srivastava. The case for automated planning in autonomic computing. In IEEE Computer Society, editor, *the Second International Conference on Autonomic Computing*, 2005.

[53] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The architecture tradeoff analysis method. In *Proceedings of ICECCS98*, pages 68–78, Monterey, CA, August 1998.

[54] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, January 2003.

[55] J. O. Kephart and W. E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *POLICY '04: Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04)*, page 3, Washington, DC, USA, 2004. IEEE Computer Society.

[56] P. Kähkipuro. Uml based performance modelling framework for object-oriented distributed systems. In *Proc. 2nd International Conference on the Unified Modeling Language: beyond the standard, UML '99*, pages 356–371, 1999.

[57] G. Kiczales. Beyond the black box: Open implementation. *IEEE Software*, January 1996.

[58] M. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, and H. Lipson. Attribute-based architecture styles. In *in Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, pages 225–243, San Antonio, TX, 1999.

[59] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[60] M. M. Kokar, K. Baclawski, and Y. A. Eracar. Control theory-based foundations of self-controlling software. *IEEE Intelligent Systems*, 14(3):37–45, May/June 1999.

[61] R. Laddaga. Creating robust software through self-adaptation. *IEEE Intelligent Systems*, 14(3):26–29, May/June 1999. (guest editor's introduction).

[62] J. C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *15th Int. Symp. on Fault Tolerant Computing (FTCS-15)*, pages 2–11, Ann Arbor, MI, USA, 1985.

[63] B. Littlewood, P. Popov, and L. Strigini. Modeling software design diversity: a review. *ACM Press, New York NY, USA*, pages 177–208, 2001.

[64] C. Liu and D. J. Richardson. Raic: Architecting dependable systems through redundancy and just-in-time testing. In *Workshop on Architecting Dependable Systems (WADS), ICSE*, Orlando, Florida, US, 2002.

[65] J. Matevska-Meyer, S. Olliges, and W. Hasselbring. Runtime reconfiguration of j2ee applications. In *DÉCOR*, pages 77–84, 2004.

[66] A. Mos. *A Framework for Adaptive Monitoring and Performance Management of Component-Based Enterprise Applications*. PhD thesis, Dublin City University, Ireland, 2004.

[67] A. Mos and J. Murphy. Performance management in component-oriented systems using a model driven architecture approach. In *The 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, Lausanne, Switzerland, September 2002.

[68] A. Mos and J. Murphy. Compas: Adaptive performance monitoring of component-based systems. In *Workshop on Remote Analysis and Measurement of Software Systems (RAMSS) at 26th International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, UK, May 2004.

[69] D. J. Musliner, R. P. Goldman, M. J. Pelican, and K. D. Krebsbach. Self-adaptive software for hard real-time environments. *IEEE Intelligent Systems*, 14(3):23–29, May/June 1999.

[70] Object Management Group (OMG). *UML documentation*. http://www.rational.com/uml/resources/documentation/.

[71] Object Management Group (OMG). *UML Profile, for Schedulability, Performance, and Time*. OMG document ptc/2002-03-02, http://www.omg.org/cgi-bin/doc?ptc/2002-03-02.

[72] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May-June 1999.

[73] Noel De Palma, Sara Bouchenak, Daniel Hagimont, and Fabienne Boyer. Autonomic administration of clustered j2ee applications. In *International Multi-Conference in Computer Science & Computer Engineering*, Las Vegas, Nevada, US, June 2005.

[74] T. Parsons and J. Murphy. Data mining for performance antipatterns in component based systems using run-time and static analysis. In *Transactions on Automatic Control & Control Science*, volume 49, pages 113–118, May 2004.

[75] Rob Pooley and Peter King. The unified modeling language and performance engineering. In *IEE Proceedings on Software*, February 1999.

[76] P. Popov, L. Strigini, and A. Romanovsky. Diversity for off-the-shelf components. In *International Conference on Dependable Systems & Networks - Fast Abstracts*, pages B60–B61, New York, USA, 2000.

[77] Marija Rakic and Nenad Medvidovic. Increasing the confidence in off-the-shelf components: A software connector-based approach. In *In Proceedings of the 2001 Symposium on Software Reusability (SSR 2001)*, 2001.

[78] Brian Randell and Jie Xu. *The Evolution of the Recovery Block Concept*. Software Fault Tolerance, Lyu, John Wiley & Sons Ltd, 1995.

[79] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 205–230, London, UK, 2002. Springer-Verlag.

[80] E. Roman, S. W. Ambler, and T. Jewell. *Mastering Enterprise JavaBeans*. J. Wiley & Sons, second edition, USA and Canada, 2002. ISBN: 0-471-41711-4.

[81] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons; 1 edition, September 2000.

[82] D. C. Schmidt and F. Buschman. Patterns, frameworks, and middleware: Their synergistic relationship. In *25the International Conference on Software Engineering (ICSE)*, pages 694–704, Portland, Oregon, May 2003.

[83] Douglas C. Schmidt, Rick Schantz, Mike Masters, Joseph Cross, David Sharp, and Lou DiPalma. Towards adaptive and reflective middleware for network-centric combat systems. *CrossTalk*, November 2001.

[84] C. U. Smith and L. G. Williams. Software performance engineering: A case study with design comparisons. *IEEE Transactions on Software Engineering*, 19(7), July 1993.

[85] Software Engineering Institute, Carnegie Mellon University. *Predictable Assembly from Certifiable Components (PACC)*. http://www.sei.cmu.edu/pacc/.

[86] Software Engineering Institute, Carnegie Mellon University. *Software Architecture and the Architecture Tradeoff Analysis Initiative*,. http://www.sei.cmu.edu/ata/.

[87] Software Engineering Institute, Carnegie Mellon University. *Component Based Software Development / COTS Integration, Software Technology Review*, 1997. http://www.sei.cmu.edu/str/descriptions/cbsd.html.

[88] Software Engineering Institute, Carnegie Mellon University. *COTS and Open Systems - An Overview, Software Technology Review*, 1997. http://www.sei.cmu.edu/str/descriptions/cots.html.

[89] Biplav Srivastava, Joseph P. Bigus, and Donald A. Schlosnagle. Using able to bring planning to business applications. *American Association for Artificial Intelligence (www.aaai.org)*, 2004.

[90] J. Strassner. How policy empowers business-driven device management. In *3rd International Workshop on Policies for Distributed Systems and Networks*, 2002.

[91] C. Szyperski and et al. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, Great Britain, November 2002.

[92] Clemens Szyperski. Component technology - what, where, and how? In *Proceedings of the 25the International Conference on Software Engineering (ICSE03)*, pages 684–693, Portland, Oregon, 3-10 May 2003.

[93] C. Tapus, I-H. Chung, and J. K. Hollingsworth. Active harmony: Towards automated performance tuning. In *Super Computing*, November 2002.

[94] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. A component- and message-based architectural style for gui software. *IEEE Transactions on Software Engineering*, 22(6):390–406, 1996.

[95] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *SIGOPS Oper. Syst. Rev.*, pages 239–254, 2002.

[96] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in shared hosting platforms. *SIGOPS Oper. Syst. Rev.*, 36(SI):239–254, 2002.

[97] A. I. Verkamo, J. Gustafsson, L. Nenonen, and J. Paakki. Design patterns in performance prediction. In *Proc. 2nd International Workshop on Software and Performance (WOSP'00)*, Ottawa, Canada, September 2000.

[98] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser. Terminology for policy-based management. Technical report, University of Wolverhamptons School of Computing and Information Technology, 2001.

[99] Jules White, Douglas Schmidt, and Aniruddha Gokhale. Simplifying the development of autonomic enterprise java bean applications via model driven development. In *ACM/IEEE 8th Internation Conference on Model Driven Engineering Languages and Systems (MoDELS/UML2005)*, Montego Bay, Jamaica, October 2005.

[100] Lloyd G. Williams and Connie U. Smith. Performance evaluation of software architectures. In *Proceedings of the Workshop on Software and Performance (WOSP98)*, Santa Fe, NM, October 1998.

[101] Daniel M. Yellin. Competitive algorithms for the dynamic selection of component implementations. *IBM Systems Journal*, 42(1), 2003.

[102] S. Zang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *DSN*, 2005.

APPENDIX

A

Instrumenting JBoss

JBoss's implementation was modified in order to allow dynamic monitoring and swapping of redundant components.

A.1 JBoss Integration with COMPAS

JBoss was instrumented so as to send monitoring events to the COMPAS monitoring and diagnosis tool. These include method request and response events and component instantiation and destruction operations. The purpose was to use COMPAS's capabilities to graphically display monitoring data and raise performance alerts during runtime. This facility can be used by a human administrator to visualise and analyse performance information during system execution. In case performance anomalies are detected, the administrator can decide to replace non-optimal components with redundant variants, so as to remedy the problem. The redundant component-swapping GUI (section 4.7) can be used for this purpose, to dynamically adapt the managed application.

The code in Listing A.1 shows the way the JBoss application server was modified so as to be integrated with the COMPAS monitoring tool. The listed code belongs to JBoss' `LogInterceptor` class (i.e. `org.jboss.ejb.plugins.LogInterceptor`). This container interceptor class was instrumented so as to dynamically extract monitoring events and send them to a COMPAS instance for further processing. In turn, COMPAS uses received events to calculate and graphically display performance data on monitored component methods. Displayed data includes method response times and throughputs, as well as the number of instances available for each EJB component used.

When the LogInterceptor is instantiated, it creates an object of the `ProxyImplementor` COMPAS class (Listing A.1, line 27). The `ProxyImplementor`'s constructor receives as parameters the name of the application server used, the EJB class name, the EJB JNDI name and the type of EJB container used (i.e., StatelessSessionContainer, StatefulSessionContainer, or EntityContainer). Then, during runtime, whenever an EJB's method is invoked, the `invoke` method of the EJB container's `LogInterceptor` is also called (i.e. Figure xx-b, line 56: `returnedObject = getNext().invoke(invocation);`). The `LogInterceptor`'s invoke method was instrumented so as to send method invocation events to its associated `ProxyImplementor` object. The `ProxyImplementor` object is the connection between JBoss and the COMPAS monitoring tool. Similarly, when

JBoss was integrated with the AQuA_J2EE framework, an `RGManager` object was used to make this connection (subsection A.2). One method invocation event is sent to the `ProxyImplementor` before the actual invocation is forwarded to the targeted EJB instance (Listing A.1, line 51: `proxyImpl.preMethodInvocation();`). A second event is sent after a response is returned from the invoked EJB (Listing A.1, line 62: `proxyImpl.postMethodInvocation( methodName );`).

Listing A.1: Instrumenting JBoss LogInterceptors to send monitoring events to COMPAS

```
1  import edu.dcu.pel.compas.monitoring.probe.ProxyImplementor;
2
3  public class LogInterceptor extends AbstractInterceptor
4  {
5
6  protected ProxyImplementor proxyImpl;
7
8  public void create(){
9
10 //JBoss original code
11 //...
12
13 //PEL container instrumentation code
14 //...
15
16 //PEL code for COMPAS integration
17
18 //get the current container type: Session or Entity
19 //e.g., bean type = org.jboss.ejb.StatelessSessionContainer
20 String ejbContainerType = getContainer().getClass().getName();
21 String simpleEJBContainerType = "session";//"session" or "entity"
22 if( ejbContainerType.toLowerCase().indexOf("entity") != -1 ){
23 simpleEJBContainerType = "entity";
24 }
25
26 //create ProxyImplementor instance   one per container
27 proxyImpl = new ProxyImplementor( "JBoss", ejbClass, ejbName, simpleEJBContainerType
       );
28
29 //....
30
31 }//create
32
33 //
34 //-----------------------------
35 //
36
37 public Object invoke(Invocation invocation) throws Exception{
38
39 //original JBoss code
40 // ...
41 //original JBoss code commented by PEL
42 //// return getNext().invoke(invocation);
43
44 //      //PEL code for integration with COMPAS
45
46 //call preMethodInvocation only if this invocation is not for a remove operation
```

```
47  //  if method name != remove
48  if (!isRemove){
49
50  //method pre-invoke
51  proxyImpl.preMethodInvocation();
52
53  }//if
54
55  //get the result returned by the next interceptor in the chain
56  returnedObject = getNext().invoke(invocation);
57
58  //call postMethodInvocation only if method name != remove
59  if(!isRemove){
60
61  //method post-invoke
62  proxyImpl.postMethodInvocation( methodName );
63
64  }//if
65
66  //return the invocation result
67  return returnedObject;
68
69  //      //end PEL code
70
71  }//invoke
72
73
74  }//LogInterceptor
```

A.2 JBoss Integration with AQuA_J2EE

JBoss was instrumented in order to send monitoring events to AQuA_J2EE. The instrumentation approach was similar to the ne taken for integrating JBoss with COMPAS, as presented in the previous section. Namely, JBoss's LogInterceptor class (i.e. `org.jboss.ejb.plugins.LogInterceptor`) was modified so as to send method invocation and response events to AQuA_J2EE, more precisely to instances of the `RGManager` class. The JBoss `LogInterceptor create()` method was modified to obtain an instance of the `RGManager` class (Listing A.2, lines 1-84). The `LogInterceptor invokeHome` and `invoke` methods were modified to send monitoring events on intercepted methods to the associated `RGManager` instance A.2, lines 90-175 and 178-212, respectively). The instrumentation code is explained through the inserted implementation comments.

Listing A.2: Instrumenting JBoss LogInterceptors to send monitoring events to AQuA_J2EE

```
1  public void create() throws Exception{
2
3      ////JBoss original code
4      super.start();
5
6      ////PEL added code
7
```

```
8       //get metadata on the new container being created and
9       //  on its managed EJB component
10      md = getContainer().getBeanMetaData();
11      ejbClassName = md.getEjbClass();
12      ejbName=md.getEjbName();
13      jndiName=md.getJndiName();
14
15      //find out whether this is a stateful container
16      //(Stateful Session container)
17      isStatefulSessionContainer = this.isStatefullContainer();
18
19      ////code for creating a local AQuA_J2EE framework instance for this EJB component
            /container
20
21      //check if RGManager for this component exists
22      //get the singleton  RGManagersAdministrator instance
23      RGManagersAdministrator admin = RGManagersAdministrator.
            getRGManagersAdministrator();
24      //get the RG name of the RGManager instance that manages this component
25      //  (given the component JNDI name)
26      String rgName = admin.getRGManagerName( jndiName );
27      if( null != rgName ){//a RGManager instance already exists for this component
28
29      //get the RGManager instance for this component
30      this.rgManager = admin.getRGManagerInstance( rgName );
31      //verify if this RGManager is currently active
32      this.isManaging = rgManager.isActive();
33
34      if( null == this.rgManager ){
35
36          //should NOT happen => warning message
37          //messages = warning messages....
38          //MessagePrinter.printWarning( this, "create", messages );
39
40          //instantiate a RGManager
41          this.rgManager = this.instantiateRGManger( jndiName, ejbClassName );
42          this.isManaging = rgManager.isActive();
43          if( this.isManaging ){
44              this.timerExtractionStrategy = rgManager.getTimeExtractionStrategy();
45              this.isTimeInNanos = this.timerExtractionStrategy.inNanos();
46          }
47      }//if
48      else{// => this.rgManager != null
49          //verify if this RGManager should be active
50          this.isManaging = rgManager.isActive();
51          if( this.isManaging ){
52              this.timerExtractionStrategy = rgManager.getTimeExtractionStrategy();
53              this.isTimeInNanos = this.timerExtractionStrategy.inNanos();
54          }
55      }
56
57
58      }//if   RGManager instance exists
59
60      else {
61          // =>  NO RGManager instance exists for this component
62          // => create a RGManager instance
63
```

```
64          //instantiate RGManager, using:
65          //  jndi name of the managed EJB component   unique identifier
66          //  class of the managed EJB - used to get the monitored methods
67          this.rgManager = this.instantiateRGManger( jndiName, ejbClassName );
68          //determine if this is should be an active RGManager
69          this.isManaging = this.rgManager.isActive();
70          if( this.isManaging ){
71              this.timerExtractionStrategy = rgManager.getTimeExtractionStrategy();
72              this.isTimeInNanos = this.timerExtractionStrategy.inNanos();
73          }//if
74
75      }//else
76
77
78      //end PEL added code
79
80      //JBoss original code
81      ejbName = md.getEjbName();
82    callLogging = md.getContainerConfiguration().getCallLogging();
83
84  }//end create
85
86  //
87  //-------------------------------
88  //
89
90  public Object invokeHome(Invocation invocation) throws Exception{
91
92      ////JBoss original code
93      //...
94
95      ////PEL added code for intercepting messages and collecting monitoring data
96
97      long methodResponseTime = 0;
98    double methodResponseTimeMillis = 0;
99    long methodInvocationTime = 0;//instance in time when a method is being called
100   long methodInvocationCompletedTime = 0;//instance in time when a method returns
101
102     //verify that this container has an active, non-null RGManager instance
103     //associated with it
104   if( (this.isManaging) && (null != this.rgManager) ){
105
106       //get the current time
107       methodInvocationTime = this.timerExtractionStrategy.getCurrentTime();
108       //signal the method invocation event to the associated RGManager instance
109         //  send the JNDI name of this component and the Method invoked
110         this.rgManager.methodInvoked( this.jndiName, invocation.getMethod() );
111
112   }
113
114   //original JBoss code, commented and modified by PEL
115   //return getNext().invokeHome(invocation);
116
117     //verify that there is no hot swapping being carried out at the moment
118   if( false == Container.isHotSwapping ){
119
120         //get the response from the next interceptor down the container interceptor
              chain
```

```
121     Object objAux = getNext().invokeHome(invocation);
122
123     if( (this.isManaging) && (null != this.rgManager) ){
124             //get the current time
125         methodInvocationCompletedTime =
126             this.timerExtractionStrategy.getCurrentTime();
127         //calculate the method response time
128             methodResponseTime =
129                 methodInvocationCompletedTime - methodInvocationTime;
130         //convert the response time in milliseconds
131             methodResponseTimeMillis =
132                 this.getTimePeriodInMillis( methodResponseTime );
133
134             //signal the method invocation completion event to the RGManager instance
                    associated
135             //send the JNDI name of this component, the invoked Method and the
                    response time
136         this.rgManager.methodInvocationCompleted( this.jndiName,
137             invocation.getMethod(), methodResponseTimeMillis );
138     }
139
140         //return the method invocation result
141     return objAux;
142
143 }
144
145         //in case the container is executing a hot swapping operation
146     else{  //true == LogInterceptor.delayRequests
147         if( this.isStatefulSessionContainer ){//this container manages a session bean
148
149             //induce delays until hot swap is complete
150                 while( true == Container.isHotSwapping ){
151                     Thread.sleep( 1000 );//[ms]
152           }//while
153
154         //hot swap is complete at this point
155
156         }//if
157
158         Object objAux = getNext().invokeHome(invocation);
159
160         if( (this.isManaging) && (null != this.rgManager) ){
161             methodInvocationCompletedTime = this.timerExtractionStrategy.
                    getCurrentTime();
162             methodResponseTime = methodInvocationCompletedTime - methodInvocationTime
                    ;
163             methodResponseTimeMillis = this.getTimePeriodInMillis( methodResponseTime
                    );
164
165             this.rgManager.methodInvocationCompleted( this.jndiName, invocation.
                    getMethod(), methodResponseTimeMillis );
166         }
167
168         return objAux;
169
170     }//else
171
172     //end PEL added code
```

```
173
174 }//end invokeHome
175
176 //
177 //-------------------------------------
178 //
179
180 public Object invoke(Invocation invocation) throws Exception{
181
182     ////JBoss original code
183     //...
184
185     ////PEL added code to intercept messages and collect monitoring data
186     double methodResponseTimeMillis = 0;
187     long methodResponseTime = 0;
188     long methodInvocationTime = 0;//instance in time when a method is being called
189     long methodInvocationCompletedTime = 0;//instance in time when a method returns
190
191     if( (this.isManaging) && (null != this.rgManager) ){
192
193         this.rgManager.methodInvoked( this.jndiName, invocation.getMethod() );
194         methodInvocationTime = this.timerExtractionStrategy.getCurrentTime();
195         //forward method invocation on the next interceptor and get the response
196         Object objAux = this.getNext().invoke( invocation );
197
198         methodInvocationCompletedTime = this.timerExtractionStrategy.getCurrentTime()
                ;
199         methodResponseTime = methodInvocationCompletedTime - methodInvocationTime;
200         methodResponseTimeMillis = this.getTimePeriodInMillis( methodResponseTime );
201
202         this.rgManager.methodInvocationCompleted( this.jndiName, invocation.getMethod
                (),
203                                 methodResponseTimeMillis );
204
205         //return the method invocation response
206         return objAux;
207     }
208     else{//this is not a managing interceptor container => use original JBoss code
209     //original JBoss code:
210             return getNext().invoke(invocation);
211     }
212
213     ////end PEL added code
214 }
```

An additional application was implemented to collect workload information from servlet components executing on JBoss. The JBoss distribution used in the experimental work was integrated with the Tomcat Web server to provide support for servlet components. JBoss's JMX infrastructure was used to extract runtime information on the incoming servlet requests. This data was used to calculate workloads on the tested applications' web tiers and compare it with workloads measured on the EJB application tier. The implementation code for obtaining this information is provided in Listing A.3 and explained through the inline comments.

Listing A.3: Obtaining servlet workload information via JBoss's JMX infrastructure

```
1  package monitoring;
2
3  import java.util.Properties;
4  import javax.management.ObjectName;//from: jmx-basic.jar
5  import javax.management.j2ee.statistics.TimeStatistic;//jboss-client.jar, jboss-jsr77
       .jar
6  import javax.naming.InitialContext;//jmx-basic.jar
7  import javax.naming.Context;
8  import org.jboss.jmx.adaptor.rmi.RMIAdaptor;//jmx-adaptor-plugin.jar
9  import org.jboss.management.j2ee.statistics.ServletStatsImpl;
10
11 //...
12
13 public class ServletMonitor {
14
15     private static final String PROVIDER_URL = "ada-dell.pel.eeng.dcu.ie";
16     private static final String SERVLET_NAME = "jboss.management.local:
           J2EEApplication
17 ␣␣␣␣=3statefulDukesBank.ear,J2EEServer=Local,
18 ␣␣␣␣WebModule=web-client.war,j2eeType=Servlet,name=Dispatcher";
19
20     private RMIAdaptor server = null;
21     private ObjectName objName = null;
22
23     //constructor
24     public ServletMonitor(){
25
26         try{
27             //set the initial context properties
28             Properties props = new Properties();
29             props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
30                 "org.jnp.interfaces.NamingContextFactory");
31             props.setProperty(Context.PROVIDER_URL, ServletMonitor.PROVIDER_URL );
32             props.setProperty("java.naming.factory.url.pkgs",
33                     "org.jboss.naming:org.jnp.interfaces" );
34
35             //create the initial context with the set properites
36             InitialContext initCtx = new InitialContext( props );
37
38             //lookup the targeted RMIAdaptor instance in the initial context
39             server = (RMIAdaptor)initCtx.lookup("jmx/rmi/RMIAdaptor");
40             //create the object name fot the targeted servlet to be monitored
41             objName = new ObjectName( ServletMonitor.SERVLET_NAME );
42         }
43         catch(Exception ex){
44             ...
45         }
46     }//end constructor
47
48
49     //returns the number of requests received by the monitored servlet
50     public long getServletRequestCount(){
51
52         long servletRequestCount = 0;
53         try{
```

```
54              //get the statistics info on the targeted monitored servlet
55              ServletStatsImpl stats = (ServletStatsImpl)server.getAttribute( objName,
                    "Stats");
56              //get the time statistics from the general servlet statistics
57              TimeStatistic timeStatistic = stats.getServiceTime();
58
59              //get the servlet request count from the time statistics
60              servletRequestCount = timeStatistic.getCount();
61
62          }
63          catch(Exception ex){
64              ...
65          }
66
67          //return the servlet request count
68          return servletRequestCount;
69
70      }//end getServletRequestCount
71
72  //example of how to use the ServletMonitr class for getting the servlet request count
        :
73  //  public static void main( String[] args ){
74  //      ServletMonitor monitor = new ServletMonitor();
75  //      monitor.getServletRequestCount();
76  //  }
77
78  }//end ServletMonitor class
```

A.3 JBoss Instrumentation for EJB Component-Swapping

JBoss was modified to support dynamic swapping of EJB components, without necessitating functional interruptions. The JBoss hot-swapping functionality was used as part of this goal. However, the available JBoss implementation did not support swapping of EJBs while under heavy user loads (section 4.7). JBoss EJB containers (e.g. the `org.jboss.ejb.Container` and `org.jboss.ejb.EntityContainer` classes) were modified to solve the existing issues and support runtime swapping of EJB components (Listing A.4). The implemented strategy is described in section 4.7 and explained via inline comments below.

Listing A.4: instrumenting JBoss for component-swapping support

```
1  //// org.jboss.ejb.Container class
2  //PEL variables
3      public static final String STATELESS_SESSION_CONTAINER_TYPE = "
           StatelessSessionContainer";
4      public static final String STATEFUL_SESSION_CONTAINER_TYPE = "
           StatefulSessionContainer";
5      public static final String ENTITY_CONTAINER_TYPE = "EntityContainer";
6
7      public static boolean isHotSwapping = false;
8   //end PEL variables
```

```
9
10    //PEL methods
11            //gets the container type: stateless, statefull or entity
12     public String getContainerType(){
13         String containerType = null;
14         containerType = this.getClass().getName();
15         return containerType;
16     }
17         //marks the fact that the container is currently carrying out a hot swapping
               operation
18     public static void setIsHotSwapping( boolean isHotSwapping ){
19         Container.isHotSwapping = isHotSwapping;
20     }
21
22   //end PEL methods
23
24 //
25
26 ////org.jboss.ejb.EntityContainer class
27 protected void stopService() throws Exception
28    {
29
30         //PEL code added for hot swap
31         //waits for zero workload on this EJB
32             //  and then flushes the Entity EJB cache
33
34         //test if hot swapping is being carried out at the moment
35         if( true == Container.isHotSwapping ){
36
37             int methodsStillInUse = -1;
38
39             //delay the stopping action of
40             //  the container service
41             EntityCache cache = (EntityCache)this.getInstanceCache();
42
43             //delay the stopping the container service
44             //while there are still EJB instances
45             //in the cache
46             while( 0 != cache.getCacheSize() ){
47                 Thread.sleep( 1000 );//1 sec
48
49                 //verify if methods are still in use
50                 methodsStillInUse = this.getMethodsStillInUse();
51                 //if methods are not in use anymore,
52                 //  then flush the cache
53                 if( 0 == methodsStillInUse ){
54                     //delay 1 sec
55                     Thread.sleep( 1000 );
56                     cache.flush();
57                 }
58             }//while
59
60             //flush the cache
61             cache.flush();
62             cache = null;
63         }//if
64
65 //JBoss original code for stopping the container service
```

```
66  //...
67
68  }//end stopService
69
70  //////PEL defined methods
71
72    //verifies methods' loads to determine if the methods are still in use
73    //returns 0 if no methods are in use
74      private int getMethodsStillInUse(){
75          //get the monitored methods of this EJB component
76          //asks the RGManagerAdministrator using the component name
77          //...
78
79          //get the RGManager and MonitorDataDispatcher instances for this component
80          //...
81
82          //for each monitored method, get the MonitoringDataHandler instance
83          //use it to verify the method load
84          //...
85
86
87      }// getMethodsStillInUse
88
89  //-----------------------------------------------
90
91  ////org.jboss.ejb.EJBDeployer - class
92
93  //PEL modified methods
94
95      public void stop(DeploymentInfo di)
96        throws DeploymentException{
97
98          //PEL added code
99          //set the container isHotSwapping flag to true
100         org.jboss.ejb.Container.setIsHotSwapping(true);
101         //end PEL added code
102
103         //JBoss original code for stopping the service
104         //...
105
106     }//end stop
107
108
109   public synchronized void start(DeploymentInfo di)
110             throws DeploymentException {
111
112         //JBoss original code for starting the service
113         //...
114         //end of JBoss original code
115
116         //PEL added code
117         //set the container isHotSwapping flag to false
118         org.jboss.ejb.Container.setIsHotSwapping( false );
119         //end PEL added code
120
121     }//end start
```

APPENDIX

B

JBoss Configuration for Experimental Work

B.1 JBoss Configurations for EJB Containers

EJB *containers* are software entities that manage EJB instances at runtime. In JBoss, a container is an instance of the `org.jboss.ejb.Container` class. JBoss crates one separate container instance for each separate EJB configuration that is deployed. The `EJBDeployer` MBean manages the creation of the container instance.

JBoss externalises most of the EJB container setup, allowing for most of the container properties to be configured and customised. Such properties include the interceptors to use in the interceptor chain, as well as security, persistence, transaction policy, caching and pooling configurations. xml files are used for specifying the EJBs container configurations. The correct xml format for container configuration files is specified in a standard .dtd file, specific to the JBoss server. For example, xml container configuration files for JBoss 3.2.X must conform to the format specified in the standard `jboss_3_2.dtd` file. The `.dtd` file also provides the full list of configurable parameters for JBoss containers. A detailed description of EJB containers and their configuration in JBoss is available from section 4.1. This subsection details the caching and pooling configurations of JBoss EJB containers.

In JBoss, EJB containers are configured via container configuration files, in XML format. Container configurations can be made at two different levels. First, global, or default-level configurations can be specified for all instantiated containers. Such standard container configurations are performed via the `standardjboss.xml` file. Second, local-level configurations can also be made, specifically for each deployed EJB application. Such local container configurations are performed via the `jboss.xml` file. The local configuration file is archived together with the application files into a deployable application bundle (e.g., a `EJB.jar` application archive).

In the performed experimental tests, separate container configurations were specified for the individual EJBs involved. The corresponding `jboss.xml` files were used for this purpose to individually configure the various EJB components tested. Codes B.1 and B.2 exemplify a `jboss.xml` configuration file used to customise EJB containers. The example shows JBoss-specific configurations of two EJBs from the Duke's Bank Application (Section C), namely the

`Account` Entity EJB and the `AccountController` Stateful Session EJB.

Code B.1 shows the way customised containers can be specified to manage individual EJBs. Code B.2 subsequently shows the configurable elements that can be used to customise EJB containers with respect to caching and pooling functions.

Listing B.1: `jboss.xml` – enterprise beans configuration example

```
<jboss>

  <enterprise-beans>

    <entity>
      <ejb-name>AccountEJB</ejb-name>
      <jndi-name>MyAccount</jndi-name>
      <configuration-name>
        Custom_Cache_INSERT_after_ejbPostCreate_Container
      </configuration-name>
      <ejb-ref>
        <ejb-ref-name>ejb/customer</ejb-ref-name>
        <jndi-name>MyCustomer</jndi-name>
      </ejb-ref>
      <resource-ref>
        <res-ref-name>jdbc/BankDB</res-ref-name>
        <jndi-name>java:/DefaultDS</jndi-name>
      </resource-ref>
    </entity>

    <!-- ... other enterprise bean declarations ... -->

<enterprise-beans>

<!-- ... other declarations ... -->

</jboss>
```

Listing B.2: `jboss.xml` – custom container configuration example

```
<container-configurations>

    <container-configuration extends="Standard_CMP_2.x_EntityBean">
        <container-name>
            Custom_Cache_INSERT_after_ejbPostCreate_Container
        </container-name>
        <insert-after-ejb-post-create>
            true
        </insert-after-ejb-post-create>
        <instance-cache>
            org.jboss.ejb.plugins.InvalidableEntityInstanceCache
        </instance-cache>

        <container-cache-conf>
            <cache-policy>
                org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy
            </cache-policy>
            <cache-policy-conf>
                <min-capacity>10</min-capacity>
                <max-capacity>1000000</max-capacity>
```

```
21                  <overager-period>10</overager-period>
22                  <max-bean-age>10</max-bean-age>
23                  <resizer-period>10</resizer-period>
24                  <max-cache-miss-period>60</max-cache-miss-period>
25                  <min-cache-miss-period>1</min-cache-miss-period>
26                  <cache-load-factor>0.75</cache-load-factor>
27              </cache-policy-conf>
28          </container-cache-conf>
29
30          <container-pool-conf>
31              <MinimumSize>10</MinimumSize>
32          <MaximumSize>100</MaximumSize>
33              <strictTimeout>600000</strictTimeout>
34          </container-pool-conf>
35
36          <commit-option>A</commit-option>
37
38      </container-configuration>
39
40      <!-- ... other container configurations ... -->
41
42  </container-configurations>
```

The main elements that are used to specify EJB configurations in custom `jboss.xml` deployment descriptors are briefly explained below. The following elements are used to specify an EJB component in the deployment descriptor file:

- `<ejb-name>`: the name used to refer to this EJB. (Note: this name must match the `ejb-name` used for this EJB in the `ejb-jar.xml` file).
- `<jndi-name>`: the name assigned to this EJB in the JNDI directory.
- `<configuration-name>`: the name of the customised container to be used for managing this EJB.
- `<resource-ref>`: information on any resources that this EJB needs to use. For example, the `AccountEJB` EJB uses a database resource named `"jdbc/BankDB"` and with the jndi name `"java:/DefaultDS"`.
- `<ejb-ref>`: information on any other EJBs that this EJB needs to use. For example, the `AccountControllerEJB` EJB uses two other EJBs, as follows. The first EJB is named `"ejb/account"` and is registered with the JNDI directory under the name `"MyAccount"`. (Note: The `jndi-name` value in the `ejb-ref` field must match the `jndi-name` of the referred EJB.) The second used EJB is named `"ejb/customer"` and registered with the JNDI directory under the name `"MyCustomer"`.

In the `jboss.xml` file, each EJB can be configured to be managed by a customised container (code B.2). The main container configuration elements are described over the following paragraphs.

The `<container-configurations>` element is used to encompass the definitions of all customised containers.

The `<container-configuration>` element is used to describe a customised container configuration. It includes specifications for all plugins that the customised container is to use,

as well customised configurations for these plugins. As such, each container configuration can specify the container's default invoker type, interceptor makeup, instance cache and instance pool settings, persistence manager, or security. It is possible for a container configuration to be specified starting from an existing configuration. Specifically, a container configuration can have an `'extends'` attribute, indicating the name of the container being extended by the currently defined container. The example in code B.2 shows the main configurations elements used to customise the caching, polling and commit policies of EJB containers on JBoss. A similar EJB container configuration was used for experiments on Duke's Bank example (appendix C and section 5.2). These elements are briefly discussed over the following paragraphs.

The `<container-name>` element specifies the name of the customised container. The container name is used in the EJB components' configurations to indicate the customised container that is to manage the configured EJBs.

The `<instance-cache>` element indicates the fully-qualified class name of the `org.jboss.ejb.InstanceCache` interface implementation. Clearly, this element is only meaningful for cacheable EJB types (i.e. Entity and Stateful Session Beans). In the example in code B.2, the `org.jboss.ejb.plugins.InvalidableEntityInstanceCache` class was set for the cache instance. This setting allows deployers to provide customised implementations of the JBoss container caching functionality.

The `<container-cache-conf>` element indicates the caching policy and associated policy configurations to be used for the utilised caching implementation. In the example in Figure B.2, the `org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy` was used. The particular caching configuration for this policy is subsequently presented, as part of the `<cache-policy-conf>` element. Another possible cache policy is the `org.jboss.ejb.plugins.NoPassivationCachePolicy`, which never passivates instances. This second policy does not support the `<cache-policy-conf>` configuration element.

The `<cache-policy-conf>` element specifies the configuration parameters for the particular container cache used. Thus, each caching policy configuration will be specific to the particular caching implementation used. For example, the following cache parameters can be configured for the `LRUEnterpriseContextCachePolicy` caching policy. The `<min-capacity>` sub-element is used to indicate the minimum cache capacity. Similarly, the `<max-capacity>` element indicates the maximum cache capacity, which must be greater than or at least equal to the cache `min-capacity` value. The `<overager-period>` sub-element specifies the period (in seconds) between runs of the *overager* task. The purpose of the overager task is to see if the cache contains EJB instances with an age greater than the `max-bean-age` element value. Any beans meeting this criterion will be passivated. A cache's `max-bean-age` represents the maximum period (in seconds) that an inactive EJB instance is kept in the cache. After being inactive for more than the `max-bean-age` period, the EJB instance will be passivated by the overager process. The `<max-bean-age>` sub-element is used to configure this caching policy element. The `<resizer-period>` sub-element specifies the period (in seconds) between succeeding runs of the *resizer* task. The purpose of the resizer task is to contract or expand the cache capacity, based on various cache-miss related configurations. The `<remover-period>` sub-

element sets the period (in seconds) between subsequent runs of the *remover* task. The remover task removes passivated beans that have not been accessed in more than `max-bean-life` seconds. This task prevents Stateful Session Beans that were not removed by users from filling-up the passivation store. Finally, the `<max-bean-life>` sub-element sets the maximum inactivity period (in seconds) for an EJB instance before being removed from the passivation store.

A similar set of configuration elements are used to customise the EJB container instance pools, as follows.

The `<instance-pool>` element specifies the fully-qualified class name of an `org.jboss.ejb.InstancePool` interface implementation to use as the container `InstancePool`. This parameter was not specifically set in the example in code B.2. Consequently, the customised container used the default instance pool class, as inherited from the standard JBoss container. The standard instance pool implementation used was subsequently customised based on the specific configuration elements available. The `<container-pool-conf>` element specifies the configuration parameters for the container instance pool. The instance pool parameters that could be configured are briefly described next. The `<MinimumSize>` sub-element indicates the minimum number of EJB instances to be kept in the pool. (Note: the JBoss 3.2.5 version used did not initialise an `InstancePool` with the `MinimumSize` number of EJB instances, upon container instantiation). Similarly, the `<MaximumSize>` sub-element specifies the maximum number of EJB instances that are allowed in the pool. Normally, the `MaximumSize` represents the maximum number of EJB instances that are kept available. Nonetheless, additional instances will be created if the number of concurrent client requests exceeds the `MaximumSize` value. The `<strictMaximumSize>` sub-element must be used in order to limit the maximum concurrency of an EJB. Setting the `strictMaximumSize` element to true causes the instance pool to be strictly limited to the `MaximumSize` capacity value. In this case, only `MaximumSize` EJB instances may be active at anyone time. When the number of `MaximumSize` active instances has been reached, any subsequent requests are blocked until an instance is freed and returned in the instance pool. Finally, the `<strictTimeout>` pooling configuration sub-element indicates the time (in milliseconds) for which a request should block and wait for an EJB instance to become available in the pool. This parameter is meaningful when a `strictMaximumSize` pooling policy is used and the maximum number of EJB instances has already been reached in the pool. A `strictTimeout` value can be specified to be less than (or equal to) zero, in order to configure requests not to wait for EJB instances. In case a request times-out while waiting for an available EJB instance, an exception is thrown and the call is aborted.

Several commit options are available to specify the manner in which an EJB container will synchronise with the persistence storage used. In JBoss, this parameter can be configured via the `<commit-option>` element, with the possible commit values of `A`, `B`, `C` or `D`. The associated commit option policies are briefly described as follows (JBoss 3.2.5 - documentation). The commit option configuration used can have a significant impact on the application's performance as well as on its correctness and reliability characteristics. The set commit option configuration should be selected based on the predicted usage pattern of the targeted EJB component.

When `commit-option A` is used, the container caches the EJB instances' state between sub-

sequent transactions. This option assumes that the container is the only client accessing (or modifying) the persistent store. This assumption allows the container to synchronize the in-memory state from the persistent storage only when absolutely necessary. This occurs before the first business method executes on a found bean or after the bean is passivated and reactivated to serve another business method. This behaviour is independent of whether or not the business method executes inside a transactional context.

When commit-option `B` is used, the container caches the EJB instances' state between subsequent transactions. However, unlike option `A`, the container does not assume exclusive access to the persistent store. Therefore, the container will synchronize the in-memory (or cached) state at the beginning of each transaction.

When commit-option `C` is used, the container does not cache EJB instances. The in-memory state is synchronized on every transaction start.

The commit-option `D` is a JBoss-specific feature, not available in the standard EJB specification. It is a lazy-read scheme where the EJB instance state is cached between transactions (as with option `A`), but the state is periodically resynchronized with that of the persistent store. The default time between reloads is 30 seconds, but may be configured using the `<optiond-refresh-rate>` element.

B.2 JBoss Configurations for Database Persistence

The Hypersonic and MySQL databases were used for testing the Duke's Bank application. Hypersonic is an embedded database provided by JBoss. It was used for preliminary, functional testing, where the database could be collocated with the JBoss server. The reported experimental results (chapter 5) were obtained by testing Duke's Bank on a distributed deployment platform, where the JBoss server and the databases where running on different machines (section 5.2.5). The MySQL database was used in the thesis experimental tests to provide persistent storage for the Duke's Bank application. Special-purpose xml files must be used to configure JBoss to work with a certain targeted database (e.g. `hsqldb-ds.xml` for the Hypersonic DB and `mysql-ds.xml` for the MySQL DB). The xml database configuration file allows the specification of the database location and access configurations. The database parameters specified in this file are described below. The parameter values indicated were used to configure the MySQL DB for the performed experimental tests on the Duke's Bank (code B.3).

The `<jndi-name>` element is used to identify each particular database configuration (e.g. `DefaultDS`). This name must be subsequently used for referring to a certain targeted database.

The `<connection-url>` element indicates the url string for the connection JDBC driver. This parameter was set to point to the remote server used to run the MySQL database (i.e. `jdbc:mysql://10.10.105.146:3306/DukesBankDB`). This setting configures JBoss to use the MySQL database on a remote server, identified by the respective IP address, port and database name. This configuration is consistent with typical deployment scenarios,

in which the database and the application server are installed on different stations.

The `<min-pool-size>` element indicates the minimum number of connections that a database connection pool should hold. The minimum number of connections is lazily loaded upon the first call for a connection. A minimum of 50 connections was set for the connection pool. Similarly, the `<max-pool-size>` element sets the maximum number of connections the database connection pool holds. This is the maximum number of database connection that will be created in the pool. A maximum of 100 connections was set for the database connection pool. Finally, the `<blocking-timeout-millis>` element indicates the maximum time (in milliseconds) to wait for a database connection to become available. In case a connection does not become available after this set time, an exception is thrown. The actual time needed to create a database connection is not included in this period. The blocking timeout parameter was set to 1200,000 ms, or 20 minutes.

Listing B.3: `mysql-ds.xml` – datasource specification example

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <datasources>
4
5      <local-tx-datasource>
6          <jndi-name>DefaultDS</jndi-name>
7          <!-connect to DB on: server-ibm-113.pel.eeng.dcu.ie ->
8          <connection-url>jdbc:mysql://10.10.105.146:3306/DukesBankDB</connection-url>
9          <driver-class>com.mysql.jdbc.Driver</driver-class>
10         <transaction-isolation>
11             TRANSACTION_READ_UNCOMMITTED
12         </transaction-isolation>
13         <user-name>root</user-name>
14         <password></password>
15         <min-pool-size>50</min-pool-size>
16         <max-pool-size>100</max-pool-size>
17         <blocking-timeout-millis>1200000</blocking-timeout-millis> <!    20 min    >
18     </local-tx-datasource>
19
20 </datasources>
```

B.3 JBoss Web Server Configurations

The Tomcat Web server was used for deploying and running the web-related components of Duke's Bank application (i.e. jsp files and servlet components). The utilised JBoss server distribution provided an embedded version of the Tomcat web server. The bundled Tomcat server was configurable via the `server.xml` file. The following `http-connector` parameters were set for the Tomcat web server for the performed tests.

The `<acceptCount>` element indicates the maximum number of requests accepted in the waiting queue, when all possible request processing threads are already in use. Any processing requests received when the queue is full will be refused. The accept count parameter was set to 1000. The `<connectionTimeout>` element specifies the time (in milliseconds) that the connector will wait between accepting a connection and receiving the URI line that it needs to access. This parameter was set to 600,000 ms, or 10 minutes. The `<maxThreads>` element

sets the maximum number of processing threads that should be created and used. This parameter was set to 1500. The `http-connection` configuration used to customise Tomcat for the performed tests on the Duke's Bank is shown in code B.4.

Listing B.4: `server.xml` – HTTP connector specification example

```
<!-- A HTTP/1.1 Connector on port 8080 -->
<Connector port="8080" address="${jboss.bind.address}"
        maxThreads="1500" minSpareThreads="25" maxSpareThreads="75"
        enableLookups="false" redirectPort="8443" acceptCount="1000"
        connectionTimeout="600000" disableUploadTimeout="true"/>
```

In addition, the web session timeout can be set via the `web.xml` configuration file provided. The default value of 30 minutes was kept for this parameter for the performed tests (code B.5).

Listing B.5: `web.xml` – web session configuration example

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

B.4 JBoss Server Configurations

The JBoss server services, such as security, transactions, deployment, or logging, can be configured via the `jboss-service.xml` file, located in JBoss's configuration directory. This file was used to configure JBoss transactions for the performed experimental tests. More precisely, the `TransactionTimeout` attribute was used to set JBoss' transaction timeout to 900 seconds (15minutes) – code B.6.

Listing B.6: `jboss-service.xml` – JBoss transaction configuration example

```
<mbean code="org.jboss.tm.TransactionManagerService"
        name="jboss:service=TransactionManager"
        xmbean-dd="resource:xmdesc/TransactionManagerService-xmbean.xml">

    <attribute name="TransactionTimeout">900</attribute>
    <depends optional-attribute-name="XidFactory">
        jboss:service=XidFactory
    </depends>

</mbean>
```

The maximum amount of memory available to the JBoss server was configured at the JVM level using the `'-xms'` and `'-xmx'` java options. These options were used when starting the JBoss application server, in order to set the available memory (e.g. modified the `run.bat` file for starting JBoss on a Windows OS platform: `set JAVA_OPTS=%JAVA_OPTS% -Xms100m -Xmx512m`).

This approach was used for some of the performed tests (section 5), in order to simulate an execution environment with limited memory resources.

B.5 Extending JBoss

JBoss implementation was modified and new libraries were added and utilised. JBoss was subsequenlty configured to use these libraries when compiling its modified code, as well as find the libraries during runtime. These procedures were necessary as the JBoss distribution used (JBoss 3.2.5) does not support the direct addition of new libraries to its `CLASSPATH` (if the JBoss classpath was modified, exceptions were raised during JBoss start-up, preventing the server from starting).

An initial configuration was required to allow modified JBoss classes to be correctly compiled. The modification was needed for several JBoss classes which used implementations available from external libraries. For example, several JBoss classes were instrumented so as to send runtime monitoring events to the AQuA framework (e.g. JBoss `LogInterceptor` instances communicate with AQuA's `RGManager` instances). The `<JBoss_HOME>/server/build.xml` file was modified in order to allow the required libraries to be found at JBoss compilation time (e.g. the `APeMCA.jar` library, containing all AQuA's constituent classes). Code B.7 shows the way additional pathelements were appended to the JBoss server library path (in the server's `build.xml` file), so as to indicate the location of the additional external libraries. Modifications can be made to any of JBoss' constituent packages, similarly to the exemplified modifications made on the server module. In addition, JBoss can be generally configured to recognise external libraries by adding the corresponding pathelements into its global `build.xml` file (i.e. `<JBoss_HOME>/build/build.xml`).

Listing B.7: `build.xml` – finding external libraries at JBoss compile time

```
1 <!--additional entries: new libraries needed to compile the extended JBoss server -->
2 <pathelement path="C:/Libraries/PELJBoss/APeMCA.jar"/>
3 <pathelement path="C:/Libraries/PELJBoss/jboss_adaptor.jar"/>
4 <pathelement path="C:/Libraries/PELJBoss/jmx-basic.jar"/>
```

JBoss must subsequently be configured so as to find the required external libraries during runtime. This was achieved by adding the external libraries (e.g. `APeMCA.jar`) to the JBoss's output library directory (e.g. `<JBOSS_HOME>/build/output/jboss-3.2.5/server/default/lib/APeMCA.jar`). In addition to external libraries JBoss must also be able to find any property files used by its classes at runtime. For this purpose, the property files were added to JBoss' `working` directory, or `bin` directory. For example, for classes in the `APeMCA.jar` library, the `apemcaProperties.properties` file was added to JBoss' `bin` directory (i.e. `<JBOSS_HOME>/build/output/jboss-3.2.5/bin/apemcaProperties.properties`).

APPENDIX

C

Duke's Bank J2EE Application

C.1 Presentation

Duke's Bank application is a sample J2EE application provided by Sun Microsystems. It represents an online banking application that allows customers to perform online banking operations, such as accessing account histories and performing bank transactions. Administrators can also use Duke's Bank application to manage customer records and accounts. Information on customers, accounts and banking transactions is stored in a database (DB) and accessed via Entity EJBs. Client sessions are managed via Stateful Session EJBs. The EJB components in the application layer are accessed via web components in the presentation layer. Web components include jsp files and servlets. Figure C.1 provides a high-level overview of the J2EE components contained by the Duke's Bank application and shows the way these components interact with external clients and with the database (DB).

C.1.1 Enterprise Java Beans (EJBs) in the Duke's Bank

The Duke Bank's application comprises several Enterprise Java Beans (EJBs), as depicted in Figure C.2. Three Stateful Session beans are used to handle user requests and maintain client sessions. These are the `AccountController`, `CustomerController` and `TxController` EJBs. Three Entity beans are used to handle persistent DB data at the application level. Each Entity bean represents one of the business entities in the Duke's Bank. Namely, the `Customer` Entity bean represents banks customers, the `Account` Entity bean represents banking accounts and the `Tx` Entity bean represents banking transactions. External access to the Entity beans is mediated by the Stateful Session beans. The state of the Entity beans is persisted into a relational database (DB), in the corresponding `customer`, `account` and `tx` DB tables, respectively.

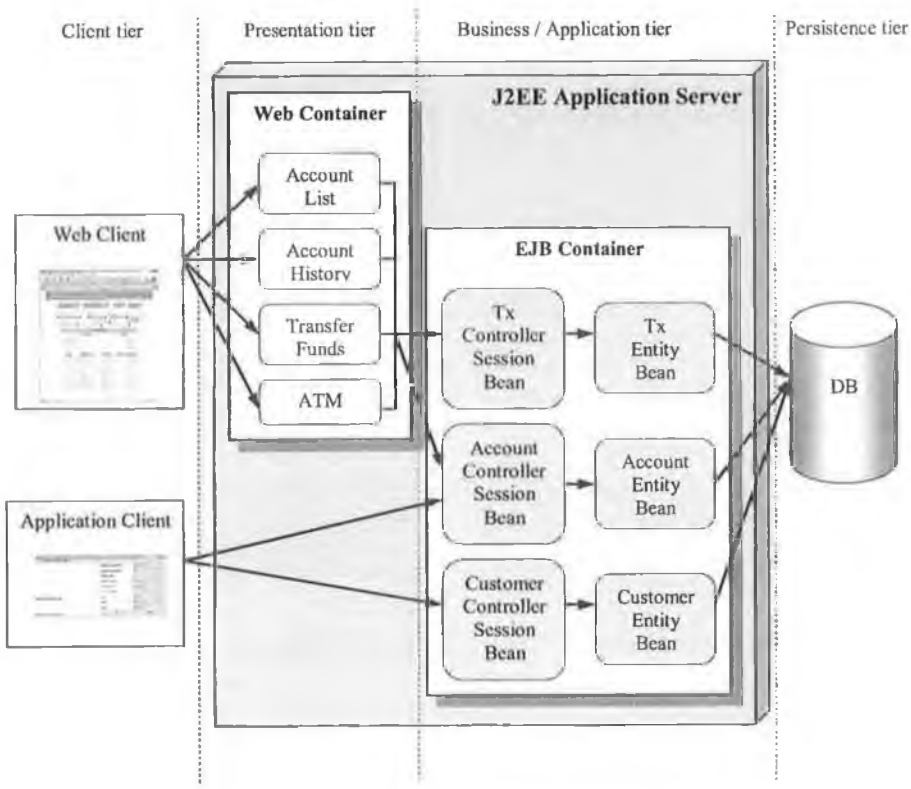

Figure C.1: Duke's Bank application – high-level architecture

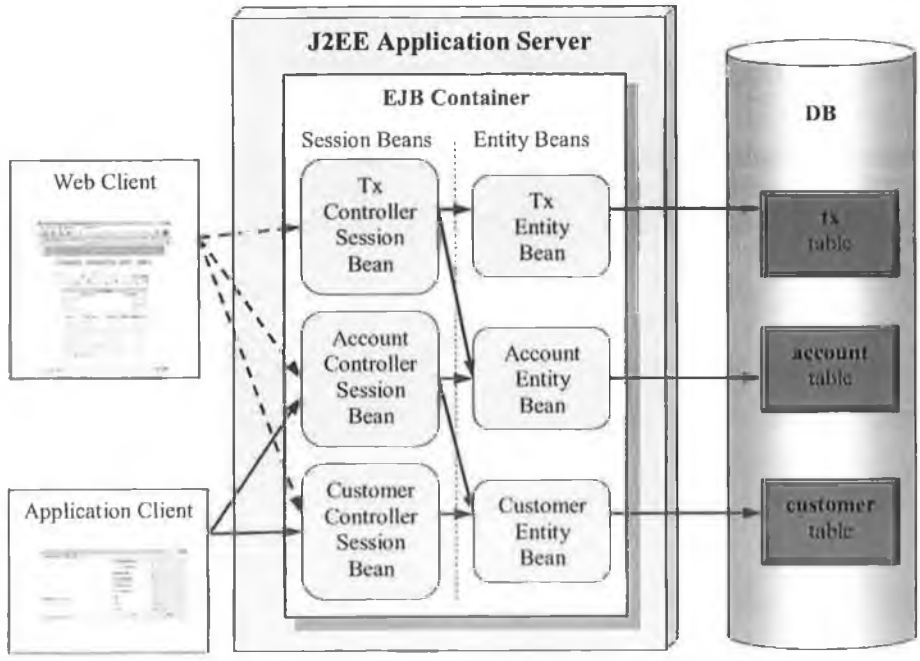

Figure C.2: Duke's Bank – EJBs overview

C.1.2 Relational Database - Persistence Support for the Duke's Bank

A relational database (DB) was used to provide persistence support for the Duke's Bank application (i.e. the MySQL DB and the Hypersonic DB embedded in JBoss). The database tables were designed so as to reflect the main business entities in the application – bank accounts, customers, and banking transactions. These business entities are mapped to tables in the relational database as follows (Figure C.3). An `account` table holds information on banking accounts, a `customer` table holds information on the bank's customers and a `tx` table holds information on performed banking transactions. These tables are related so as to represent the existing connections between the real business entities they represent. Thus, a many-to-one relation exists between the transactions table (`tx`) and the accounts table (`account`). This relation implies that many transactions can be associated with each account, but each transaction can only belong to a single account. In addition, a many-to-many relation exists between the customers table and the accounts table. This relation implies that each customer can own many accounts and each account may belong to multiple customers. This relation is implemented by an additional table (i.e. the `Customer_Account_XRef table`), which holds records of all customer-to-account mappings.

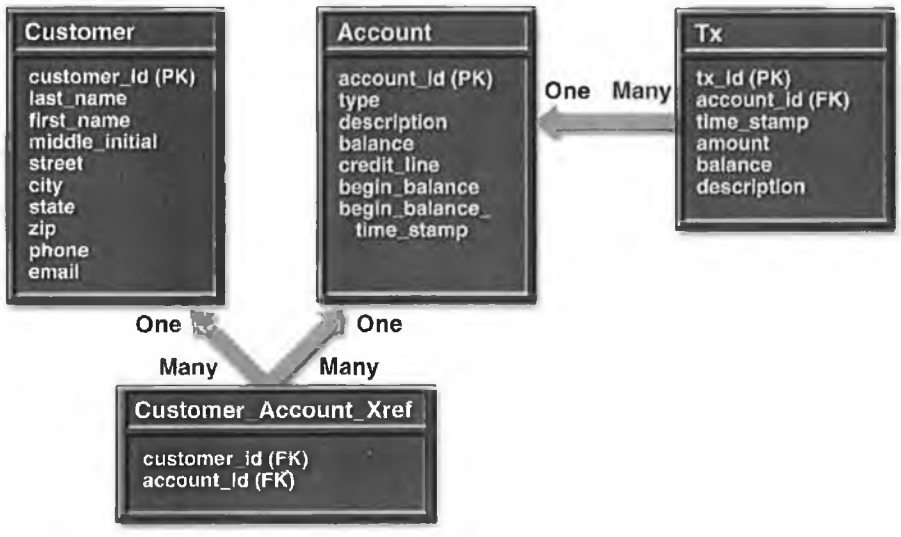

Figure C.3: Duke's Bank persistence – DB tables overview

C.2 Identified Exceptions and Troubleshooting

The original distribution of the Duke's Bank application contains several bugs, which prevent the application from running correctly or performing properly. This appendix describes the functional bugs detected in the Duke's Bank application. It also shows the way the original Duke's Bank distribution was modified so as to correct the detected faults. Several modifications made for improving the Duke's Bank application performance are also discussed.

C.2.1 Concurrent users not supported

Observed Symptom

Exception thrown when testing Duke's Bank with multiple concurrent users:

```
ERROR [org.jboss.ejb.plugins.LogInterceptor] EJBException:
javax.ejb.EJBException: Application Error: tried to enter
Stateful bean with different tx context
```

Detected Problem

The original Duke's Bank application does not support simultaneous accesses from multiple users. In other words, the application can only be accessed by one user at a time. This is a serious bug, as it prevents the application from being load-tested. It would be an unacceptable limitation for a real online banking application.

Identified Cause

The reason the original Duke's Bank was not supporting concurrent users was identified as follows. The problem was caused by the fact that all jsp files use the same single instance of the `BeanManager` JavaBean, for all client requests. This is done in the original distribution by setting the scope of the `BeanManager` JavaBean to the `'application'` value, in all jsp files. This means that one single instance of the `BeanManager` class will be used for the entire application. At construction time, the `BeanManager` creates and obtains references to instances of the Duke's Bank `AccountController`, `CustomerController` and `TxController` Stateful Session Beans. The `BeanManager` instance uses these session bean instances during its entire lifecycle. Thus, all incoming calls, from all client sessions, will be consequently handled using the same Stateful Session Bean instances. Instead, for the application to function correctly, each customer session should be handled using a separate Stateful Session Bean. Concurrent access to Stateful Session Beans is not supported (as tested on JBoss). Consequently, an exception is usually thrown when more than one simultaneous users access the application: `"Application Error: tried to enter Stateful bean with different tx context"`.

Implemented Solution

The solution to the aforementioned problem is to ensure that Stateful Session Beans are not concurrently accessed by multiple clients. For this purpose, the scope attribute value for the `BeanManager` JavaBean was modified from `'application'` to `'session'`. This modification was made in all jsp files using the `BeanManager` JavaBean. As a consequence of the new setting, a separate `BeanManager` instance will be assigned to each different user session, rather than per application. Thus, every distinctive user will be assigned a separate `BeanManager` instance and will in effect use a separate set of Stateful Session Bean instances.

This solution was implemented as follows. The scope of the `BeanManager` class was

changed from 'application' (as in the original distribution) to 'session', by modifying all relevant jsp files. First, the session="true" statement was appended to the end of the <%@page import="..." %> declaration. In the <jsp:useBean id="beanManager" class="...BeanManager" scope="session"/> the scope was set to 'session' and not to 'application' (as in the original). The following jsp files were affected by this change: accountHist.jsp, accountList.jsp, atm.jsp, atmAck.jsp, links.jsp, transferFunds.jsp and transferAck.jsp (Note: in this jsp file the <%@ page ...%> clause is present twice, so the clause session="true" was accordingly inserted twice).

In the provided solution, the scope of the BeanManager JavaBean was changed from 'application' to 'session'. Additional modifications had to be consequently made in order to accommodate this modification. First, the instantiation of the BeanManager JavaBean had to be performed whenever a new http session was being created. Second, the BeanManager instance associated with a http session has to be destroyed whenever the session terminates and is invalidated. This is different from the original Duke's Bank version, in which the BeanManager was only instantiated once for the entire application and destroyed when the application terminated. These additional modifications were performed by implementing an additional class, the SessionListener. The SessionListener's role is to listen to session creation and destruction events and accordingly performs the BeanManager instantiation and deletion operations (code C.1 below). For this purpose, the SessionListener class implements the HttpSessionListener interface, which allows it to listen to session-related events (e.g. session created, or session destroyed events), obtain session references (e.g. event.getSession()) and set session attributes (e.g. session.setAttribute("beanManager", new BeanManager())). In the original Duke's Bank distribution, the instantiation and removal of the BeanManager JavaBean were performed only once, at application start-up and shutdown, respectively. These functionalities were performed by the ContextListener class, which implemented the ServletContextListener interface.

A final modification, the removal of the BeanManager instance when an http session destroyed event occurs was moved from the SessionListener class to the logoff.jsp file (code C.2 below). The reason was that in order to know which BeanManager instance to destroy upon receiving a session destruction event, the actual session with which the BeanManager instance was associated was needed. Nonetheless, this http session did no longer exist as its termination was the one actually triggering the session destruction event. Thus, the session was already inaccessible at the point when the destruction event was received. The BeanManager instance's removal was consequently moved to the logoff.jsp file. This solution is suitable for the Duke's Bank application, since the moment a user logs off, their client session is terminated and the associated BeanManager can safely be removed.

Listing C.1: The SessionListener class

```
1 package com.sun.ebank.web;
2 //...
3 public class SessionListener
4 implements HttpSessionListener {
5
```

```
6       HttpSession session = null;
7
8   //react to session creation events
9   //create the BeanManager instance and
10  //set it as an attribute with the new http session
11      public void sessionCreated( HttpSessionEvent event ) {
12          this.session = event.getSession();
13          session.setAttribute( "beanManager", new BeanManager() );
14      }
15
16  //react to session destruction events
17      public void sessionDestroyed(HttpSessionEvent event) {
18  //logic in this method was moved to the logoff.jsp file
19      }
20
21  }
```

C.2.2 Logging-off functionality broken

Observed Symptom

The original Duke's Bank distribution throws an exception when users access the *log-off* functionality. The following error is returned to the user when attempting to log-off Duke's Bank:

```
Server Error
Your request cannot be completed.  The server got the
following error:  null
```

Identified Cause

The cause of this erroneous behaviour was identified in the `logoff.jsp` file. The log-off code tries to access a client http session after already having invalidated this session. As the user does not need their client session after logging off, the session associated with their logoff request can be safely invalidated. Nonetheless, no attempts to access this session should be made after that.

Implemented Solution

In the `logoff.jsp` file, the line that tries to access an http session that has already been removed was commented (i.e. `<%-- HttpSession newSession = request.getSession(true); --%>`).

C.2.3 Incomplete EJB passivation and activation

Detected Problem

When an EJB instance is to be passivated, all the non-serialisable instance variables must be released before the actual passivation takes place. Examples of non-serialisable resources include DB connections and references to other EJBs. When the EJB instance is activated these variables need to be reallocated. An EJB's `EJBpassivate` and `EJBactivate` methods are used for this purpose. These operations were not properly performed in the original Duke's Bank distribution, potentially causing memory leaks, or general inefficient use of resources.

Implemented Solution for Passivating and Activating Stateful Session EJBs

The original Duke's Bank distribution was modified so as to correctly manage non-serialisable variables during passivation and activation operations. The `EJBpassivate` method was updated in the following Stateful Session EJBs in order to release non-serialisable EJB variables. In the `AccountControllerBean` EJB: the `AccountHome accountHome`, `Account account` and `Connection con` variables were released. In the `CustomerControllerBean` EJB: the `CustomerHome customerHome`, `Customer customer` and `Connection con` variables were released. Finally, in the `TxControllerBean` EJB: the `TxHome txHome`, `AccountHome accountHome` and `Connection con` variables were released.

Implemented Solution for Passivating and Activating Entity EJBs

The following variables needed to be released in the `EJBpassivate` method of the Duke's Bank Entity EJBs, as follows. The `Connection con` variable was released in the `AccountBean`, `CustomerBean` and `TXBean` EJBs.

C.2.4 Database Connections Not Properly Released

Detected Problem

Duke's Bank application uses Entity beans with Bean Managed Persistence (BMP). This means that the logic required for persisting the entities' state in the database (DB) must be provided in the beans' code. EJB persistence logic involves connecting to the DB, retrieving, updating or storing data in the DB and disconnecting from the DB. When disconnecting from the DB, the original Duke's Bank code was releasing DB connections correctly but did not release the connection variables pointing to these connections. This caused exceptions to be raised when the server/container was trying to passivate the EJB instances holding these variables.

Implemented Solution

The EJB variables holding DB connections were set to `null`, in the `releaseConnection()` method of the following classes: `AccountBean.java`, `TxBean.java`, `CustomerBean.java`, `AccountControllerBean.java`, `TxControllerBean.java`, `CustomerControllerBean.java`, as follows. The `releaseConnection()` method was modified by adding the following statement: `con = null;`. The statement was added somewhere after the statement closing the database conection `con.close();`.

C.2.5 Stateful Session Bean Instances Not Released at the End of HTTP Sessions

Detected Problem

In the original Duke's Bank distribution, the `BeanManager` class's constructor creates instances of several Stateful Session EJBs: `AccountController`, `CustomerController` and `TxController`. These EJB instances should be removed when the http session associated with the `BeanManager` class terminates. For this purpose, the `BeanManager` class provides a `destroy` method, but this method is never actually called. The `destroy` method of a `BeanManager` instance should be called before the invalidation of the http session using that instance. The `destroy` method should be implemented so as to release the Stateful Session bean instances associated with the `BeanManager` instance. As each `BeanManager` instance is responsible for managing one user session, the EJB instances used during that session should be released at the end of the session. However, in the original Duke's Bank, the `destroy` method was never actually called. In addition, the `destroy` method was merely setting the variables pointing to the EJB instances to `null`, without actually releasing the instances first. Consequently, the Stateful Session bean instances were not released. Thus these instances were unnecessarily kept in the container cache until passivated or/and until they expired and were removed by the EJB container.

Implemented Solution

In order to release Stateful Session EJB instances, the `BeanManager`'s `destroy()` method was modified as follows. Statements for releasing the Stateful Session bean instances were added: `custctl.remove(); acctctl.remove(); txctl.remove();`. These additions were placed before the statements that were setting the `acctctl`, `custctl` and `txctl` variables to `null`. In addition, The `logoff.jsp` file was modified so as to call the `BeanManager`'s `destroy()` method upon the http session's termination (code C.2).

Listing C.2: Removing the `BeanManager` from terminated HTTP sessions

```
1 BeanManager beanManager = (BeanManager)(session.getAttribute("beanManager");
2 beanManager.destroy();
3 session.removeAttribute("beanManager");
4 request.getSession(false).invalidate();
```

C.2.6 Result Sets Not Closed After No Longer Used

Observed Symptom

JBoss raises the following runtime WARNING:

```
WARN  [WrappedConnection] Closing a result set you left open!
Please close it yourself.
java.lang.Exception: STACKTRACE
        at org.jboss....WrappedStatement.registerResultSet(...)
        at org.jboss....WrappedPreparedStatement.executeQuery(...)
        at com.sun.ebank.ejb.account.AccountBean.loadAccount(...)
        at com.sun.ebank.ejb.account.AccountBean.ejbLoad(...)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(...)
        at sun.reflect.NativeMethodAccessorImpl.invoke(...)
```

Detected Problem

In Duke's Bank, Result Sets are used to implement the Entity EJBs' Bean Managed Persistence (BMP) functionality. The Result Sets are used to store collections of data, obtained in response to SQL requests to the DB. Result Sets can be browsed iteratively for accessing individual data items and should be closed after no longer used, to safe system resources. Though, in the original Duke's Bank distribution several Result Sets were not being closed even after no-longer used, potentially impacting on the system's performance.

Implemented Solution

The Duke's Bank methods that were using Result Sets were modified so as to close the opened Result Sets after no longer using them. The following code was added for this purpose: `rs.close(); rs = null;` (where `rs` is the Result Set variable). The addition was inserted before the prepared statement for making SQL requests to the DB was closed (i.e. `prepStmt.close();`). This modification was made for all EJB methods that used Result Sets, as follows. In the `com.sun.ebank.ejb.account.AccountBean` Entity bean, the `loadCustomerIds`, `loadAccount`, `selectByCustomerId(String customerId)` and `selectByPrimaryKey(AccountBean)` methods were modified. In the `com.sun.ebank.ejb.tx.TxBean` Entity bean, the `selectByAccountId(TxBean)`, `loadTx(TxBean)` and `selectByPrimaryKey(String primaryKey)` methods were modified to close Result Sets.

APPENDIX

D

Configuring OpenSTA to Simulate Client Load

The Open System Testing Architecture (OpenSTA) is distributed software architecture for performance testing. The current OpenSTA toolset can perform scripted http and https tests and execute performance measurements on Windows platforms. OpenSTA provides support for a 'record and replay' metaphor that allows specifying and running heavy-load testing scripts. Specifically, testers can record client sessions using their own browsers. Recorded sessions are automatically transformed into simple testing scripts, which can be further edited and controlled via a high-level scripting language. A high-performance load-generation engine can be subsequently configured to play-back scripted sessions and simulate multiple users. This methodology can be used to generate realistically heavy loads, simulating the activity of thousands of virtual users.

Results and statistics can be collected and graphically displayed during the performed tests. Various data collection mechanisms can be selected, configured or specified by the user, including scripted timers, SNMP data, Windows Performance Monitor stats and http results and timings. Logged results can be graphically viewed during runtime and subsequently viewed, graphed, or exported to more sophisticated tools for further processing (e.g. Microsoft Office Excel).

The OpenSTA toolset was used in the thesis experimental work to simulate user load on the Duke's Bank J2EE application (section 5.2.7). This section describes the most important configurations and scripting implementations performed on OpenSTA for this purpose.

D.1 Test Executer Initialization

Several parameters can be set to customise OpenSTA's HTTP Test Executer's[1] operation at runtime. This can be achieved via the `TestExecuter_web.ini` initialisation file, located in OpenSTA's installation Engines directory (i.e. `<OpenSTA_HOME>/Engines/TestExecuter_web.ini`).

[1]OpenSTA Documentation – HTTP Test Executer Initialization File; `http://www.opensta.org/docs/ug/os-appen.htm`

Code D.1 shows the configuration settings used to load test the Duke's Bank.

Listing D.1: Test Executer initialization file configurations

```
1  [FILES]
2  TraceLevel=0
3
4  [SOCKET]
5  MaxSocketDataBuffersCount=256000
6  SocketDataBuffersGrowingCount=200
7  MaxSSLConcurrentReq=8000
8  SSLGrowingBuffCount=100
9  Timeout=720000
10 ReuseAddr=0
11 Linger=0
12
13 [TEST]
14 BrowserParallelism=4
15 InitialVirtualUserCount=1000
16 VirtualUserGrowBy=10
17
18 [THREAD POOL]
19 ThreadPoolSize=4
20 ThreadPoolConcurrentThreads=100
```

The configuration parameters that were set to differ from their initial default values are described next.

The HTTP playback request timeout was configured to change the default value of 60,000 ms (1 minute) to an increased 720,000 ms (12 minutes) value. The reason was that the default 1 minute timeout value was too small for the high-load limited-resources testing conditions. Increasing this parameter's value eliminated timeout errors that previously occurred. This configuration was achieved by adding a `Timeout` parameter to the `[SOCKET]` section of the `TestExecuter_web.ini` file. The `Linger=0` and `ReuseAddr=0` settings were also added to the `[SOCKET]` configuration section to prevent `'socket address already in use'` errors. Such errors occurred when the OpenSTA load injector could no longer allocate new sockets for making requests. This is due to the fact that Windows limits the maximum number of open sockets (i.e. 5000 maximum user ports minus the first 1024 reserved ports).

D.2 Scripting Configurations

A client usage scenario for the Duke's Bank was recorded and stored as an OpenSTA script. The scenario involved several operations, including the client logging in, listing all their bank accounts, then listing all banking transactions from a selected account and finally logging out of the bank. The recorded script was subsequently modified so as to utilise variable values for the users' identities and accessed banking accounts. This was necessary when simulating multiple concurrent users, in order to avoid the unlikely situation in which a single user logs in and manages the same bank accounts in parallel, at the same time.

A `USERNAME` variable was defined to represent a user's identity as part of each simulated client session (code D.2). The username variable was set to take values corresponding to valid

user ids in the Duke's Bank customer database table (i.e. user id values between 300 and 1299). The scope of the username variable was set to SCRIPT, meaning that the variable was shared between all concurrent script instances simulating parallel running clients. Similarly, another variable was defined to provide different account id values for the managed banking accounts. The ACCOUNT_ID variable was set to take valid account id values corresponding to the Duke's Bank account database table (i.e. account id values between 0 and 999). The username and account id variables were set to cyclically take one of the possible valid values defined (i.e. NEXT USERNAME, NEXT ACCOUNT_ID). A MUTEX was used to synchronise concurrently simulated clients and avoid multiple script instances from simultaneously acquiring the same username and account id values. As such, each simulated client session acquires the mutex token before allocating variable values and releases it afterwards (Figure D.3). A third variable was defined to allow random waiting times between sequential user and account id value allocations. The purpose of such random delays was to avoid having multiple simulated client scripts allocating identical user and/or account ids at the same time.

Finally, the defined variables were used to replace hard-coded script values that were initially set during the client session recording procedure. Figure D.4 shows how the username and account id variables are incorporated into the recorded http requests to the Duke's Bank.

Listing D.2: Defining scripting variables

```
1  CHARACTER*512 MY_USERNAME, LOCAL
2  CHARACTER*512 USERNAME ( "300", "301", "302" &
3          , "303", "304", "305" &
4          , "306", "307" ... "1298", "1299" ), SCRIPT
5
6  INTEGER       ONE_RANDOM_WAIT, LOCAL
7  INTEGER       RANDOM_WAIT ( 0, 1, 2 &
8          , 3, 4, 5 &
9          ...
10         , 196, 197 &
11         , 198, 199, 200 ), SCRIPT
12
13 CHARACTER*512 MY_ACCOUNT_ID, LOCAL
14 CHARACTER*512 ACCOUNT_ID ( "0", "1", "2" &
15         , "3", "4", "5" &
16         , "6", "7", "8" &
17         , "9", "10", "11" &
18         , "12", "13", "14" &
19         ...
20         "999" ), SCRIPT
```

Listing D.3: Synchronously allocating sequential variable values

```
1  ACQUIRE MUTEX "LOGIN"
2  NEXT RANDOM_WAIT
3  SET ONE_RANDOM_WAIT = RANDOM_WAIT
4  WAIT ONE_RANDOM_WAIT
5  NEXT USERNAME
6  SET MY_USERNAME = USERNAME
7  NEXT ACCOUNT_ID
8  SET MY_ACCOUNT_ID = ACCOUNT_ID
9  RELEASE MUTEX "LOGIN"
10
11 LOG "MY_USERNAME: ", MY_USERNAME
```

```
12 LOG "MY_ACCOUNT_ID: ", MY_ACCOUNT_ID
```

Listing D.4: Inserting and using variables into http client calls

```
1  PRIMARY POST URI "http://localhost:8080/bank/j_security_check HTTP/1.0" ON 2 &
2  HEADER DEFAULT_HEADERS &
3  ,WITH {"Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*", &
4      "Referer: http://localhost:8080/bank/logon;jsessionid=1C2C69E348B8" &
5      "342EB1C28AD450C9D41C", &
6      "Accept-Language: en-ie", &
7      "Content-Type: application/x-www-form-urlencoded", &
8      "Connection: Keep-Alive", &
9      "Content-Length: 30", &
10     "Pragma: no-cache", &
11     "Cookie: "+cookie_1_0} &
12 ,BODY "j_username="+MY_USERNAME+"&j_password=j2ee"
13
14
15 PRIMARY GET URI "http://localhost:8080/bank/accountHist?accountId="+MY_ACCOUNT_ID+"&
       date=0&year=20" &
16     "04&sinceMonth=8&sinceDay=1&beginMonth=8&beginDay=1" &
17     "endMonth=8&endDay=1&activityV" &
18     "iew=0&sortOption=0 HTTP/1.0" ON 4 &
19 HEADER DEFAULT_HEADERS &
20 ,WITH {"Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*", &
21     "Referer: http://localhost:8080/bank/accountList", &
22     "Accept-Language: en-ie", &
23     "Connection: Keep-Alive", &
24     "Cookie: "+cookie_1_0}
```