Automating the Performance Management of Component-Based Enterprise Systems through the use of Redundancy *

Ada Diaconescu†
Performance Engineering Laboratory
Dublin City University
Dublin, Ireland
diacones@eeng.dcu.ie

John Murphy
Performance Engineering Laboratory
University College Dublin
Dublin, Ireland
j.murphy@ucd.ie

ABSTRACT

Component technologies are increasingly being used for building enterprise systems, as they can 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.
This paper presents a framework for automating the performance management of complex, component-based systems. The adopted approach is based on the alternate usage of multiple component variants with equivalent functional characteristics, each one optimized for a different running environment. A fully-automated framework prototype for J2EE is presented, along with results from managing a sample enterprise application on JBoss. A mechanism that uses monitoring data to learn and automatically improve the framework's management behaviour is proposed. The framework imposes no extra requirements on component providers, or on the component technologies.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software configuration management, Software quality assurance (SQA)*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*

General Terms

Management, Measurement, Performance

*The presented work is funded by Enterprise Ireland Informatics Research Initiative 2001
†An important part of the presented implementation and testing work was carried out while working in the SARDES Project, at INRIA Rhone-Alpes, INPG UJF, France

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASE'05, November 7–11, 2005, Long Beach, California, USA.
Copyright 2005 ACM 1-58113-993-4/05/0011 ...$5.00.

Keywords

autonomic management, J2EE, decision policies

1. INTRODUCTION

Component technologies [11], such as J2EE[1] and .NET are increasingly being used for building complex enterprise applications. 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. 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 task. This paper presents a framework for automatically managing the performance and availabilty of complex, component-based software systems. The presented framework is referred to as *AQuA (**A**utomatic **Qu**ality **A**ssurance)*. AQuA's goal is to enable applications to fluidly mould to their constantly changing execution environments. The adopted approach is based on the alternate usage of multiple component variants with equivalent functional characteristics, each one optimized for a different running environment. In short, the management framework uses runtime monitoring data to detect changes in the execution environment and application performance; it then automatically adapts the application so as to optimise it for the current running environment. Mon-

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

itoring data is also used to enable the framework to learn and improve its management behaviour over time, without requiring any human intervention. The learning process is used to automatically build accurate specifications of the components' performance characteristics, in the current deployment context. Based on this information, the framework can successfully decide how to adapt the application in different running conditions. Decision policies used to specify and configure the framework's management behaviour are clearly separated from the actual implementation of supporting framework operations. This allows system managers to state performance goals and to configure the management process for their systems without requiring a thorough understanding of the underlying framework mechanisms and without the need to modify the framework implementation. A proof-of-concept framework implementation is presented for the J2EE component technology, along with test results from managing a sample enterprise application, the Duke's Bank[2], on JBoss[3]. The implemented prototype is referred to as *AQuA_J2EE*, currently targeting the automatic management of performance and availability in J2EE systems. Further management capabilities, for quality attributes such as reliability and dependability, will be subsequenlty added. AQuA_J2EE can be employed for managing any J2EE application on JBoss, at the EJB level, without requiring any design or implementation changes; clearly, the decision policies need to be configured according to the particular quality goals of each managed application. In addition, the general framework design allows it to be employed for managing J2EE applications at different granularity levels, or for managing component technologies other than J2EE, without requiring major, conceptual-level modifications.

2. FRAMEWORK OVERVIEW

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 [13], [3] that there are frequently no unique component implementations or configurations that can yield optimal performance under all possible execution environments. Based on this consideration, the presented research proposes employing multiple component variants, providing equivalent functionalities, but each one being optimised for a different execution environment. Such component variants 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 [2], [3] and [4]. The principal idea behind this approach is to have multiple redundant components prepared at runtime and only use the one that is optimal under the current execution environment. At any time, for providing certain functionality the system will select and use a single redundant component from the RG providing that functionality. The selected redundant component will be the one that is most likely to yield optimum performance under the current execution environment. If the execution environment changes, the framework selects another redundant component 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. It could be argued that an alternative approach would be to specify all behaviours for all possible 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. Hence, adaptation logic and redundant components can be independently added, modified, or removed from the running system, as needed.

In the targeted component technologies [11] (e.g. Enterprise JavaBeans (EJB)), 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. The tests presented in this paper were performed using redundant components with variations at the component configuration level. An example of redundant components with variations at the implementation level was presented in [3]. The quality metrics currently considered for evaluating AQuA's benefits are performance and availability. Performance metrics include response times and throughput, but also CPU, bandwidth and memory usage. Metrics considered for a component's execution environment include workload and the software and hardware resources available to that component.

The main management functionalities that the AQuA framework provides consist of *system monitoring*, *learning*, *anomaly detection*, *component evaluation*, *adaptation decision* and *component activation*. A framework prototype, AQuA_J2EE, was implemented in order to test the way the presented functionalities work together for managing the performance of J2EE applications. These functionalities are briefly introduced over the following paragraphs. The learning mechanism is presented in more detail next. A thorough description of these functions and of the way they operate is available from [3], [2], or [4].

AQuA's *monitoring* functionality is responsible for collecting runtime data from the managed components and their execution environment. This data is used for detecting performance anomalies and important variations in the components' execution environment. In addition, as part of the framework'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.

AQuA's *anomaly detection* functionality is responsible for identifying and signalling the occurrence of performance problems, or of relevant variations in the execution environment. Availability concerns are also raised in case exceptions are being caught by the monitoring module (e.g. out of memory Exception). Performance anomalies are generally sig-

[2]Sun Microsystems - Duke's Bank sample J2EE application: http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Ebank.html
[3]JBoss J2EE application server: www.jboss.org

nalled 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 [2]. 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; workload and resource availability variations are considered for this purpose. The principal idea behind this strategy is that if a system is presently meeting its functional and performance-related requirements, it will generally continue to do so unless a change intervenes to alter this state. In the presented test case, AQuA detects and uses changes in the incoming workload to trigger the application adaptation. As a possible further improvement, the environmental variations themselves can sometimes also be predicted. In such cases, applications can be pre-emptively adapted to deal with the predicted changes before they actually happened. For example, certain applications may experience workload variations with the time of day, week, month or year (e.g. banking applications may expect reduced user loads during non-working hours; e-commerce applications would expect increased loads before certain events or during sales). When such periods can be predicted, system managers can instruct AQuA to automatically activate a different redundant component during each interval.

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. Initial performance information can be provided at component deployment time, from test results and previous experiences with the considered components [3], [2] and [4]. As part of the framework's learning process, this information is dynamically validated and constantly updated, based on accurate monitoring data obtained from the targeted managed system. Additionally, in case initial descriptions are not provided, the learning mechanism is used to obtain this information from scratch. Component evaluation results have a confidence level associated with them, depending on the reliability of the information used in the evaluation process. 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. In addition, 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.

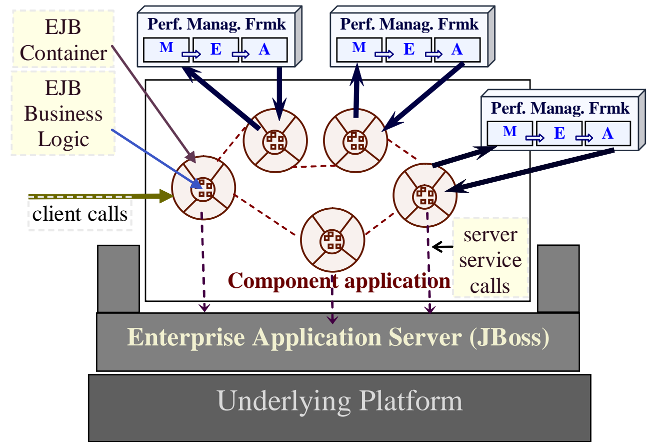

Figure 1: Component-level management ***M - monitoring and detection, E - evaluation and decision, A - component activation***

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, as indicated by the evaluation module. The costs of performing the actual adaptation operations over the potential benefits are considered at this level. The current AQuA_J2EE framework implementation uses detection, evaluation and decision functionalities with local, component-level scopes. This means that each individual RG is being optimised separately from other RGs. Figure 1 shows an overall view of the way the presented management framework is integrated with an EJB server. As indicated in the figure, multiple framework instances are created - one for each managed component. In order to reduce overheads associated with monitoring and analysing *all* components, it is possible to specify a selected *subset* of components to be managed; framework instances will then only be created to manage the selected components. Consequently, this approach can considerably reduce the framework's footprint and impact on system performance. 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 [2]. Global evaluation and decision processes will be subsequently introduced to control local management processes from a higher level, as presented in [4]; these capabilities are important for avoiding situations in which local optimisations lead to an overall degradation in application performance, or cases in which a certain set of decisions are being taken in a cyclical manner.

AQuA's *component activation* functionality is used for dynamically swapping components, while maintaining system consistency and the validity of existing user references [2].

2.1 The Learning Mechanism

AQuA's *learning* functionality enables it to: i) avoid requiring initial performance descriptions [3], [2] to be supplied at components' deployment time; ii) avoid completely relying on monitoring results obtained when components were integrated in a different system (e.g. a testing platform). As such, the *learning* mechanism was devised to automatically acquire performance information on managed components, in the current system. The adopted strategy was implemented and partially tested. AQuA_J2EE is to be subsequently updated so as to avail of this capability and thus complement, or replace certain tasks that component providers and testers have to commonly perform at present. This section describes the learning strategy and how it supports the evaluation process.

The proposed learning mechanism uses monitoring data to infer performance information on the managed components. 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 *reliability 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.

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: i) raw monitoring data, as collected from the system; ii) inferred performance information clusters, as obtained by analysing and processing the raw monitoring data. The methodology used to infer performance information from raw monitoring data is subsequently described.

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 [Figure 2]; 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 2. The manner in which new data samples are used to update the existing information of similar clusters is described next.

The value of each parameter in a new data sample is used to update the value of the corresponding parameter in the existing inferred sample 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

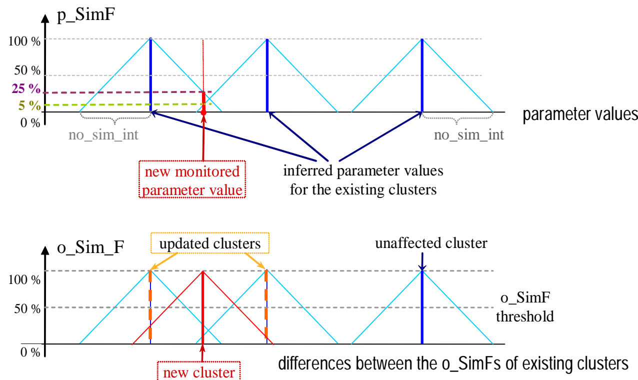

Figure 2: The *learning* process: updating current performance information with new monitorng data

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.

The rest of this section describes how the evaluation process uses the inferred performance information in the available clusters. 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; 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, 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. The current implementation selects a single cluster to represent each redundant component. From these representative clusters, the ones with information that indicates possible availability problems (e.g. thrown exceptions) are being discarded. As a next step, the performance parameter values of the remaining clusters are being compared (i.e. response times, throughput and resource usage). 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.

3. FRAMEWORK PROTOTYPE FOR J2EE

J2EE is the component technology standard specified by Sun Microsystems for building multi-tiered enterprise applications. J2EE specifies different component types for implementing the various enterprise application tiers, such as *servlets* for the web tier and *Enterprise JavaBeans (EJBs)* for the application tier. EJBs are server-side software components that encapsulate application business logic. At runtime, EJBs are managed by *EJB containers*, which represent runtime environments within the J2EE server [Figure 1]. EJB containers provide system-level services, such as transactions, security, or lifecycle management to deployed EJBs. This provides a clear separation between the business logic and the middleware services of distributed enterprise systems built using this technology. Xml configuration files are used to specify the manner in which the server container must manage individual EJBs at runtime. EJB configuration files are bundled together with the EJB business logic into deployable component packages, or archives. An important propriety of EJB applications is that all client calls to EJB instance methods must go through the EJB container [Figure 1]; clients can never access EJB instances directly. This provides the opportunity for intercepting, analysing and processing all accesses to EJB instances.

This section discusses AQuA_J2EE, the current framework prototype implementation for J2EE systems. Certain 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 to interact with the managed component-based system, in order to obtain monitoring data and perform swapping operations, respectively. These are system-dependent functionalities that need to be customised for each targeted platform. As presented in [2], several approaches exist for implementing such system-dependent functionalities. For the current framework prototype, the adopted approach was to modify the application server on which the targeted managed applications were deployed and run; JBoss was selected as the J2EE application server used. This section describes the way each framework functionality was implemented for the current AQuA_J2EE prototype.

The *monitoring* functionality was implemented by instrumenting one of JBoss's container interceptors. The interceptor was enabled to extract monitoring data from all incoming and outgoing EJB method calls. 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 and to determine method call paths through the J2EE application. Call path information can be used to dynamically create accurate models of the running application [2]; no requirements are placed on application assemblers to provide such models. Monitored data samples are sent to the anomaly detection functionality and to the learning module for further processing. Future work will enable the monitoring module to intercept exceptions thrown during system execution.

The *anomaly detection*, *evaluation* and *decision* functionalities were implemented using a *decision policy*-based approach. Decision policies were implemented using the ABLE

Rule Language (ARL)[4]. Rules are specified in dedicated *arl* files, separated from the rest of the framework implementation. As such, rules can be added, deleted and modified by system managers without the need to understand, modify, or re-compile any of the underlying framework mechanisms. Different inference engines can be specified for executing different rule sets (e.g. forward or backward chaining, script, or fuzzy engines). Several types of rules, or decision policies, have been devised, so as to implement the anomaly detection, evaluation and decision functionalities. The main roles of these policies are discussed next.

Detection policies analyse the incoming monitoring data and determine the circumstances in which system optimisations are necessary, or possible. Detected cases are signalled to the evaluation module. For the presented testing scenarios, detection policies were used to spot cases in which monitored parameter values (i.e. load, response time and throughput) crossed certain, predefined thresholds. Additional policies were also implemented in order to avoid false or cascaded alarms. As such, occurrences of small variations across the specified thresholds are being ignored and not signalled to the evaluation functionality.

Evaluation policies are used to determine the optimal redundant components in the current execution environment. Currently, these policies specify which redundant components are optimal in which execution conditions; this information was obtained from extensive tests run on the targeted application and execution platform. The learning mechanism will subsequently be used to obtain and update this information instead. For each evaluation rule, the current environment parameters (e.g. workload) are compared against the parameters specified in the rule's conditions. In case of a match, the rule is triggered, its action indicating the optimal redundant component.

Adaptation decision policies are used to determine whether the system should actually be adapted; if yes, the system is reconfigured by activating the optimal redundant component suggested by the evaluation policies. In the current implementation, the redundant component indicated as optimal is selected for activation. Additional policies are then used to prevent reactions to false alarms. Conforming to these policies, the application will not be adapted if another adaptation operation was executed within a certain preceding interval. This avoids cascading adaptation decisions to be triggered based on monitoring data obtained during recent system adaptations. Additional policies will be subsequently specified to also 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, since the monitored adaptation operations proved to have a limited, sustainable impact on system performance. Decision policies will also be designed to deal with eventual, conflicting optimisation demands.

The current implementation of the *component activation* tier is based on the hot-deployment facility provided by JBoss. The main difficulty with this 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, when JBoss performs a hot-deployment operation, it first undeploys 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. This problem was solved by intercepting and delaying all incoming requests for the duration of the hot-deployment operation. A second problem occurs 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. 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 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 the swapping process 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.

In the performed tests, several redundant components containing Entity beans were dynamically swapped to accommodate variations in the incoming workload. As all Entity beans in the tested application happened to be called by Stateful Session beans, any swapping operation needed to wait for at least the length of a client session before it could be executed. Thus, the presented test-case represents one of the worst-case scenarios in terms of induced delays and throughput decline during the adaptation operations. Even so, the measured performance overheads were limited. No availability concerns were raised as no exceptions were thrown and none of the client transactions expired.

4. TESTING SCENARIOS AND RESULTS

4.1 Test application

An enterprise banking application, the Duke's Bank, was used to demonstrate AQuA_J2EE's performance management capabilities. Duke's Bank is a sample J2EE application that allows customers to perform banking operations online. Such operations include accessing account histories and performing banking transactions. Duke's Bank is designed as a typical three-tiered enterprise application, with a web, application and a database tiers, respectively. There are three main business entities in the Duke's Bank application: customer, account and banking transaction. Each of these business entities is implemented by a separate Entity bean in the application tier and by a corresponding table in the relational DB. The application is designed so that all Entity beans are accessed via Stateful Session beans. Conforming to the EJB specification, instances of Stateful Session EJBs maintain their state for the entire duration of a client session accessing them. Thus, in the Duke's Bank, the Entity bean instances must also be maintained available for

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

the entire duration of the client sessions using them. This implementation detail had an important role in the outcome of the presented test cases and adaptation scenarios.
Several redundant components were built and used for enabling the 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 these 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 conditions. The max-bean-age parameter dictates the amount of time an inactive EJB instance is kept in a cache before being passivated (i.e. the instance's state is persisted and the instance is removed from the cache). The idea behind this strategy is to avoid consuming caching resources for EJB instances that are no longer being used. However, 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 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 maximum-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 thrown causing system availability to degrade. To avoid the occurrence of such 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. Nonetheless, when the system is lightly loaded, extended max-bean-age configurations mean that EJB instances are kept in the cache for long periods, even when sometimes no longer reused or needed by the application. Memory resources are being inefficiently used in effect. Ideally, the managing application server (e.g. JBoss) would '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, 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. 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.

4.2 Testing platform

Three stations were used for installing the J2EE system and performing the tests. One station was used for running the J2EE application on JBoss, a second one for running a relational DB and a third one for simulating client activity on the 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 the Duke's Bank application. It was run on Microsoft Windows Server 2003 Enterprise Edition, on an Intel Pentium III processor at 866MHz and 128 MB of RAM. The OpenSTA[5] 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[6] tool was used to measure memory consumption for the Java process which ran the J2EE application and JBoss server.

4.3 Testing scenarios and procedures

Two redundant components were devised for the Entity EJBs employed in the tested usage scenarios. Each redundant component was prepared so as to be optimal under a different system load. The redundant components were built to differ at the instance caching configuration level. 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. Duke's Bank was tested under two different workload conditions, starting with a low workload (i.e. 15 users) then increasing the workload for a certain period (i.e. 60 users) and then decreasing it back to the initial lower workload. For this purpose, the OpenSTA load generator was used to simulate different numbers of concurrent users accessing the application. The usage scenario run by each user involved several operations: the user logs in, lists all the banking transactions of a selected account and logs out, terminating the client session. Each user was configured with a unique identity, meaning that once a user's session was completed, the EJB instances of that user were no longer needed; at that point, these instances should be promptly removed from the cache, to free system memory.
Two different scenarios have been tested: one in which the application adaptation was *not* employed (i.e. AQuA_J2EE was *not* integrated with JBoss) and another in which application adaptation was used to optimise the application to changes in its running environment.

4.4 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; the initial 10_bean_age variant was consequently swapped with the new 500_bean_age variant. During the interval immediately following this adaptation operation, the special-purpose policies in the decision module prevented further adaptations to be performed, so as to avoid oscillating adaptations. Such situation would have occurred in this case due to the decreased workload detected on the adapted components, during the actual swapping process. The workload decrease was caused in this case by the component activation process, which blocked requests on RGs while swapping their redundant components. Nonetheless, the detection module is unaware of this aspect when interpreting monitored data. It consequently alerts the evaluation function, which determines the optimal redundant

[5]Open System Testing Architecture: www.opensta.org
[6]Tagtraum, GCViewer: www.tagtraum.com/gcviewer.html

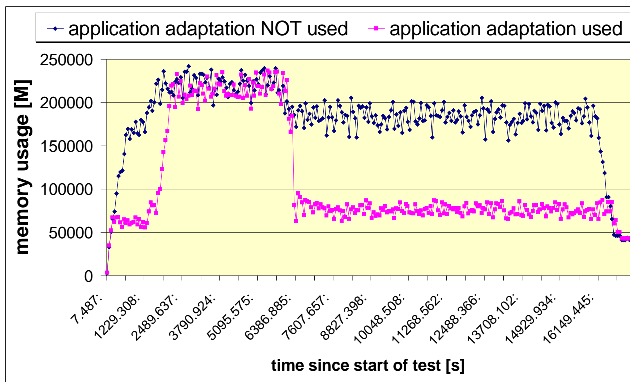

Figure 3: Adaptation impact on memory usage

component in the apparent low workload context. Activating this recommended component at this point would be undesirable management behaviour; the 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. Continuing the tested scenario, when the workload later decreased back to a low level, the application was automatically adapted again, in a similar manner, so as to use the 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.
Figure 3 shows the memory usage levels recorded during the two separate test scenarios. Results clearly indicate the benefits of the application adaptation on 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 the 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 (database 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 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. [12]). 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. This application consumed about 150MB of memory. In this scenario, the memory saving benefits could be observed at the performance level of the 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 the 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-consumption application, availability issues were solved, but response times of the Duke's Bank were impacted. Figure 4-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 [s]) 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 4-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 be considered when evaluating an adaptation operation, to ensure the potential benefits would outweigh the induced overheads. The impact the presented solution has on other system quality attributes, such as reliability, will also be considered.

The 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 in-

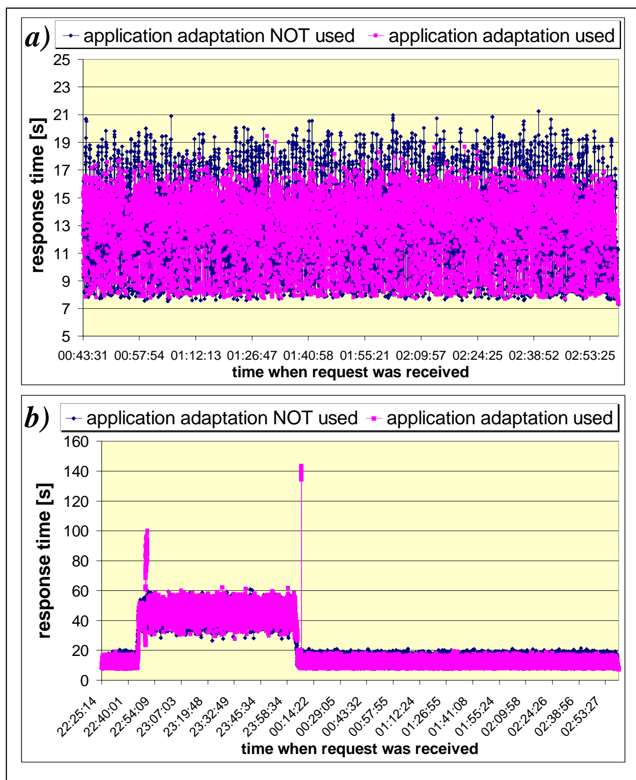

Figure 4: Application adaptation impact on response times - limited memory availability: a) low workloads b) low and increased workloads

creased workloads, but is sub-optimal under low workloads.

4.5 Discussion

A few notes are in order for showing the way monitored parameter values should be interpreted in the tested scenarios. In the exemplified application, the parameter that ultimately dictates which redundant component is optimal at anyone time is the component *response time*. It was previously shown that the inactivity of a cached EJB instance can be interpreted differently, depending on the system load: under low loads, it means the instance is no longer needed and can be discarded; under high loads, it can also mean that the instance is blocked waiting for needed resources to become available. 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. Namely, 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 successfully be 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; thus they would not influence the monitored component workloads. In such cases, the combined variations of workloads and system resource usage need to be considered. This 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.

5. RELATED WORK

To the best of our knowledge, there are no similar frameworks that employ monitoring, learning and adaptation for applications based on contextual composition frameworks [11], at the component level. General frameworks for self-adaptive systems are presented in [8] and [5], 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 [11].
Component redundancy-based adaptation techniques, such as presented in [13] are similar to the application adaptation approach in AQuA. 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 for each redundant component, or replacement mechanisms for each separate pair of redundant variants. The decision algorithm presented in [13] can be used to specify AQuA's decision policies, for systems in which the adaptation operations take considerable periods to complete.
Efforts towards standardizing and implementing hot - deployment functionalities in J2EE applications are being made in research initiatives such as [10] and [7]. The authors intend to comply with specified standards and possibly adopt available hot-deployment implementations for AQuA_J2EE.
Several projects, such as JAGR [1] and JADE [9], propose automatic frameworks for managing the availability and dependability of component-based enterprise applications, with focus on J2EE systems. JAGR [1] uses component level micro-reboots as a 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 is equivalent with re-deploying or re-booting that component. JADE [9] 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, transactions), or individual application components (e.g. EJBs). AQuA 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.

6. CONCLUSIONS AND FUTURE WORK

Performance management of complex, enterprise software systems is becoming an increasingly difficult process. Completing this process manually is consequently becoming more costly and error-prone [6]. This paper presented AQuA, a framework for automating the management of component-based enterprise systems. AQuA provides the means for automating administrative tasks which system managers should commonly perform at present. Administrators can use their knowledge to configure the AQuA framework on how to automatically manage applications. No extra requirements are placed on component providers or the component technologies used. A prototype framework, AQuA_J2EE, was implemented and tested for managing a sample J2EE application on JBoss. Presented results showed the benefits of automatic application management on system performance and availability. An automated learning mechanism for acquiring component performance information was devised. Future work will focus on integrating this mechanism with the current AQuA_J2EE implementation and testing its efficiency on several sample systems. A number of additional applications with diverse adaptation requirements will be tested, in order to validate and extend AQuA_J2EE's management capabilities as necessary.

In summary, the main contributions of the presented research consist of: i) indicated the existence of a performance optimisation problem, namely, the difficulty of devising and managing a single component implementation or configuration that is optimal under all possible running conditions; ii) implemented and tested example applications in which the presented problem occurred, using the targeted component technologies; iii) proposed a component redundancy-based solution for addressing this problem; iv) devised AQuA, a management framework for implementing the proposed solution and automatically managing applications' quality; AQuA's general design presents several distinctive characteristics, such as the learning facility and the combined decentralised and centralised decision processes, which differentiate it from similar work in the area; v) implemented AQuA_J2EE, a framework prototype for managing J2EE applications, on JBoss; this involved finding and implementing solutions for monitoring and swapping EJB components on JBoss; vi) showed the benefits of using AQuA_J2EE for automatically managing an example J2EE application.

When automating the management process and allowing applications to re-configure themselves, it is paramount to ensure that system dependability and performance are not actually put at risk, at any point. Ideally, the framework will perform similar operations to those an expert human manager normally would, except it will do so at lower costs and in a less error prone manner. Thus, the goal is to capture the existing expertise of human system managers and implement it as an automated process that accurately simulates current management best-practices. AQuA aims at achieving this goal, for complex component-based systems. It currently offers the means to specify system management knowledge in terms of decision policies. The learning mechanism provided will initially support the framework and the system administrators in taking informed management decisions when reconfiguring application components. Subsequent releases will also be capable of inferring and adjusting the actual decision policies, at runtime.

7. REFERENCES

[1] G. Candea, E. Kiciman, S. Kawamoto, and A. Fox. Autonomous recovery in componentized internet applications. *Cluster Computing Journal*, 2005.

[2] A. Diaconescu, A. Mos, and J. Murphy. Automatic performance management in component-based software systems. In *International Conference on Autonomic Computing (ICAC04)*, NY USA, 2004.

[3] 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), ICSE03*, Portland, Oregon USA, May 2003.

[4] 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), ICSE04*, UK, 2004.

[5] 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.

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

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

[8] 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.

[9] N. D. Palma, S. Bouchenak, D. Hagimont, and F. Boyer. Autonomic administration of clustered j2ee applications. In *International Multi-Conference in Computer Science & Computer Engineering*, Las Vegas, Nevada, US, June 2005.

[10] Sun Microsystems. *JSR88: J2EE Deployment API Specification*, November 2003. http://java.sun.com/j2ee/tools/deployment/.

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

[12] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. *SIGOPS Oper. Syst. Rev.*, 2002.

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