A Component Redundancy Framework for Automatic Performance Management of Enterprise Applications

Ada Diaconescu*
Performance Engineering Laboratory
Engineering and Computing, Dublin City University
Glasnevin, Dublin 9, Ireland
+353-1-700-7644
diacones@eeng.dcu.ie
ACM membership number: 2076784
Advisor: Dr. John Murphy

1. Problem and Motivation

Component technologies [1], such as EJB and .NET, are being increasingly adopted for building complex enterprise software systems. The reason for this is they promote software modularity and reusability, thus reducing time to market and decreasing development, testing and management costs. However, the particular characteristics inherent to component technologies, including component encapsulation, dynamic inter-component binding and application server complexity, introduce new challenges to building and managing large-scale enterprise systems [2].

Software systems need to be functionally correct, as well as provide quality guarantees, such as performance, reliability or robustness. In component-based applications, the individual behaviour of each component involved 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 addition, runtime system changes, caused for example by component versioning or platform reconfigurations, and execution context variations, caused for example by incoming workload fluctuations, can render initial performance optimisations obsolete. This is due to the fact that different application integration, design and implementation strategies are optimal in different running contexts [3]-[8]. Therefore, performance testing, analysis and optimisation operations need to be performed not only statically, or off-line, but also during system runtime, as continuous or periodic tasks. Manual execution of such complex management operations is highly expensive and error-prone. The solution is

* The author’s work is funded by Enterprise Ireland Informatics Research Initiative 2002
to automate the management of complex systems [10], to more easily and reliably maintain their quality characteristics and decrease management costs.

The presented work is targeted at automating the performance management of component-based enterprise applications. The application server is enhanced with new capabilities that enable it to automatically adapt and optimise the performance characteristics of running applications. A component redundancy-based approach was adopted for dynamically adapting applications, optimising their performance in changing conditions of their execution environments.

2. Background and Related Work

Considerable research and work has been carried out for automating software system management processes. This includes efforts for specifying and building adaptive and autonomic software systems, featuring self-optimisation and self-healing capabilities [3] or [8]-[12]. However, to the best of our knowledge, no frameworks have been devised for performing automatic component-level monitoring, diagnosis, and adaptation operations for applications based on contextual composition frameworks [1] such as EJB or .NET. The particular characteristics of such applications entail special requirements for the management systems administering them. As a result, certain aspects of the existing management frameworks, which target different types of component-based applications, can prove difficult to simply apply to component-based applications based on contextual composition frameworks [1]. For example, it is typical for the execution platforms of applications built using component technologies such as EJB or .NET to be highly complex. For instance, the execution platform for EJB applications includes an EJB-compliant application server, one or several Java Virtual Machines (JVMs), an Operating System (OS) and hardware platform. Each of these layers can be acquired from various providers, exhibit different runtime behaviours and performance characteristics and provide diverse configuration options. Thus, understanding the performance of complex component-based applications running on such platforms becomes a difficult task, at best. As a result, static or off-line practices for estimating or predicting the performance characteristics and/or resource requirements of individual components (e.g. individual EJBs) to be deployed and run on such platforms can prove unfeasibly difficult and time-consuming. In addition, the exact products and configurations to be used as part of the execution platform may not be known in advance to component developers or providers. For certain types of component-based applications, in which one component can be represented by a C++ library executing directly on top of the OS, by an entire server-side application, or by a Web service, accurate performance predictions at component level, may be feasible. However, for the aforementioned reasons, the same performance prediction practices would be difficult to apply, if at all, to EJB applications at the fine-grained EJB component level. The presented work aims at solving this issue by employing runtime monitoring, information analysis and learning processes for generating performance-related knowledge at both component and overall application levels.

General frameworks for self-adaptive systems are presented in [8],[11] or [12] with similar goals to our proposed framework. The presented research work aligns with these frameworks, employing inter-related monitoring, analysis and adaptation
modules, while specifically targeting enterprise applications based on contextual composition middleware [1].

Component redundancy-based adaptation techniques, such as presented in [3], or [8] are similar to the presented application adaptation approach. The adaptation strategy adopted in the presented research differs from the aforementioned approaches in one main significant aspect; it requires component providers to supply neither accurate resource requirements or performance information for each delivered component variant, nor replacement mechanisms for each pair of redundant variants. The presented framework can use initially provided information if available, but also employs monitoring, analysis and learning processes to obtain and validate performance information during runtime.

Redundancy as a means of achieving dependability for Internet systems (i.e. Web services-based) is proposed in the RAIC [8] project. The addressed problem domain of this project however, is somewhat different in scope from the presented work. This is because Web services systems typically rely on services offered by different providers, from different locations. No single enterprise has complete control over the entire distributed system. A system developer is unlikely to have any knowledge of, or access to the implementations or execution platforms of the Web services they need to use. Therefore, in such cases, (redundancy-based) management support cannot be provided at each Web services site involved. Instead, in RAIC, management support is implemented at the Web services client site, more specifically, built into the software application that uses the remote Web services [8]. Thus, while the RAIC approach can be used to select the optimal redundant Web service site for a certain client, the presented framework can be used to optimise and manage the performance of a certain Web services site. This makes the two projects complementary rather than related.

3. Uniqueness of the Approach

The presented framework is aimed at enabling applications to manage themselves, so as to dynamically self-optimise and adapt to changes in their internal software configurations and external environmental conditions. The proposed strategy is to provide different component implementation variants at runtime, with equivalent supported functionalities, and to enable applications to automatically analyse and select the most appropriate one(s) to use in each particular execution context. The concept of component redundancy is introduced to support this idea and the framework for implementing the proposed solution is presented over the following subsections.

Some of the key features of the proposed performance management solution include:

- Separation of application business logic from management code, increasing system flexibility and manageability
- No extra-requirements imposed on component developers or providers. Any standard component can be managed by the proposed framework. No detailed component performance information or component resource requirements information is required at deployment time. It is not compulsory for multiple
component variants to be provided and available at runtime. Component variants can be acquired from multiple providers.

- Support for both centralised and decentralised management approaches
- No extra framework implementation or installation efforts required for managing new applications. The framework is implemented once and then used for managing all applications subsequently deployed and run on a certain application server
- Framework modularity, allowing for each functional module to be built separately and/or subsequently replaced without impacting on the other modules
- Clients remain completely unaware of runtime system management operations.

3.1 Component Redundancy

*Component redundancy* is a concept defined as the availability, at runtime, of multiple component variants providing identical or equivalent functionalities but with different design and implementation strategies. These component variants are referred to as *redundant components*. A set of redundant components providing an equivalent functionality, or method, constitutes a *redundancy group* (with respect to that functionality). Any component variant in a certain redundancy group can be functionally replaced with any other component variant in the same redundancy group. However, each variant should be optimised for a different execution context, such as for different incoming workloads, inter-component communication patterns, or available system resources. Only one of the redundant components providing certain functionality is assigned for handling a client request for that functionality. The selected variant is referred to as the *active* component variant. Redundant components can be dynamically added, updated, or removed from the system. Redundancy groups can contain atomic components, as well as composite components or sets of components [1] as depicted in Figure 1.

Figure 1: component redundancy granularities:

a) atomic component; b) composite component; c) set of components
3.1.1 Example

An example scenario was implemented and tested to show the applicability of the component redundancy-based approach [6]. The Enterprise JavaBeans (EJB) component technology was selected for implementing the example.

Two different component variants were implemented for providing the same functionality: repeatedly retrieving information from a remote database (DB). The two component variants differ at the design level. The first design variant (session-only variant) involves a single Session Bean, containing SQL code for directly accessing the remote DB. In the second design variant (session façade variant), a Session Bean uses an Entity Bean as means of interacting with the database. The Entity Bean acts as a local cache for data in the remote DB.

The response delays for each component variant were measured in different environmental conditions (i.e. available bandwidth on the network link to the remote DB). When the link to the DB was lightly loaded, the session-only variant proved optimal (i.e. smaller response times). However, for increased network loads the session façade variant became the optimal variant [Figure 2]. This was caused by the fact that the inter-process communication and CPU overhead of the session façade variant became lower than the repeated database access overhead introduced by the sessions-only variant.

![Figure 2: Example Scenario: response-time variation with network load for different component variants](image)

These test results indicate that an informed alternation of redundant components, each one optimised for a different execution context, would provide better overall performance than either component alone could provide.

3.2 Framework Description

The presented framework was developed for supporting and managing redundant components. It capitalizes on component redundancy for continuously optimising applications and adapting them to changing environmental conditions, in order to meet their performance goals. The framework is divided into three functional modules: monitoring and diagnosis, evaluation and decision and component activation [4]-[7]. The three modules enable software systems to:
i) monitor themselves and their environment 
ii) analyse collected monitoring data and detect performance problems 
iii) evaluate available adaptation and optimisation alternatives 
iv) decide on changes to be performed in order to overcome problems detected and/or improve performance parameters 
v) dynamically enforce decisions taken by modifying applications at runtime.

This process functions in a feedback-loop manner. The monitoring module collects performance information on the newly adapted application. This information can then be analysed for assessing the benefits of the used adaptation strategy. Thus, management framework instances can learn and improve their adaptation logic over time. Special purpose evaluation and decision logic is used to address stability issues, which may be induced by the feedback-loop (‘Decision Policies’ Section). The main roles and functionalities of each module are presented in more detail in the following subsections. The framework aligns with the autonomic computing initiative proposed by IBM [10] for automating management of complex systems.

3.2.1 Monitoring and Diagnosis

The monitoring and diagnosis module is concerned with obtaining run-time information on software applications, including response times and throughputs, as well as on the applications’ execution environments, including incoming workloads, or available CPU and I/O resources. Only active component variants are being monitored. Collected information is analysed and potential ‘problem’ components identified. The evaluation and decision module (Section 3.2.2) is notified of problems detected.

Runtime application monitoring and basic problem-detection facilities are supported by the current framework implementation; parts of the COMPAS monitoring platform [4],[13] have been used for achieving this (Section 4).

3.2.2 Evaluation and Decision

The evaluation and decision module determines which redundant component(s) to activate and when, in order to optimise application performance. This functionality requires:

i) accumulating information on the available redundant components and their running environment 
ii) processing available information and determining the optimal redundant component(s), in certain contexts.

Information is collected by the monitoring and diagnosis module at runtime and then processed and stored in formal descriptions (‘Component Descriptions’ Subsection). Component descriptions and current monitoring data are used as input to various types of decision policies (‘Decision Policies’ Subsection); decision policies constitute the performance management logic of the framework.

Two main approaches are being considered for designing and implementing the evaluation and decision module, with respect to module distribution [4],[6]. The first is a centralised approach, in which evaluation and decision tasks are performed
globally for the entire application. The second approach is to perform such tasks locally, for each individual component, or group of components, in a decentralised manner. While centralised management is potentially more accurate, decentralised management is more scaleable. As explained in [4] and [6], a combined solution seems to be optimal with respect to performance optimisation versus management overhead tradeoffs. The mixed solution involves evaluation and decision modules with various scopes, such as managing single components, component groups, or the entire application. Individual evaluation and decision modules intercommunicate via a clearly specified protocol. This approach allows for local application problems (e.g. at component level) to be solved locally, when possible, while also supporting global optimisations, when necessary. Complex management logic, required for administering large-scale, dynamic systems, may prove difficult to devise in a centralised manner. Rather, a decentralised approach would allow for the needed complex management behaviour to emerge from multiple, interacting evaluation and decision modules with reduced scopes. Decentralised management also allows for evaluation and decision modules at different levels to be dynamically activated or deactivated as needed, in order to minimise overhead. Future efforts will evaluate and test the advantages of the two approaches.

Work is underway to design and implement the evaluation and decision module (Section 4). This includes work on specifying the module architecture, component descriptions and decision policies.

Component Descriptions

Component performance information is obtained by the monitoring module, at runtime. This information is processed and formally represented as a component description, facilitating its automatic interpretation, analysis and modification. In addition, component providers can optionally supply initial component descriptions, to be available at deployment time. Initial information can be acquired from test results, estimations, or previous experience with provided components. Consequently, if available, initial information is to be used as general guidelines rather than as absolute figures, since it was obtained in conditions that were different from the current system ones (e.g. different incoming workloads, used application server, JVM, OS, or hardware platform). Initial descriptions are subsequently verified and updated at runtime with accurate monitoring information for the actual execution contexts. The evaluation and decision module can thus ‘learn’ over time about the performance characteristics of the software components and application it has to manage. This includes knowledge on optimal individual redundant components, as well as on optimal combinations of redundant components. Current efforts focus on determining and specifying standard ways of representing, analysing, validating and updating component information.

Decision Policies

Decision policies are sets of rules, dictating the actions to be taken in case certain conditions are satisfied. They can be customised for each deployed application, in order to serve the specific application goals, such as requested performance attributes, their values and criticality. Decision policies can be added, modified or deleted at runtime. The use of decision policies separates performance management strategies from application data and business logic.

Rules can be split into two main categories: basic rules and high-level rules.
There are several types of basic rules, based on their intended functionality [5]:

i) detection rules, for identifying performance problems. They are used to analyse monitoring data as it becomes available, searching for patterns that would indicate the occurrence of a performance problem

ii) adaptation rules, for evaluating the current situation and finding optimal solutions for overcoming detected problems. They are used to process the available relevant information and take optimisation and adaptation decisions

iii) inference rules, for deducing new facts or information, from existing, collected data. They are used to analyse existent component descriptions and current monitoring data, inferring new information and validating or modifying existing information. For example, inferred information could indicate the optimal component variant in a redundancy group, for certain execution contexts. When taking optimisation decisions, adaptation rules favour the usage of inferred information, if available, to the repeated analysis of unprocessed monitoring data. This approach can reduce the overhead and improve the efficiency of the adaptation process.

High-level rules are used for analysing the behaviour of basic rules, in order to detect and rectify unwanted management behaviour. Such behaviour can consist of a set of rules being fired repetitively, or entering an infinite loop, and causing oscillating states in the system, chain reactions, or inefficient reasoning (e.g. not being able to reach a decision after consuming considerable amounts of time and resources).

3.2.3 Component Activation

The component activation module is responsible for dynamically enforcing optimisation decisions into the managed application. This involves activating the redundant components indicated by the evaluation and decision module as being optimal. A request indirection mechanism is used for implementing this functionality. Incoming client calls are directed to an instance of the active component variant upon arrival. When the active component is changed, new incoming requests are directed to instances of the new active component. State transfer is not needed in this case, as client requests are not transferred between instances of different components; a particular interaction always finishes execution with the component instances it started with. An instance of this mechanism has been implemented for the Enterprise JavaBeans (EJB) technology (Section 4).

4. Results and Contributions

Current implementation work focuses on managing the performance of Enterprise JavaBeans (EJB) applications, running on the JBoss [14] application server. Nonetheless, the specific framework implementation can be modified to work with other EJB application servers, or other component technologies.

Proof-of-concept versions of the monitoring and diagnosis and the component activation modules have been implemented [4], as shown in Figure 3. Work is underway to automate the evaluation and decision module functionalities, currently performed by a human system manager.
The JBoss containers and instance pools have been modified, so as to be able to extract runtime information on individual method calls, call paths through the system and the number of instances created for each component. Method call information, such as response times, can be analysed over time for detecting performance problems or suboptimal behaviour. Call path information can be used to detect component interaction patterns and their impact on global performance characteristics. Component instance information can be used for example, for selecting redundant components with optimal instance pool size configurations, for different incoming workloads.

Part of an existing, generic EJB monitoring platform, the COMPAS [13] client, has been integrated with the JBoss server, in order to provide part of the framework’s performance monitoring and diagnosis functionality. COMPAS receives monitoring data from the running JBoss server and displays it in a graphical console in real-time. COMPAS is also able to analyse collected information and detect performance problems; new problem detection strategies have been added to COMPAS. This facility helps human system managers to observe system performance behaviour at runtime, and decide on system or framework re-configurations. Such reconfiguration decisions can also be assisted, or entirely performed by an automated evaluation and decision module [4].

A human manager currently needs to perform evaluation and decision-related tasks, assisted by the automated monitoring and diagnosis facilities provided [Figure 3]. A graphical user interface (GUI) is currently provided for accessing the automated component activation functionality. The system manager uses this GUI to enforce performance optimisation and adaptation decisions into the running system. The current framework implementation for the evaluation and decision module uses the Jess rule engine [15] for storing and interpreting data and decision policies.

The component activation module has been implemented and tested on the JBoss application server. The currently adopted approach is based on the dynamic modification of deployment descriptor configurations, as presented in [4].

The presented framework uses component redundancy to automatically optimise the performance of complex software applications and dynamically adapt them to changes in their execution environment. Clients of the system remain unaware of such runtime management operations. The framework is designed to be independent from the individual applications it manages. It employs monitoring, analysis and learning.
processes to collect and infer performance information on managed components. Therefore, it imposes no extra requirements on component developers or providers. Certain framework functionalities have been implemented and tested for the JBoss application server platform. Work is underway to complete a proof-of-concept framework implementation.

5. References