A Framework for Using Component Redundancy for Self-Adapting and Self-Optimising Component-Based Enterprise Systems

Ada Diaconescu
Performance Engineering Laboratory
Dublin City University, Ireland
+353-1-7007644
diacones@eeng.dcu.ie

ABSTRACT
We propose a framework that uses component redundancy for enabling self-adaptation, self-optimisation and self-healing capabilities in component-based enterprise software systems.

Categories and Subject Descriptors
D.2.9 [**Software Engineering**]: Management – *software configuration, management, software quality assurance (SQA).*

General Terms
Management, Performance, Design.

Keywords
Redundancy, middleware, self-adaptation, self-optimisation, decision policy, component-based enterprise systems.

1. INTRODUCTION AND PROBLEM
A significant problem in the information technology (IT) industry at present is *complexity* [1]. The extensive use of software systems in various domains imposes certain requirements on their quality characteristics (e.g. performance, dependability). Thus, building, managing and optimising such complex systems is becoming a growing concern. Component technologies [2], such as EJB, CCM or .NET, address many of the complexity related difficulties, by facilitating software modularity and reusability. Nevertheless, such technologies introduce new challenges. The way individual components behave and interact in a system, as well as their runtime environment, strongly influence global system performance. However, lack of system internal information, plus the dynamic nature of component-based applications, makes the performance of complex systems hard to analyse and predict. Component developers do not know the running context(s) of their components and application integrators do not have access to component internal information. In addition, runtime system modifications and execution context changes can render initial optimisations obsolete, as different design and implementation strategies are optimal in different running contexts [3], [4]. Thus, ensuring quality guarantees for complex component-based systems becomes a challenging task at best.

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

2. PROPOSED SOLUTION
We propose enabling component-based applications to automatically change their implementation at runtime in order to tune themselves and continuously adapt to variations in their environment (e.g. workload, usage patterns, available resources). For this goal to be met, the following must be provided: i) different design and implementation strategies for software components, available at runtime; ii) a mechanism for automatically alternating the available strategies at runtime, as needed for reaching the high-level goals of software applications.

2.1 Component Redundancy
Component redundancy is a concept we introduce for addressing the former requirement. It means the presence, at runtime, of multiple component variants providing identical or equivalent services but with different implementation strategies. We refer to these component variants as *redundant components*. Only one of the redundant components providing a service is assigned, at any moment in time, for handling a certain client request for that service. The selected variant is referred to as the *active* component variant. Redundant components can be added, updated, or removed at runtime.

We implemented and tested an *example scenario* that shows the applicability of our approach [4]. The EJB component technology was used for implementing this example. Different strategies were selected for implementing two distinct component variants providing the same functionality: repeated retrieval of information from a remote database (DB). The first variant consists of stateless session beans only and uses SQL code for directly accessing the DB. The second variant (session façade) employs a stateless session bean as a wrapper to an entity bean, which encapsulates persistent data. The entity bean acts as a local cache for data in the remote DB. We measured the response delays for each variant, in different environmental conditions (i.e. available bandwidth on the network link to the DB). When the link to the DB was lightly loaded, the session-only variant proved optimal. For increased network loads however, the session façade variant became optimal, as its inter-process communication and CPU overhead became lower than the repeated database access overhead introduced by the sessions-only variant. As these results indicate, knowledgeably alternating the usage of redundant components, optimised for different running contexts, ensures better overall performance than either component variant could provide.

2.2 Our framework

We propose a framework for supporting and managing redundant components, capitalizing on their redundancy to continuously adapt and optimise applications and meet their quality goals (e.g. response times, throughputs). The framework is divided into three main logical tiers: monitoring, evaluation and action.

The *monitoring* tier is concerned with obtaining run-time information on software applications, exclusively on active components (response times, throughput), as well as on their execution environments (incoming workload, CPU, I/O usage). Collected information is analysed and potential 'problem' components identified.

The *evaluation* tier is responsible for deciding which components to (in)activate and when, in order to obtain quality improvements. This involves two main activities: i) accumulate information on components and their running environment; ii) process information and determine the optimal redundant component(s), in certain contexts. Component information is represented as a formal component *description*. Component providers can optionally supply initial component descriptions, at deployment time. An initial description can indicate the used implementation strategy, or the running context for which a component was optimised. It can also provide relative quality attribute values, and/or their variation with environmental conditions. This sort of information can be acquired from testing results, estimations, or previous experience with provided components. These descriptions are then updated at runtime with accurate monitoring information for the actual execution contexts. Component descriptions and runtime monitoring information are used as input to *decision policies*. These are sets of rules, dictating the actions to be taken in case certain conditions are being satisfied. Decision policies can be customised for each deployed application, for serving the specific application goals (e.g. requested quality attributes and their values) and can be added, modified or deleted at runtime.

The *action* tier enforces decisions taken in the evaluation tier into the running application, using a request indirection mechanism. That is, incoming client calls are directed to an instance of the active component variant, upon arrival. When the active component changes, 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 executes with the component instances it started with.

The three logical tiers operate in an automated, feedback-loop manner: application performance is monitored and evaluated; optimal redundant component(s) are identified and activated (action); and the resulting application is (re-)monitored and (re-)evaluated. Component descriptions and decision policies are tuned, or updated in effect. The evaluation tier can thus improve its decisions, in time, as it gradually 'learns' about the performance and behavioural characteristics of the component-based application it has to manage.

We are implementing our framework in a manner that makes it independent of the specific applications it has to manage. Conceptually, our framework belongs to the execution platform (e.g. J2EE) on which applications are deployed and run, being at the same level with already provided services (e.g. security, transaction support, connectivity). This makes our framework transparent to clients of applications deployed on such platforms.

For improved scalability, we decentralise our three-tiered framework and employ different intercommunicating instances of our framework to manage applications at different granularities (e.g. single component, component group, or global application). We organise these instances in a hierarchical manner and specify a clear protocol for their intercommunication. This allows for local application problems (e.g. at component level) to be dealt with locally, when possible, while also supporting global optimisations, when necessary.

We instrumented the open source JBoss application server so that to provide call path related information. This allows instances of our framework dynamically detect when and what method(s), of what EJBs, are being called by a certain method of a certain EJB instance. This information, together with component response time and throughput information, is used to determine whether provided services are meeting their performance requirements, as well as help detect 'problem' components.

3. CONCLUSIONS AND FUTURE WORK

Our main contribution is a framework that uses component redundancy to automatically manage complex software applications and meet their quality goals. The main features of our framework that differentiate it from similar work in the area (e.g. [3], [5]) are its independence from specific components or applications and its decentralised operation. The framework has no strict requirements on the initial information to be provided on the deployed components. It is devised to be able to collect information, and learn in time about the component-based application it has to manage. Management activities at different granularity levels (e.g. component, application) can be switched on and off dynamically, as needed for meeting quality goals, while introducing minimum overhead. The evaluation and decision mechanism is critical to our approach and is therefore the focus of our ongoing research. We intend to adopt existing solutions, relevant to our framework, such as monitoring solutions (e.g. the COMPAS project [6]), rule engines, or learning methods.

4. ACKNOWLEDGMENTS

This work is funded by Enterprise Ireland Informatics Research Initiative 2002s.

5. REFERENCES

[1] J. O. Kephart, D. M. Chess, "The Vision of Autonomic Computing", IEEE Computer, January 2003

[2] C. Szyperski et al. "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, November 2002

[3] Daniel M. Yellin, "Competitive algorithms for the dynamic selection of component implementations", IBM Systems Journal, Vol. 42, no 1, 2003

[4] A. Diaconescu, J. Murphy, "A Framework for Using Component Redundancy for Self-Optimising and Self-Healing Component Based Systems", WADS workshop, ICSE'03, Hilton Portland, Oregon USA, May 3-10, 2003

[5] S. Cheng, D. Garlan, et al., "Using Architectural Style as a Basis for Self-repair", WICSA'02, Montreal, Aug. 2002

[6] The COMPAS project home page: www.ejbperformance.org