A Framework for Automatic Performance Monitoring, Analysis and Optimisation of Component Based Software Systems

Ada Diaconescu[*], John Murphy[**]
Performance Engineering Laboratory
Dublin City University, University College Dublin
diacones@eeng.dcu.ie, J.Murphy@ucd.ie

Abstract

A framework for automating the runtime performance management of component-based software systems is presented. The framework leverages static performance information obtained at component development time, if available, and executes performance monitoring, analysis and optimisation operations during runtime. The dynamic performance optimisation process is based on the automatic selection and activation of one of multiple functionally-equivalent implementation variants, available at runtime, each one optimised for a different running context. The framework consists of three main modules: monitoring & diagnosis, evaluation & decision and component activation. Current implementation work targets Enterprise JavaBeans systems.

1. Introduction and Problem

Component technologies [1], such as J2EE 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. 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 every component and the collective behaviour of interacting components in the specific execution environment determine the overall application performance. Nonetheless, system complexity and lack of information make performance of enterprise applications hard to analyse and predict. In addition, runtime system modifications (e.g. component versioning) and execution context changes (e.g. workload fluctuations) can render initial optimisations obsolete, as different integration, design and implementation strategies are optimal in different running contexts [2], [3], [4]. This imposes that performance testing, analysis and optimisation operations are 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 to automate the management of component-based applications, to more easily and reliably maintain their quality characteristics and decrease managing costs.

This paper proposes a solution for enabling applications to manage themselves, in order 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 implementations at runtime, with the same supported functionalities, and to enable applications to automatically analyse and select the most appropriate one(s) to use in each specific context. The concept of component redundancy is introduced to support this idea and a framework for implementing the proposed solution is presented. The proposed solution aims at imposing no development or testing overheads on component providers. The rest of the paper is structured as follows. Sections 2 and 3 define component redundancy and present the proposed framework, respectively. Section 4 describes current and future work and Section 5 discusses related work. Conclusions are presented in Section 6.

2. 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; each strategy should be optimised for a different execution context (e.g. workload, inter-component communication patterns, available system resources). We refer to these component variants as *redundant components.* Only one of the redundant components providing certain functionality is assigned for handling a client request for

[*,**] The authors' work is funded by Enterprise Ireland Informatics Research Initiative 2001

that functionality. The selected variant is referred to as the *active* component variant. Redundant components can be dynamically added, updated, or removed. The component redundancy concept, as well as an example showing the potential benefits of the component redundancy based approach, is presented in more detail in [4].

3. Framework Overview

A framework is proposed for supporting and managing redundant components, capitalizing on their redundancy to continuously optimise applications and adapt them to changing environmental conditions, in order to meet their performance goals. The framework is divided into three functional modules: monitoring & diagnosis, evaluation & decision and component activation. The three modules enable software applications to i) monitor themselves and their environment, ii) analyse collected 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 improve performance parameters and v) dynamically enforce decisions taken by modifying themselves at runtime. This process functions in a feedback-loop manner; the monitoring module collects performance information on the newly adapted application, which can then be analysed for assessing the benefits of the used adaptation strategy; this allows management framework instances to 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 (Section 3.5).

Each functional module can be designed and implemented independently, as well as subsequently replaced without affecting the other modules. Thus, various strategies can be separately selected for determining performance problems and their causes, defining adaptation logic, or implementing the component activation functionalities; centralised or distributed approaches can be taken for implementing the monitoring & diagnosis and the evaluation & decision modules. The main roles and functionalities of each module are presented in more detail in the following subsections.

3.1. Monitoring & Diagnosis

The monitoring & diagnosis module is concerned with obtaining run-time information on software applications (e.g. response times, throughput), exclusively on active redundant components, as well as on their execution environments (e.g. incoming workload, CPU, I/O usage). Collected information is analysed and potential ‘problem’ components identified (Section 3.5). The evaluation & decision module 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 [6] have been used for achieving this (Section 4).

3.2. Evaluation & Decision

The evaluation & decision module determines *which* redundant component(s) to activate and *when*, in order to optimise application performance. This functionality requires: i) accumulating information on 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 module at runtime and then processed and stored in formal *descriptions* (Section 3.4). Component descriptions and current monitoring data are used as input to various types of *decision policies,* or rules (Section 3.5); decision policies constitute the performance management logic of the framework.

Two main approaches can be considered for designing and implementing the evaluation & decision module, with respect to module distribution [2], [4]. 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. As explained in [2] and [4] a combined solution seems to be optimal with respect to the performance optimisation versus management overhead tradeoffs; this solution involves evaluation & decision modules with various scopes (i.e. managing single components, component groups, or the entire application), intercommunicating 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; it also allows for evaluation & decision modules at different levels to be dynamically activated or deactivated as needed, in order to minimise overheads. Work is underway to design and implement the evaluation & decision module (Section 4); this includes work on specifying component descriptions (Section 3.4) and decision policies (Section 3.5).

3.3. 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 & 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). Problems related to dynamic component replacement can still occur, as shown in software versioning, evolution, or regression testing research areas. Ways in which the solutions developed in these fields can be adopted and integrated with our framework will be analysed.

3.4. Component Descriptions

Component performance information is obtained by the monitoring module, at runtime. This information is processed (Section 3.5) and formally represented as a component *description*, or metadata, facilitating its automatic interpretation, analysis and modification. Component providers can optionally supply initial component descriptions, at deployment time. An initial description can indicate the implementation strategy used or the running context for which a component was optimised (e.g. increased workload and available CPU). It can also provide relative values for performance attributes such as delay or throughput, and/or their sensibility to environmental conditions variation (e.g. the response time for a certain method increases exponentially with the incoming workload on that method). This sort of information can be acquired from test results, estimations, or previous experience with provided components. Consequently, initial information is to be used as general guidelines rather than as absolute figures, since it was obtained in conditions (e.g. workload, application server, JVM, OS, hardware platform) that were different from the current system ones. Initial descriptions are then verified and updated at runtime with accurate monitoring information for the actual execution contexts. The evaluation & decision module can thus ‘learn’ over time about the performance characteristics of the software components and overall 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 such information.

3.5. 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 (e.g. requested performance attributes, their values and criticality) and 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: i) rules for detecting performance problems (detection rules), ii) rules for evaluating the current situation and finding optimal solutions for overcoming detected problems (adaptation rules), iii) rules for inferring new facts, or information, from existing, collected data (inference rules). Detection rules are part of the monitoring & diagnosis module, whereas adaptation and inference rules are part of the evaluation & decision module. High-level rules are used for analysing the behaviour of basic rules, in order to detect and rectify unwanted management behaviour; such behaviour can be 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). The different basic rule types are shortly discussed next.

Detection rules are triggered periodically, as new monitoring data on component performance or environmental conditions becomes available. These rules analyse current and history information, searching for patterns that would indicate the occurrence of a performance problem or the possibility for a performance optimisation (e.g. due to significant changes in the running environment). Various such patterns can be available [2]; a system manger is able to decide and dynamically select and configure the detection pattern(s) to be currently used. For example, one pattern can be set to detect when current performance values exceed certain preconfigured thresholds. Another pattern can add the constraint that thresholds need to be exceeded for a certain period of time. Patterns for detecting relevant running environment changes can also be set. When detecting a performance problem, detection rules trigger the execution of adaptation rules.

Adaptation rules are used to process the available information (i.e. current component descriptions and/or monitoring data) and take optimisation and adaptation decisions; these rules can be triggered by the monitoring module as shown above, but also periodically, for optimisation purposes, when sufficient resources are available. Adaptation and optimisation decisions are sent to the component activation module in order to be implemented into the running application.

Inference rules are triggered by new monitoring data becoming available. They are responsible for analysing existent component performance and execution environment data, inferring new information and/or modifying existing information. For example, information on which redundant component is optimal, with respect to certain performance parameter(s) and in a certain execution context, can be inferred from collected monitoring data on the available redundant components that ran in that context; this information can also be provided at component deployment time, based on off-line testing, predictions, or as guidelines based on the implementation strategy used; information inferred at runtime is then used to verify the validity of initial, or current information; non-accurate information can potentially be detected and modified in effect. Individual component descriptions are also updated as new monitoring data and inferred information becomes available during runtime. When taking optimisation decisions, adaptation rules favour the usage of inferred information, if available, rather than analyse unprocessed monitoring data each time; the inference rules are responsible for regularly performing this task instead. This approach can reduce the overhead and improve the efficiency of the adaptation process.

4. Current and Future Work

Current implementation work focuses on managing the performance of Enterprise JavaBeans (EJB) applications, running on the JBoss [5] application server. However, the way the framework is conceived and designed allows it to be extended so as to manage other quality attributes, such as reliability or robustness; the specific framework implementation can also be modified to work with other EJB application servers, or other component technologies.

The monitoring and component activation modules of the presented framework have been implemented [2], as shown in Figure 1. Part of an existing, generic EJB monitoring platform, the COMPAS [6] client, has been integrated with the framework, in order to provide some of the runtime performance monitoring functionality. COMPAS receives monitoring data from the running JBoss server, stores and manages it; received information is displayed in a graphical console in real-time, showing method-level execution times and current number of instances for each component. This facility helps human system managers to observe system performance behaviour at runtime, and possibly decide on system or management framework re-configurations; such reconfiguration decisions can also be assisted, or entirely performed by an automated evaluation & decision module [2]. As the COMPAS client is based on Java Management Extensions (JMX), the COMPAS client and JBoss server can run on separate stations.

The JBoss application server has been modified so that to be able to extract runtime information on individual method calls and on the number of instances created for each component. The information acquisition approach is based on the fact that any client request to instances of an EJB goes through the container that manages that EJB [Figure 1]. In case an EJB client makes a request, the request also goes through the client container of that EJB client. Thus, JBoss containers were modified to intercept incoming and outgoing method invocations; for any intercepted method call, information is obtained on the method invocation and response times, method caller and EJB component on which the method is being called. This allows for accurate performance information be collected and sent to the COMPAS client; in addition, call paths through the system (i.e. sequences of component method calls) can also be determined for each externally provided function, even in the presence of runtime application changes (e.g. dynamic inter-component bindings, component versioning). Call path information can be used to detect component interaction patterns and their impact on global performance characteristics. JBoss instance pools were also modified to provide information on instance creation and deletion events; component instance information can be used for example, for selecting redundant components with optimal instance pool size configurations, for different incoming workloads.

A human manager currently needs to perform evaluation and decision related tasks, assisted by the automated monitoring and diagnosis facilities provided [Figure 1]; the automatic component activation functionality is used to consequently enforce performance optimisation and adaptation decisions into the running system. Work is underway to develop inference, detection and adaptation rules (Section 3.5) for analysing stored monitoring data, identifying problems and taking appropriate adaptation decisions. The current framework implementation for the performance management logic uses the Jess rule engine [7] for storing and interpreting data (facts) and decision policies (rules).

The component activation module has been implemented and tested on the JBoss application server. The current adopted approach is based on modifying deployment descriptor configurations, at runtime [2]. The metadata indicating the EJB class and/or container configuration in the deployment descriptor are dynamically modified. This causes method implementations and/or container-provided services to accordingly change, at runtime. However, the externally visible interfaces (i.e. home, local and remote interfaces) are kept unchanged. This allows clients to remain unaware of redundant component replacement actions, as the external view of components does not change. This approach does not require any prior preparation of the

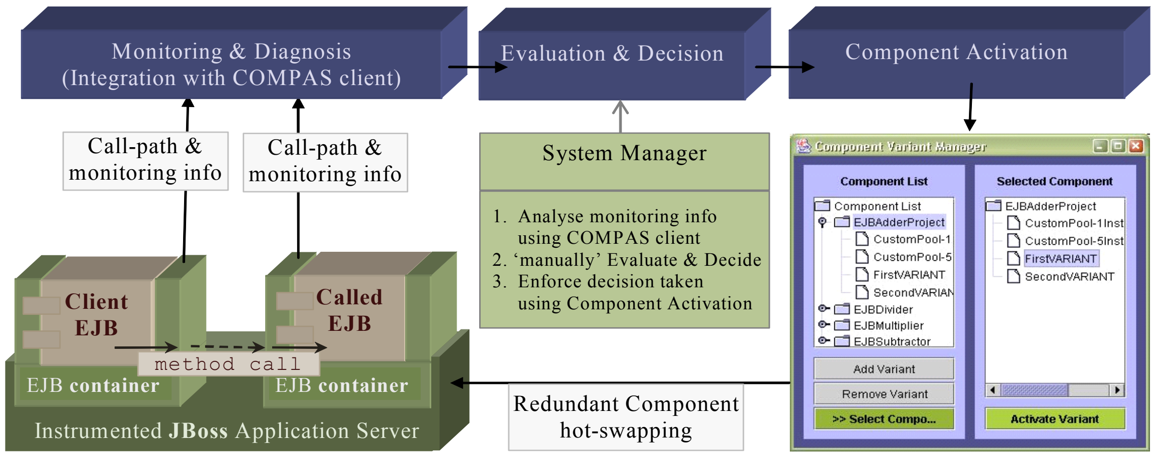

Figure 1: Performance management framework: current status

software components being swapped, or any special support from the JVM. The only support required is the hot-deployment facility of the application server. Other approaches will be considered and assessed in the future [8], [9]. A GUI is currently provided to allow system managers to operate on the component activation module, as shown in Figure 1.

5. Related Work

To the best of our knowledge, there are no similar frameworks that employ monitoring, diagnosis, and adaptation for applications based on contextual composition frameworks [1] such as EJB, at the software component level. General frameworks for self-adaptive systems are presented in [10], [11], or [12], featuring inter-related monitoring, analysis and adaptation tiers. Our framework aligns with these frameworks, while specifically targeting enterprise applications based on contextual composition middleware [1].

Component redundancy-based adaptation techniques, such as presented in [3], or [12] are similar to our proposed application adaptation approach. Our adaptation strategy differs from these approaches in that it requires component providers to supply neither accurate resource requirements or performance characteristics information for each variant, nor replacement mechanisms for each pair of redundant variants. We believe that such component-level information would be difficult to predict or calculate with sufficient accuracy, especially when complex execution platforms, such as J2EE are used. Our framework can use initial information if available, but also employs monitoring, analysis and learning processes to obtain and validate information during runtime.

6. Conclusions

Optimising the performance of enterprise software systems is becoming an increasingly difficult task, due to system complexity and dynamically changing runtime conditions. This paper presents a framework that uses component redundancy to automatically manage the performance of complex software applications and meet their quality goals. 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.

7. References

[1] C. Szyperski et al. "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, November 2002
[2] A. Diaconescu , A. Mos, J. Murphy, "Automatic Performance Management in Component Based Software Systems", International Conference on Autonomic Computing (ICAC2004), full paper (to appear), May 2004, New York, USA
[3] Daniel M. Yellin, "Competitive algorithms for the dynamic selection of component implementations", IBM Systems Journal, Vol. 42, no 1, 2003, pp. 85-97
[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] JBoss open source application server, http://www.jboss.org/index.html
[6] The COMPAS project: www.ejbperformance.org
[7] Jess – the Rule Engine for the Java^{TM} platform, http://herzberg.ca.sandia.gov/jess/index.shtml
[8] A. Orso, A. Rao, and M.J. Harrold, "A Technique for Dynamic Updating of Java Software", Proceedings of the IEEE International Conference on Software Maintenance (ICSM'02), October 2002, Montreal, Canada, pp. 649-658
[9] JFluid project, from Sun Microsystems, http://research.sun.com/projects/jfluid/index.html
[10] D. Garlan, S. Cheng, B. Schmerl, "Increasing System Dependability through Architecture-based Self-repair", in Architecting Dependable Systems, Springer-Verlag, 2003
[11] P. Oriezy et al., "An Architecture-Based Approach to Self-Adaptive Software", IEEE Intelligent Systems, Volume 14, Issue 3, May/June 1999, pp. 54-62
[12] C. Tapus, I. Chung, J. K. Hollingsworth, "Active Harmony: Towards Automated Performance Tuning", Clusters and Computational Grids for Scientific Computing, September 2002