Automatic Performance Management in Component Based Software Systems

Ada Diaconescu[1], Adrian Mos[2], John Murphy[3]
Performance Engineering Laboratory
Dublin City University
{diacones,mosa,murphyj}@eeng.dcu.ie

Abstract

A framework for automatic performance tuning of component-based enterprise applications is presented. A non-intrusive monitoring and diagnosis module is employed by an application adaptation module that automatically chooses optimal component implementations for different execution contexts. Both modules can optimize their overhead by automatically focusing on the application hot-spots, making the framework suitable for long-running systems. Currently, implementation work is targeted at J2EE systems.

1. Introduction

Large enterprise software systems increasingly depend on component middleware platforms [1] such as J2EE, CCM or .NET in order to reduce time to market and increase modularity and reusability. Such applications have a high degree of computational complexity which is mostly due to two inter-related reasons: business logic complexity and runtime platform complexity. Component containers provide comprehensive lifecycle and systemic services such as security and persistence which support the developer-written business logic and simplify development. The resulting complexity of the running system is orders of magnitude higher than the complexity of the business logic code alone, as the component containers have a significant runtime footprint. Therefore, the performance of such systems is difficult to predict and optimise. In addition, the dynamic nature of component frameworks (e.g. dynamic inter-component bindings, component versioning) as well as runtime changes of the execution context (e.g. incoming workload, available resources) can render initial optimisations obsolete, as different design and implementation strategies perform optimally in different running contexts [2][3].

These factors contribute to highly unpredictable system performance for which static performance reasoning is often unfeasible. In contrast, a loosely-controlled, self-managing, dynamic performance optimisation methodology is ideally suited for long-running systems where human intervention is less desirable [4]. This paper presents a framework that employs an adaptive approach to performance management in component-based systems by integrating low-overhead and non-intrusive monitoring with application adaptation techniques. The framework targets long-running enterprise systems and work is under way to implement it for the J2EE platform.

2. Framework Overview

The presented framework (Figure 1) comprises two main functional modules: i) *monitoring and diagnosis* and ii) *application adaptation*.

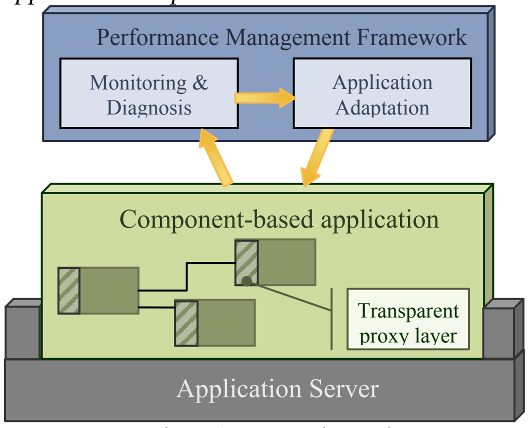


Figure 1: Framework overview

The *monitoring and diagnosis* module is responsible for acquiring run-time performance information on software components, as well as on the software application's execution environment. In addition, it analyses collected information and detects performance hot-spots. This module, part of our COMPAS framework [5][6][7], instruments software components (Enterprise JavaBeans in the J2EE case) by augmenting them with a proxy layer (Figure 1). The proxy layer is inserted in the components upon a non-intrusive introspection process that uses meta-data (e.g. deployment descriptors for EJBs) to derive essential component structural information. Monitoring data is therefore obtained at the software component level, matching the abstractions used during application development.

[1,2,3] The authors' work is funded by Enterprise Ireland Informatics Research Initiative 2001, and supported by Iona Technologies and Sun Microsystems

The *application adaptation* module is responsible for solving the performance problems signalled by the monitoring and diagnosis module. The solution proposed for providing this functionality is based on the usage of multiple, functionally equivalent component implementations, each one optimised for a different running context (e.g. incoming workload, available resources). Application adaptation is performed by selecting and activating the optimal component implementation(s) in the current running context.

In the proposed framework, the two functional modules operate in an automated, feedback-loop manner: software components and their running environment are monitored and performance problems identified; the software application is optimised and adapted to meet its high-level performance goals, in the current environment; the resulting software application is subsequently monitored and evaluated, optimisation and adaptation strategies being possibly tuned in effect. Specialised control logic is used to address the stability issues possibly introduced by the feedback loop.

In order to avoid unnecessary performance overhead, the monitoring and diagnosis module, as well as the application adaptation module can automatically adapt at runtime; the two modules can run in a standby state, with minimum induced overhead, when the managed application meets its performance requirements. They can then automatically switch to an active state, in case performance problems are being detected.

3. Adaptive Monitoring and Diagnosis

COMPAS [7] uses a proxy layer to instrument EJB applications. Specifically, each EJB is enhanced with an additional entity, the *probe*. Each probe captures performance data related to method execution and lifecycle events such as creation, activation or passivation. A *monitoring dispatcher* is used as a central management and control entity and it receives the essential monitoring data from the probes. Such data is displayed graphically in real-time charts showing the evolution of response times and stored in log files for detailed off-line analysis. Lifecycle events indicating component container activity are presented, giving insight into middleware behaviour (e.g. component creation policies, instance caches) without the need for proprietary and non-portable container-level hooks.

The monitoring dispatcher has an extensible and pluggable event-based architecture, allowing third parties to integrate proprietary event-handlers such as elaborate performance analysis tools. The integration is realised via standard interfaces which enforce the communication protocol for receiving measurements and events from the monitoring dispatcher. The graphical displays previously mentioned constitute an example of such event-handlers that use the standard extension interfaces.

In order to reduce the total monitoring overhead, the use of adaptive monitoring techniques is proposed, aiming at maintaining the minimum amount of monitoring at any moment in time while still providing enough data collection to identify performance bottlenecks. Adaptive monitoring probes can be in two main states: *active monitoring* and *passive monitoring (or stand-by monitoring)*. In the former, probes collect performance metrics from their target components and report the measurements to the monitoring dispatcher. The second state defines the light-weight monitoring capability of probes as it employs much less communication overhead. When monitoring passively, probes collect performance metrics and store them locally. In this case, measurements are not sent to the monitoring dispatcher unless a performance anomaly has been detected, or the local storage capacity (the monitoring buffer) has been depleted.

Two model-based probe management schemes for enabling diagnosis and adaptation of the monitoring process are presented: "Collaborative" (section 3.1) and "Centralised" (section 3.2). Both use dynamic models of the target EJB application.

In COMPAS terminology, a *dynamic model (or model)* consists of the monitored components (EJBs) and the dynamic relationships (*interactions*) between them. Each interaction is a set of ordered method-calls through the EJB system, corresponding to a *business scenario* such as "buy a book" or "login". The UML representation of an interaction is a sequence diagram.

Models are essential to reducing the monitoring overhead without the risk of missing performance hot-spots in the system. If no models have been obtained for an application, all components must be monitored in order to identify a potential problem. In contrast, when the interactions are known, it is sufficient to monitor top level components for each interaction [6].

3.1. Collaborative Adaptation and Diagnosis

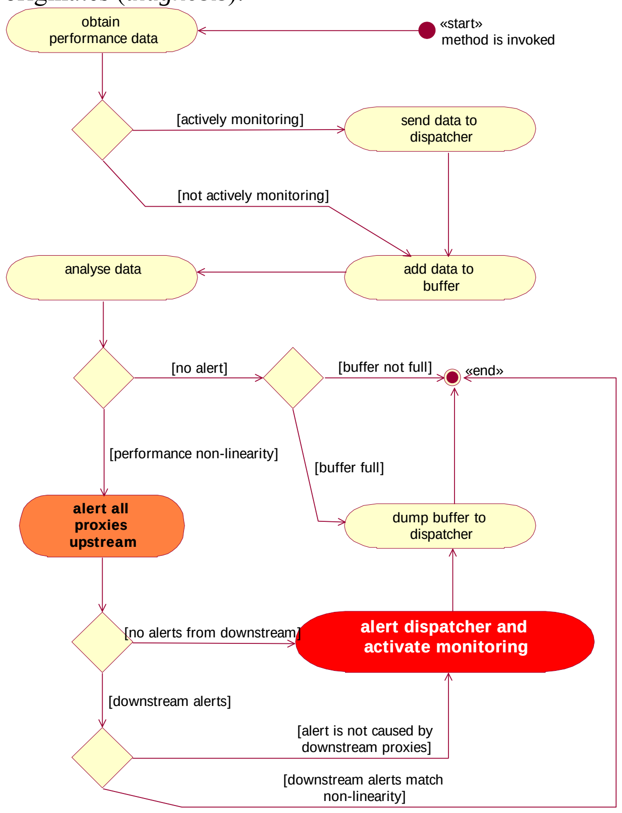
In the collaborative approach, probes have a high degree of autonomy. They collaborate among themselves to determine which component is causing particular performance degradation. Additionally, they decide which components need to be actively monitored and which components can be monitored in stand by. The monitoring dispatcher does not take any decision with regard to switching probes into stand-by or active states.

3.1.1 Probes as Independent Collaborative Agents

Each probe has knowledge about the neighbouring (*upstream* and *downstream)* probes. In relation to a Probe

X, upstream probes correspond to the EJBs which call the EJB represented by Probe X. Downstream probes are the probes corresponding to EJBs being called by the EJB represented by Probe X.

The monitoring dispatcher is responsible for sending vicinity information to all probes. This operation is performed as new interactions are discovered or recorded. The vicinity information is sent to already existing probes (corresponding to existing EJB instances) as well as to new probes as they are being created. Having the vicinity information, probes can collaboratively infer the source of performance degradation. A probe performs the following steps (Figure 2) to discover the EJB where the problem originates (*diagnosis*):



obtain performance data
«start» method is invoked
[actively monitoring]
send data to dispatcher
[not actively monitoring]
analyse data
add data to buffer
[no alert]
[buffer not full]
«end»
[performance non-linearity]
[buffer full]
alert all proxies upstream
dump buffer to dispatcher
[no alerts from downstream]
alert dispatcher and activate monitoring
[downstream alerts]
[alert is not caused by downstream proxies]
[downstream alerts match non-linearity]

Figure 2. UML Activity Diagram - Collaborative Diagnosis

1) Collects performance data when an EJB method is invoked.
2) If in active monitoring, sends performance data to dispatcher.
3) Adds performance data to its internal buffer.
4) Analyses the new buffer containing the new data.
5) If there are no performance anomalies (see section 3.3) and the buffer is full, dumps buffer to the monitoring dispatcher for storage and / or further analysis; activity ends.
6) Performance anomalies having been detected, alerts all the probes upstream. The reason is that the probes upstream can then take this notification into consideration, when deciding whether or not the performance issue originates in one of them or other probes downstream from them.
7) If other alerts from donstream have been received (by this probe), it infers that its target EJB might not contribute to the performance anomaly and activity jumps to step 8. Otherwise, the only contributor to the anomaly is its target EJB. In this case, it alerts the monitoring dispatcher of the performance problem; dumps the local buffer to the dispatcher for storage and further analysis; activity ends.
8) Since other probes downstream have exhibited performance problems, it must be decided whether they are completely responsible for the detected anomaly. The algorithm for taking this decision can be as simple as computing the numeric sum of the anomalies observed downstream and comparing it to the anomaly observed at this probe. If they are equal within an acceptable margin, it can be decided the probes downstream are the only contributors to the performance issues. The algorithm can be extended to include historical factors (section 3.3).
9) If the probes downstream are fully responsible for the performance issue, activity ends.
10) If this probe has a contribution to the performance anomaly, alerts the monitoring dispatcher and dumps its local buffer.

3.1.2 Emergent Alert Management and Generation

In the collaborative approach, probes decide collaboratively which EJBs are responsible for performance degradations. Information flow between probes is essential to the decision making process. Although numerous alerts may be raised by individual probes (in a direct correspondence to the cardinality of each interaction), only a reduced subset of the alerts are actually transmitted to the monitoring dispatcher. In this scheme, “false” alarms are automatically cancelled as soon as the real origin of the performance degradation is detected. The “real” performance hotspots thus emerge from the running system due to the collaboration between the probes.

3.2. Centralised Adaptation and Diagnosis

In the centralised scheme, probes have a smaller degree of autonomy than in the collaborative scheme. Probes send all the alerts to the monitoring dispatcher which is responsible for filtering the alerts, finding performance hot-spots and instructing probes to change their states between active and stand-by.

3.2.1 Probes as Quasi-Independent Agents

Probes are not collaborative, instead they communicate only with the monitoring dispatcher. As in the previous scheme, each probe maintains a buffer with collected performance data and has the capability to detect a performance anomaly by performing data analysis on the local buffer. Probes however do not have knowledge about their neighbours and do not receive alert notifications from downstream probes. Therefore, they do not have the capability of discerning the source of the performance issues and must report all locally observed anomalies to the monitoring dispatcher.

A probe performs the following steps for detecting a performance anomaly (see section 3.3):

1) Collects performance data when an EJB method is invoked.
2) If in active monitoring, sends performance data to dispatcher; activity ends.
3) If in stand-by monitoring, adds performance data to the internal buffer.
4) Analyses the buffer containing the new data.
5) If there are no performance anomalies and the buffer is full, dumps buffer to the monitoring dispatcher for storage and / or further analysis; activity ends.
6) If a performance anomaly has been detected alerts the monitoring dispatcher of the performance problem; dumps the local buffer to the dispatcher; activity ends.

3.2.2 Orchestrated Alert Management and Generation

Using model knowledge (e.g. obtained by an interaction recorder [5]) the monitoring dispatcher analyses each alert putting it into its interaction context. Upon receiving an alert from a probe, the dispatcher performs the following steps:

1) Parses the interaction corresponding to the probe that has generated the alert and identifies the downstream probes
2) Checks for any other alerts received from downstream probes
3) If there are no alerts from downstream, the dispatcher infers that the performance anomaly originates in the EJB corresponding to the probe that generated the alert. No other EJBs downstream have exhibited a performance problem; therefore the only contributor to the anomaly is the target EJB of this probe; sends an alert to the appropriate listeners (e.g. GUI); activates the probe that generated the alert; activity ends.
4) Since other probes downstream have exhibited performance problems, it must be decided whether they are completely responsible for the anomaly detected (see section 3.3) by this probe. The algorithm for taking this decision can be similar to the one adopted in collaborative (section 3.1.1, step 8).
5) If the probes downstream are fully responsible for the performance issue, activity ends.
6) If the alerting probe has a significant contribution to the performance degradation, sends an alert to the appropriate listeners (e.g. GUI), activates the probe.

The main difference between the collaborative and the centralised decision schemes lies in the degree of probe independence mapping to CPU and bandwidth overhead attributed to the probes and dispatcher; the advantages and disadvantages of both schemes follow the effects of this difference. In the former, more communication occurs between the probes that also use more CPU and this may not be applicable in highly distributed, low-cost deployments. On the other hand, less communication occurs between the probes and the dispatcher and less processing takes place in the dispatcher; this allows having a remote dispatcher running on a slow machine with a poor network connection, possibly over the Internet. The latter scheme is better suited for the opposite scenario where EJBs are heavily distributed across nodes and the dispatcher runs on a powerful machine connected to the application cluster via high-speed network.

3.3. Detecting Anomalies – Discussion

Detailed techniques for raising performance alerts are out of the scope of this paper. Rather, the process of narrowing down the alerts to the responsible components (the diagnosis process) is what is of interest and was described in preceding sections. This chapter serves as a discussion about possible means of identifying a potential local performance anomaly from the information observed at probe level, so that the reader can understand the feasibility of the presented approach.

Let us consider an internal data buffer present in each COMPAS probe, implementing a historical stack-like structure of collected execution times for each method in the target EJB of the probe. Each element of the method-stack represents the performance data associated with a recorded method-call event. In order to identify a performance problem, a meaningful threshold must be exceeded. The thresholds for a method can be of type:

- *Absolute value X*: at any time, the execution time *t* for the method must not exceed X ms, where X is a user-defined value.
- *Relative*: at any time, the execution time *t* for the method must not exceed the nominal execution time N of the method by more than a factor F times N, where F is a user-defined value. Nominal execution time is a loosely defined term here; it can denote the execution time of a method in a warmed-up system with a minimal workload for example.
- *Random complexity*: at any time, the execution time *t* for the method must satisfy the relationship $t \leq f(k)$; with $f : \{0, 1, 2, \ldots n-1, n\} \rightarrow Q$, where
  - k is the discrete event counter, increasing with each method call, $0 \leq k \leq n$
  - n is the size of the buffer
  - Q is the interval of acceptable performance values (e.g. execution times)
  - f is the custom "acceptable performance" function mapping the current call (with index k) to an acceptable performance value (e.g. execution time) and it can be based on the previous history of the method's performance. Developers can write this function and plug it in the alert detection framework.

The historical call data (the internal data buffer) in the probes can be used to derive more complex associations in regard to detected performance anomalies. For instance, the monitoring dispatcher (which in case of alerts receives the buffers from the probes regardless of the adaptive management model) can correlate performance anomalies from different probes and infer causality relationships. In addition, it can correlate such data with workload information in the system or database information in order to make more complex associations.

4. Application Adaptation

The goal of the application adaptation module is to overcome detected performance problems by automatically optimising component-based applications and adapting them to changes in their running environment (e.g. workload, available resources), at runtime. A solution for meeting this goal is proposed, based on the following requirements: i) different design and implementation strategies for software components are available at runtime; ii) a mechanism is provided for automatically alternating the usage of the available strategies at runtime, as needed for reaching the high-level goals of software applications. The following subsections briefly present how each of these requirements is being addressed.

4.1. Component redundancy

Component redundancy is a concept introduced for addressing the former requirement. It is defined as the presence, at runtime, of multiple component variants providing identical or equivalent functionalities 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 is optimised for a different execution context. Only one of the redundant components providing certain functionality is assigned, at any moment in time, for handling a client request for that functionality. The selected variant is referred to as the *active* component variant.

The component redundancy concept, as well as an example scenario showing the potential benefits of the component redundancy based approach, is presented in more detail in [3], [8]. Test results from the example implementation indicate that an informed alternation of redundant components, each one optimised for a different execution context, provides better overall performance than either component alone could provide.

Acquiring multiple redundant components, might seem to induce increased application development costs. However, the overall cost of both building as well as managing a system has to be considered, especially for systems with long life-spans. A modular design, in which the different strategies are placed into separate redundant components, each optimised for a different running context, is more flexible and easier to manage than a monolithic design, in which a single component is used to deal with all possible running contexts. Considering this, the overall system costs could actually be smaller when using the proposed modular approach for constructing software applications than when using the current monolithic approach. It is not required for multiple redundant components to be available at runtime or application deployment time [3]; they can be dynamically added, updated or removed from the application, as performance problems are detected or solved.

4.2. Adaptation mechanism

The adaptation mechanism addresses the second requirement of the proposed adaptation solution. Its role is to support and manage redundant components, and capitalize on their redundancy for continuously adapting and optimising applications in order to meet their performance goals (e.g. response time, throughput). The adaptation mechanism provides two main functionalities: *evaluation & decision* and *component activation*.

The *evaluation & decision* functionality determines which redundant component(s) to activate and when, in order to optimise application performance. This functionality requires: i) accumulating information on components and their running environment; ii) processing available information and determining the optimal redundant component(s), in certain contexts. The two requirements are presented next, followed by a description of the component activation functionality.

Component information is obtained by means of adaptive monitoring, at runtime (from individual probes or central monitoring dispatcher; section 3). This information is formally represented as a component *description*, allowing the adaptation mechanism to automatically interpret, analyse and update component information. 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 (i.e. delay, throughput), and/or their variation with environmental conditions (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. Initial descriptions are then updated at runtime with accurate monitoring information for the actual execution contexts. Thus it can be stated that the adaptation mechanism ‘learns’ in time about the performance characteristics of the software application it has to manage. This includes knowledge on optimal individual redundant components, as well as on optimal combinations of redundant components from different redundancy groups.

Decision policies are used to process the available information (i.e. current monitoring data and component descriptions) and take optimisation and adaptation decisions. 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 and their values) and can be added, modified or deleted at runtime.

Two main approaches can be considered for implementing the evaluation & decision functionality. One approach is to evaluate all redundancy groups involved in a certain interaction in a centralised manner and select an optimal combination of component variants (one variant from each redundancy group), so that the overall performance of the considered interaction is optimal [10]. The fact that a certain component can be used in multiple, separate interactions, with different performance requirements, also needs to be considered when selecting optimal component variants to activate. Expression optimisation techniques (e.g. relational database query optimisation methodologies [9]), or analytical analysis methods can be adopted when considering this centralised approach. Nonetheless, when considering large-scale component-based applications, global optimisations may not always be needed. Evaluating an overall application, potentially consisting of hundreds of components, whenever an individual component or a group of components does not meet performance expectations, might induce unnecessary overhead and not scale well. Therefore, a second approach is to implement the evaluation & decision functionality in a decentralised manner. In this approach, if a problem is detected at an individual component level, the problem is managed locally, by means of redundant component replacement; only components exhibiting performance problems are analysed and possibly affected by such local optimisations. This approach is potentially more scalable than the centralised one, as it avoids repeated and possibly unnecessary optimisations of the entire application. Nonetheless, exclusively concentrating on local optimisations might lead to a non-optimal global application. Also, certain problems such as deadlocks, oscillating states or chain reactions, cannot be detected or solved at an individual component level; a more high-level view is needed to detect and solve such cases.

For these reasons, both the centralised and decentralised approaches can be used, as needed. That is adaptation mechanisms with different scopes (e.g. single component, component group, or global application), can be employed, organised in a hierarchal manner. In this scenario, detected problems can be managed locally and/or signalled vertically up to the global level. A clear protocol is to be specified for allowing different adaptation mechanisms to communicate. 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. Adaptation mechanisms (at various hierarchical levels) can be dynamically activated or deactivated, in order to reduce overhead, while matching optimisation needs.

The *component activation* functionality dynamically enforces optimisation decisions into the managed application. This functionality involves activating the redundant components indicated by the evaluation & decision functionality as being optimal. A request indirection mechanism is used for implementing the component activation functionality. That is, 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.

5. Current Status and Future Work

The basic underlying monitoring infrastructure has been largely implemented. It employs a completely automated and portable installation procedure which can take an existing J2EE application and insert the proxy layer into each of its EJBs. The monitoring dispatcher can currently receive and feed performance data into real-time graphical consoles that display method-level events and execution time evolution charts. The dispatcher uses basic relative thresholds to indicate performance anomalies. In addition, lifecycle characteristics such as instance creation and deletion are displayed and all events are logged onto physical storage. At this stage, probes can be activated and deactivated manually and are designed to support additional control mechanisms and custom alert generation logic. The monitoring infrastructure has been verified to work with several applications (ranging from small sample ones to Sun Petstore or IBM Trade3) deployed on IBM WebSphere, BEA Weblogic and JBoss. Experimental results related to monitoring accuracy and overhead in COMPAS are presented in [6]. Work is underway to implement the two monitoring adaptation and diagnosis schemes discussed. Execution models are needed for the monitoring adaptation to work; two main approaches towards implementing the interaction recorder functionality were adopted and implemented.

The first approach uses non-intrusive probes to extract method execution events from the running application. It then orders the events into complete interactions by using time stamps collected by the probes. During training sessions, developers "record" the required scenarios (such as "buy a book") by going through the required steps in the system while an interaction recorder [5] obtains and stores the generated data. They can then visualise the interactions in automatically generated UML sequence diagrams. This approach has the advantage that the recorded interactions directly mirror business scenarios in the system and so the monitoring adaptation process can be based on the system design and has good chances of indicating the meaningful context for potential problems. To overcome clock synchronisation and precision issues in multi-node heterogeneous environments or even on locally deployed applications, the interaction recorder instructs the probes to induce a custom artificial delay into the target methods, thus guaranteeing the order of the received event. This approach has two main disadvantages: interactions can only be recorded during training sessions in "clean" environments where developers have total control of the system; and the process does not support multiple concurrent interactions.

A second approach solves these issues by intrusively instrumenting the application server container. It is based on the fact that all client requests to instances of an EJB go through the container that manages that EJB. Containers can thus be modified to intercept such requests and extract information on the initiator and targeted component of each request. A proof-of-concept implementation was devised for the JBoss application server; for any client-to-EJB interaction, the modified server is able to extract the client method, client instance, EJB method and EJB instance for that interaction. The main advantage of this approach over the non-intrusive approach is its capability of deterministically identifying the initiator of a certain method call, even in the presence of multiple simultaneous clients for that method. In addition, as information is being obtained dynamically (i.e. as calls are being made) runtime changes in the way components interact (e.g. due to dynamic inter-component bindings) can be detected. However, since the approach is intrusive in the sense that the application server container needs to be modified, a separate implementation is needed for each particular server considered.

The component activation functionality of the application adaptation module has been implemented and tested on the JBoss application server. When using the EJB technology, a client may only access an EJB component through the methods defined in the bean's interfaces (i.e. home and remote/local interfaces). These interfaces constitute the client's view of the bean. This implies that clients are completely unaware of the bean details behind these interfaces (e.g. method implementations provided by the bean class, or deployment descriptor configurations). A deployment descriptor XML document provides information on the deployment characteristics of a component (i.e. the interfaces and bean class to be used, container and database configurations). The adopted component activation approach is based on modifying deployment descriptor configurations, at runtime. This approach requires the target application server to support hot-deployment. To enable component adaptation, multiple redundant bean class and/or container configuration variants are made available at runtime. Dynamic modifications of the enterprise bean class and/or container configuration information in the deployment descriptor (for a certain component) causes method implementations used and/or container provided services to accordingly change (for that component) at runtime. With this approach, clients using a component remain unaware of variant replacement actions on that component, as the external view of the component does not change.

With respect to the evaluation and decision functionality of the adaptation module, an expected scenario is one in which a human manager initially performs such tasks, assisted by the automated monitoring and diagnosis facilities provided; the automatic component activation functionality is used to consequently enforce adaptation decisions. In a succeeding phase, based on repeated observations and data analysis, basic decision policies are specified; the adaptation module uses these policies to automatically suggest or take simple decisions, in common, clearly understood situations. More complex policies are incrementally added in time, enabling the adaptation module to automatically deal with more complicated, unpredictable conditions. The learning process of the adaptation module can be performed by a human manager, based on the manager's observations and expertise, or by the actual adaptation module (supervised by the human manager), based on automated data analysis and policy specification processes. As future work, basic decision policies will be tested and analysed in simple scenarios; decision complexity will be added incrementally, as more complicated scenarios are considered. In a subsequent phase, automated learning mechanisms for the adaptation module will be devised.

6. Related Work

To the best of our knowledge, there are no similar frameworks that employ adaptive monitoring and adaptation for applications based on contextual composition frameworks [1], at the component level.

General frameworks for self-adaptive systems are presented in [10] and [11], 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]. The monitoring, interaction recording and application adaptation elements presented in this paper leverage the particular characteristics (e.g. availability of component metadata, container management of components) and address particular issues (e.g. dynamic inter-component binding, highly complex execution platforms) of such platforms. In addition, a key differentiator of our proposal is the use of decentralised techniques for the monitoring and adaptation modules, in which adaptive elements interoperate, the emergent behaviour facilitating increased scalability and flexibility.

In [12], the authors focus on an adaptive monitoring infrastructure (JAMM) in a grid computing environment that can extract vital statistics such as CPU, network and memory for the running nodes in the cluster. Monitoring is started after detection of activity on some ports, by a port monitoring agent. There is no concept of software components or objects in JAMM, therefore no monitoring at method level or component level, as it is performed in COMPAS. In contrast, the COMPAS adaptation schemes do not rely on the detection of activity but rather on the detection of performance alerts. Additionally, JAMM does not use model information to optimize the monitoring overhead and it is mostly concerned with performance issues in the deployment architecture of a system (i.e. which nodes are performing badly and why) whereas COMPAS pinpoints performance issues in the software architecture of the system (i.e. which components are performing badly and in which execution context).

Component redundancy-based adaptation techniques, such as presented in [2] are similar to our proposed application adaptation approach. The main features differentiating our application adaptation module from these approaches are the lack of requirements on component providers to supply accurate initial performance information for each variant, or replacement mechanisms for each pair of redundant variants.

7. Conclusions

Performance aspects in large enterprise systems are increasingly difficult to address statically at design time in part due to complex, unpredictable underlying middleware and in part due to changing conditions such as workload or resources driven by business evolution. This paper proposes a framework for monitoring and adapting component-based applications in order to maintain acceptable performance levels. Adaptive monitoring techniques are presented that enable low-overhead continuous instrumentation of application components by automatically changing the instrumentation scope based on the current health of the system (occurrence of performance alerts). Corrective action based on evolving policies is subsequently taken by an application adaptation module which decides on the appropriate component variants to be used for the existing operating conditions.

The framework has an open architecture allowing different strategies to be used for determining the cause of performance alerts, for specifying the adaptation policies or for processing the instrumentation data. Significant parts of the framework have been implemented for widely used J2EE application servers and work is in progress to finalise a complete prototype for several environments.

8. References

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

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

[3] 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

[4] J. O. Kephart, D. M. Chess, “The Vision of Autonomic Computing”, *IEEE Computer*, January 2003

[5] A. Mos, *The COMPAS project*: www.ejbperformance.org

[6] A. Mos, J. Murphy. Performance Management in Component-Oriented Systems Using a Model Driven Architecture™ Approach. In *Proceedings of IEEE International Enterprise Distributed Object Computing Conference* EDOC, Lausanne, Switzerland, September 2002.

[7] A. Mos. A Framework for Performance Management of Component Based Distributed Applications. In the *ACM Student Research Competition Grand Finals* (second place). http://turing.acm.org/src/subpages/AdrianMos/compas.html

[8] A. Diaconescu, “A Framework for Using Component Redundancy for Self-Adapting and Self-Optimising Component-Based Enterprise Systems”, *ACM Student Research Competition* (3rd place), OOPSLA’03, Anaheim, USA, Oct 2003

[9] Don Batory, “On the Reusability of Query Optimization Algorithms”, *Information Science* 49, 1989, p. 177-202

[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*, May/June 1999.

[12] B. Tierney et al., “A Monitoring Sensor Management System for Grid Environments”, *International Symposium on High Performance Distributed Computing*, August 2000, Pittsburgh, PA