

A Self-Optimizing Container Design for Enterprise Java Beans Applications

Mircea Trofin^{*}, John Murphy^{**}
Performance Engineering Laboratory
Dublin City University
{mtrofin, murphy}@eeng.dcu.ie

Abstract

Contextual component frameworks, such as Enterprise Java Beans (EJB), facilitate the development of easily evolvable and modifiable enterprise applications. Support for dynamic re-composition, such as instance-level contextual composition and runtime binding are the base upon which such features are offered. Applications can be built out of third-party components and deployed on third-party platforms. At the same time, this black box nature of third-party components and platforms diminishes the effectiveness of traditional performance analysis methods. This raises the role played by runtime optimizations in addressing performance issues of such systems.

We propose a self-optimizing application server design. The optimization is driven by the discovery of inter-component communication patterns and the application of container refactorings.

1. Introduction

As software systems' complexity increases, so does the need for solution reusing. Component-based software development is a natural answer to this need. Companies increasingly rely on component-oriented technologies, such as Enterprise Java Beans (EJB) [1], and Commercial Off-The-Shelf (COTS) components, in order to build large scale applications, reduce system development costs and capitalize on third party expertise.

While componentization addresses a number of issues, such as software maintainability and reusability, it does not necessarily address system performance issues. Modern component technologies complicate such issues even further, as system composition occurs late in the development cycle, sometimes even at runtime, thus thwarting any attempts for early performance reasoning.

Quite often, in order to deal with performance issues, system developers choose to redesign the system, changing its composition, in order to reduce the impact of the underlying platform (the application server). This approach cannot be applied in a COTS scenario, since components are already developed and are meant to be used "as is".

Rather than redesigning the application, we propose that the underlying platform, notably the component containers, adapt to the application needs. The adaptation process is automatic, and is driven by the acquisition of knowledge about call patterns between a system's constituent components.

We will present a container adaptation framework that aims at improving the performance of EJB applications without changing the component set. While we do not believe that our method will solve all performance issues, it will alleviate those introduced by the application server.

2. Background

2.1. EJB components

We consider software components, or, in short, components, as "units of composition with contractually specified interfaces and explicit context dependencies only" [2]. The EJB component framework is targeted at developing enterprise applications. It distinguishes two classes of components, based on how they receive messages: synchronous and asynchronous (message-driven). Furthermore, component instances might have state. Depending on the durability of that state, there are stateless session beans (no durability), stateful session beans (temporary durability) and entity beans (persistent durability). This offers a convenient development model, as business entities map naturally to various component types (business entities to entity beans, business logic to session beans).

^{***} The authors' work is funded by Enterprise Ireland Informatics Research Initiative 2002.

2.2. EJB systems

EJB systems are built by deploying software components (the enterprise beans) on an application server. Some or all the beans, as well as the application server, can be produced by different organizations, such as in the case of a COTS scenario.

At deployment time, components simply get placed in the application server environment, but no binding takes place. The only type of binding supported in EJB is between component instances and at runtime. In essence, using only information provided by components (including their deployment descriptor), there is no guaranteed way to understand the call paths between them until runtime. The most that can be known is type dependencies, but they do not indicate when connections are created, what components are connected (since many components could implement the same business interface), and how components are used. This kind of binding allows for dynamically re-composable systems, that is, systems in which components can be changed at runtime without shutting the whole system down. This not only supports COTS, but supports systems designed for unpredictable changes in the requirement set, which need post-deployment system evolution. Many current internet-oriented applications fit in this category.

A realistic scenario is that in which the application server and some of the components are provided by various third party organizations, some components are built in-house, and, once the system is started, functionality is altered by adding or changing some components. The organization offering such a system capitalizes on the existing component set and can expediently deliver solutions to changing customer requirements.

2.3. Performance in EJB systems

The performance of a system can be reasoned upon at design time, be dealt with at deployment time, or be left for optimizations at runtime. Generally speaking, the earlier performance is taken into account, the better the results [3].

To reason about performance early in the design process implies that some overall system knowledge is available at that time. For a component-based system, this includes knowledge about components' interconnections, patterns of communication, and underlying platform. In an EJB system, neither are completely known until runtime, unless the system is developed "from scratch", by the same organization, and with a clear vision about its evolution. The latter are all conditions that do not mandate for an EJB approach in the first place, or, for that matter, for any approach that involves the use of a component framework which supports dynamic re-composition. In fact, current efforts [9] that employ traditional performance analysis methods, such as Layered Queuing Networks (LQN), on EJB systems, make the assumption that system structure, as well as component internals, are known.

Deployment time optimizations suffer from the same limitations, which are that system structure is unknown. All that is known are type dependencies, which could lead to some optimizations (such as co-locating dependent type implementations/components). Furthermore, if the system utilization patterns change, deployment time optimizations might fail to deliver the intended performance improvement.

In traditional systems, addressing performance early in the development cycle was possible, and had a strong optimizing effect on the resulting system. EJB systems, due to their dynamic and unpredictably evolving nature, may benefit less from early-design approaches and more from runtime optimizations.

2.4. Area of optimization

The support for dynamic structure appears to be a performance-impacting factor. Experimental results [4] show how, by sacrificing modularity and reusability, better performance can be obtained. The best performing design used stateless session beans only, and handled each client request without leaving the boundaries of one such bean. An alternative, modular design employed a set of collaborating beans, which modeled various aspects of the application, such as business entities (modeled as entity beans) and business logic (modeled as session beans). Even when optimized (using local interfaces and an improved virtual machine), this design exhibited half of the throughput of the stateless session beans-only design.

The challenge, then, is that not only the performance of EJB systems cannot be easily analysed using traditional approaches, but that designing along the true intent of the framework inherently affects performance. A conflict exists, then, between having well-performing systems, and having systems that are modular, easily extensible and/or built from third-party components.

2.5. Current approaches and related work

To our knowledge, there are no current solutions to the problem described in the above section. Most of the time, the system is redesigned by choosing other components and/or by designing along various performance-enhancing patterns. The problem with this approach is twofold. Changing the component set is time consuming in a COTS scenario, since it requires trials of various designs. Second, patterns target groups of collaborating components, and might address a particular style of collaboration. If one component is taken out of that context, its “pattern-aware” design might not provide any enhancements. If a new component is added, and enters collaboration with such collaborating components, the pattern assumptions might be broken. This complicates system evolution.

Once a system is built, some performance-improving techniques are possible, but they might be costly: upgrade hardware, use clustering, or expand such cluster.

A flow-oriented component system optimization approach has been described in [5]. The difference between our approach and the one presented there is that our optimizations deal with high-level, application server specific issues, while the work described in [5] deals with lower level, machine code optimizations. Such optimizations will most probably miss high-level performance causes, such as those introduced by the application server. We believe the two methods can be used together, first optimizing the application server, then applying machine-level optimizations.

Application-level optimization strategies have been proposed [6]. The authors propose a framework in which functionally-equivalent components are alternatively employed during runtime in order to optimize the whole application. This approach is orthogonal to ours, and can be employed conjointly.

Design-time refactorings have been proposed [7]. The problem with design time approaches, as previously presented, is that they can impact other quality attributes of a solution, such as reusability.

3. Proposed approach

We believe that a solution to the performance problem described in 2.4 lies with optimizing an application by adapting the software platform it runs on, i.e. the container. This type of solution might not be as powerful as an early-on performance-oriented design approach, but it allows for transparent optimizations, rather than design-disruptive ones. In other words, we believe it is possible to have systems designed for unpredictable evolution, perform like specialized, un-evolvable systems.

We intend to devise a framework for adapting containers in order to transform the execution model of an application. The key observation is that components are concepts observed at design and system composition time. While a runtime environment contract exists between components and the underlying platform, this contract does not mandate a unique realization. The current strategy is to determine component runtime support at deployment time (when component containers are determined and stubs are compiled). However, since the actual usage of such a component is unknown, the containers have to be generic with respect to the possible utilizations that would occur at runtime. For instance, transactional and security contexts have to be verified. The alternative we are proposing alters the runtime support as usage patterns are revealed.

The EJB specification [1] does not clearly mark the difference between containers and application servers. For our purposes, the application server is an entity providing support for enterprise services, such as naming, transactional isolation, or security, while the container is an entity wrapping each component and mediating that component’s runtime requirements, such as communication with other components, lifecycle, and runtime execution context properties (e.g. transactional or security)

Our framework (see Figure 1) consists of an Adaptable Container, a Monitor, a Flow Discoverer and a Refactoring Engine (the greyed elements in the figure). The Monitor collects runtime information about component instance interactions. The Flow Discoverer renders its information into a call graph structure, representing components, calls, and order of calls (similar to UML collaborations). The Refactoring Engine takes the advice of refactoring policies and, based on information provided by the Flow Discoverer, modifies containers (i.e. either changes existing containers or creates new ones). Note that it is not necessary that one container be mapped to one component, before or after a refactoring.

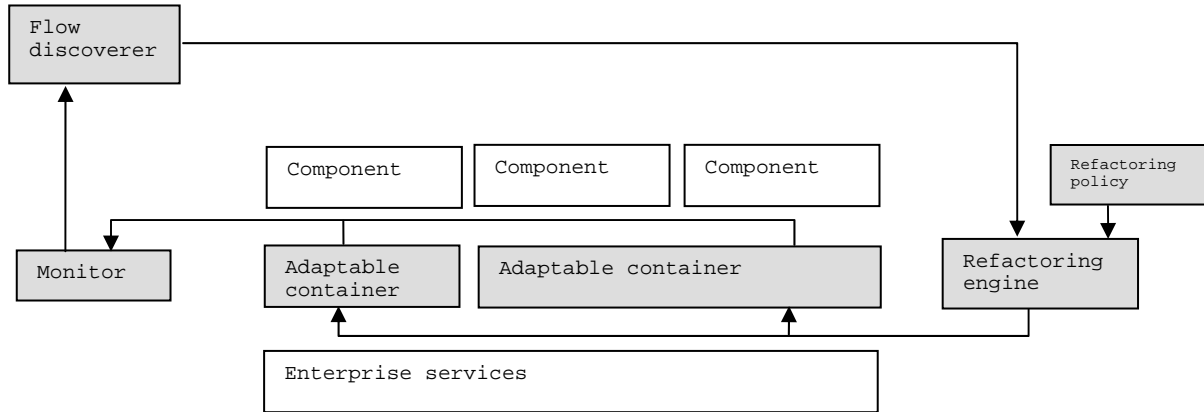


Figure 1: self-optimizing EJB platform blueprint

Our work will be focused on the Refactoring Engine, the definition of refactoring policies, and the design of an Adaptable Container. A non-intrusive monitoring and flow discovery solution has been previously proposed [8]; some application servers might offer intrusive implementations of such services as well.

The solution is meant to be delivered as a set of services of the application server. It will periodically attempt to discover if new communication patterns emerged. If so, it will attempt to refactor containers. If the system composition changes, previous refactorings can be discarded, as new communication patterns might emerge.

The precise details of the framework are yet to be defined. In the following sections we will illustrate how container refactorings can be used to optimize an e-commerce system. Based on the experience that will be gained by implementing such refactorings, more general scenarios will be analysed.

3.1. E-commerce case

It is known [10], [4], that the best performance, especially in terms of scalability, is usually offered by stateless session bean implementations. This might not be always true, however, it does hold for applications which need to handle a large workload, but do not receive requests from the same client often. Most e-commerce applications fit this description. Usually, such applications are deployed on clustered application servers, with all components available on all elements of the cluster. This means that any system client request can be handled completely on the same machine. We can assume that interactions between components are local (i.e. through local interfaces); even in this case, [4] indicates that a stateless session beans-only implementation outperforms the finer grained one.

For such applications, we will call the *ideal design* (with respect to performance) the one which implements functionality using only stateless session beans, and in such a way so that any system request is handled without leaving the boundaries of one bean.

Our attempt at this stage is to enable application servers automatically transform an (e-commerce) application built out of a mix of types of beans into a functionally identical application, which executes as if it were built according to the ideal design.

If a developer were to redesign a component-based system into a stateless beans-only system, they would probably maintain the functional decomposition of the initial system, and simply transform the components into functionally-equivalent classes. Then, they would design session beans, which would use such classes, and programmatically manage execution context issues and lifecycle of the class instances. Since the developer knows the order in which functionality is invoked, issues related to data acquisition and persistence can be optimally handled, i.e. data retrieved only when needed, and maintained in memory only as long as needed (i.e. for the duration of the overall session). Such an implementation would not use caching, but that in turn would allow for better scalability, since component instances would not be tied to any particular machine in the cluster. [4] confirms that, even without caching, a stateless implementation scales better. We aim to perform this transformation automatically.

The key for transforming the initial design into a stateless one is capturing the flow of calls between components that result as consequence of a client of the system making a request. Using this information, the realization of client requests can be optimized. If the system component set is changed, the optimizations are discarded and the optimization process resumed.

We assume that inter-component call patterns have some stability in time (i.e. component interactions are not chaotic) and that the component set is updated in time intervals much greater than the time required for the optimization process to transform the platform.

3.2. Flow optimization

As the flow discovery mechanism identifies call paths through the system, the refactoring engine detects frequently executed call paths and decides to optimize them. To do that, it creates a single custom container for all the components in the call path, which offers runtime support throughout the call path execution (this is the refactoring). Each time the same path is taken, it would execution will occur optimally inside this container.

We propose the concept of a *system interaction container*. This container offers optimized execution support for one particular call path (i.e. system interaction). The call path itself is executed (in the interaction container) as a stateless session bean. Calls outside that call path are handled as calls outside a regular component. The code of components in the call path is left unchanged; it's just the runtime support for their instances that changes.

The key is to ensure that, by employing such a container, the semantics of the execution are maintained, and that, if alternate paths are taken (alternate with respect to the optimized path), they are performed correctly. The first point refers to ensuring that the component deployment descriptor requirements for an execution context are satisfied. That means that, among others, transaction isolation properties and security constraints have to be maintained. The second point refers to the fact that, during execution, it is not guaranteed that what appears to be a known path through the system is indeed that one. Therefore, it is quite possible that the execution take a yet unknown route. We have to ensure that the execution of that route is not affected by our optimizations.

Optimization is achieved from two sources. First, a full interaction with the system is transformed, through the system interaction container, into a stateless session bean, thus allowing for better scalability. Second, the system interaction container tailors services offered for inter-component calls part of that interaction so that no unnecessary context checks are performed. This is possible, since the properties of the execution context at any point in the call path can be known, based on the deployment descriptors of the previous components and the order in which methods were called.

3.2.1 Example

Figure 2 presents a call path through the system. The circles indicate points of interception of calls by component containers. At these points, each container performs lifecycle and context handling services (e.g. transactions, security). The arrows represent calls, and the labels - the order in which the calls are performed. For simplicity, we will refer to methods in terms of their order. In this example, all the methods have the same transaction and security requirements; assume they are “*required*” for transaction and unspecified for security. In terms of types of beans, A and B are stateless session beans, while C and D are entity beans. Let E be a bean that might be called by B, but that has never happened before, and thus it is unknown to our system.

To optimize this call path, we generate the system interaction container. Since the call path is known, all container behaviour can be pre-compiled, thus eliminating performance impact of context verifications, for example. This is done while the system is running.

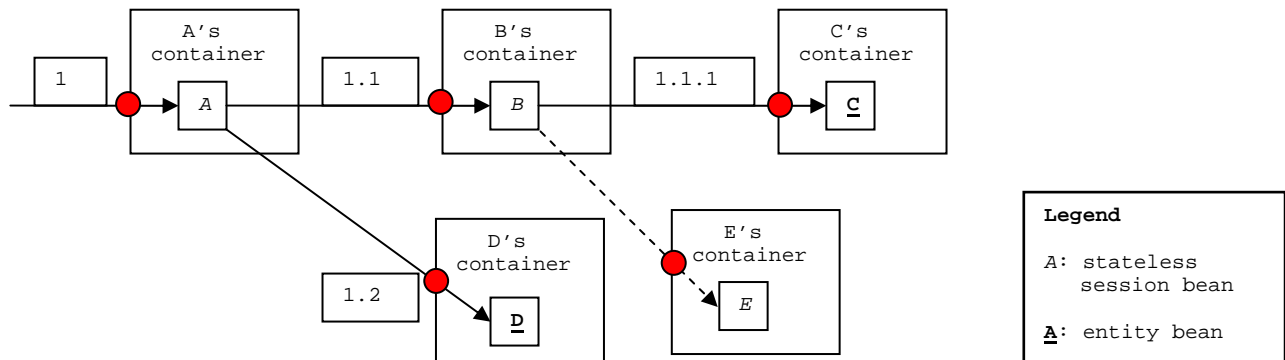


Figure 2: un-optimized system client request handling

Refer to Figure 3: when the server receives a call from an external client to method 1, it passes it to the specialized container. Since we intend to emulate a stateless session bean, stateless session bean lifecycle services and context services (security and transaction isolation-related) have to be performed. The specialized container sets itself as the naming provider; therefore, when method 1 intends to obtain a reference to B or D, the container will provide it with a specialized stub (the grey square in the figure). This stub is specialized in handling B's methods, given that they are called from method 1 of A. Since it is known ahead of time how the context is shaped at this moment, only the needed services are performed. For our case (calling method 1.1), the only cost incurred here is the cost of an indirection. Since the specialized container is still the naming provider, B's attempts to locate C will be handled by this container, and the optimizations can be enforced as outlined before.

If, at any point, one of the methods attempts to locate components other than the ones handled by the interaction container, then un-optimized stubs will be provided by the specialized container, and the un-optimized execution of the new call path will continue as usual. This new path can be learned and optimized, if needed. In our example, if the path from B to E is taken, then a stub to E's "regular" container is given, and the execution follows an un-optimized path.

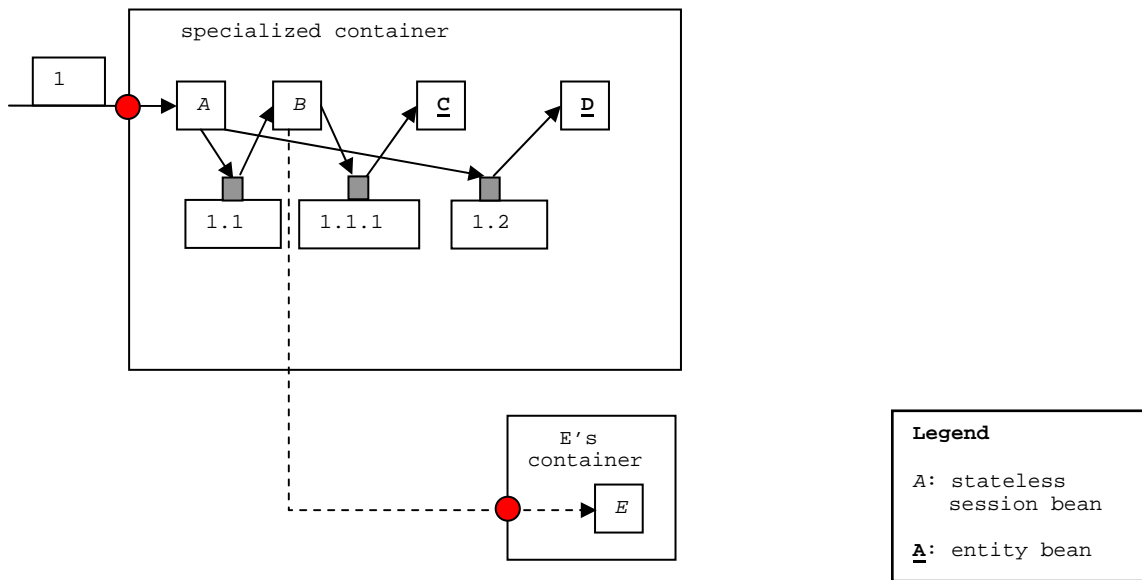


Figure 3: optimized refactoring

In terms of handling lifecycle, the specialized container will align the lifelines of the contained bean instances to the lifeline of the overall session. For entity beans, it will simply load their state when it is needed first, which can be pre-determined from the call graph. It will persist it according to transaction boundaries, which are pre-determined from the call graph as well. When the flow ends, the object is discarded. The entity bean will, in fact, execute like a developer-made data access object (DAO). Note that the bean's lifecycle methods will still be called; since their execution is data management-related, and that aspect is optimized, their execution should not impact performance.

It is quite possible that, when a system is optimized, the same component be executed within different containers, as it might be involved in different interactions. Moreover, it is possible that the same instance of an entity bean be accessed concurrently as part of parallel flows, and as part of different custom containers. Since transactional contexts are ensured and are equivalent to the ones in an un-optimized scenario, concurrency will be handled by the transaction manager (typically, the database server).

If the system is evolved, the custom containers are discarded, and the adaptation process restarted. Discarding the custom containers can be done in an optimized way, for example, if components are upgraded, it could be assumed that interactions did not change. This may hold in the case of a bug fix, for example.

Optimization in the presented scenario comes from two sources: redundant executions of context validity checks are removed, and lifecycle of stateful beans is aligned to the execution of a stateless interaction, thus the whole interaction executing like a stateless session bean.

4. Current status and future work

Currently, we are at an early stage of our research. Empirical results [4] indicate that an e-commerce application would benefit from our proposed refactoring. The first step is to create a refactoring tool targeted at performing the presented

optimization. This tool will create system interaction containers for a set of beans, given a developer-prepared description of flows. We will apply the tool on a test application and measure performance improvements.

The long term goal is to identify a set of container refactoring primitives. Various application optimization strategies can be implemented in terms of these primitives. Application servers could offer an interface through which container refactorings can be performed in terms of such primitives. Then, expert knowledge about optimizing particular interconnection patterns or particular applications can be provided, independent of the application server.

We intend to analyse how the container refactoring method could be further applied in a full J2EE environment, to optimize interactions between web applications and EJB applications, for example.

5. Conclusions

We have presented an approach for runtime optimization of EJB applications. Our proposal consists of providing self-optimizing capabilities to the software platform such applications run on. We believe that, due to the dynamically evolving nature of EJB systems, and the black box nature of components, runtime optimization approaches have a more effective impact on performance than traditional, early design stage methods have.

We target the adaptation of the runtime support offered to components. We believe that there is a decoupling between design time concepts and runtime concepts that allow for such adaptations. The adaptations consist of container refactorings, and are driven by recognizing patterns in inter-component interactions. We have described an e-commerce scenario in which, by refactoring the realization of interactions with the system into calls to stateless session beans, we expect that better performance and scalability be obtained. The optimization procedure consists of creating system interaction containers, which offer specialized runtime support for components that participate in realizing individual system interactions. Based on previously-observed calls between participating components, optimization is achieved by removing un-needed lifecycle and context-related services. As a benefit, optimized interactions can be executed by any machine in a cluster, since they are stateless. When all system flows are thus optimized, the whole system becomes stateless.

We intend to implement a refactoring engine capable of performing such transformations. This engine will be used to optimize sample applications, thus evaluating the effectiveness of our method. The experience gained implementing this engine will constitute a start point for identifying a general container refactoring solution targeted at the EJB framework. Extending the method to other container types, such as web containers, will be further analysed.

It is expected that our approach would offer a solution to the conflict between having component systems that are evolvable and functionally-extensible through the support for dynamic re-componentization, and obtaining high performance from such systems.

6. References

- [1] Sun Microsystems. "Java 2 Platform Enterprise Edition", <http://java.sun.com/j2ee/>
- [2] C. Szyperski with D. Gruntz and S. Murer. "Component Software. Beyond Object-Oriented Programming". Second edition. Addison-Wesley, 2002
- [3] C.U. Smith, L.G. Williams, "Performance and Scalability of Distributed Software Architectures: An SPE Approach", *Parallel and Distributed Systems*, Vol. 13, No.2, February 2002
- [4] E. Cecchet, J. Marguerite, W. Zwaenepoel. "Performance and scalability of EJB applications". In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* November 2002, Seattle, WA
- [5] A. Gal, P.H. Fröhlich, M. Franz. "An Efficient Execution Model for Dynamically Reconfigurable Component Software". In *Seventh International Workshop on Component-Oriented Programming (WCOP 2002) of the 16th European Conference on Object-Oriented Programming*. June 2002, Malaga, Spain
- [6] A. Diaconescu, J. Murphy, "A Framework for Using Component Redundancy for Self-Optimising and Self-Healing Component Based Systems", To appear in *ICES 2003 Workshop on Software Architectures for Dependable Systems*, May 2003, Portland, OR
- [7] Brett McLaughlin. "Building Java Enterprise Applications Volume I: Architecture". *O'Reilly*, 2002
- [8] A. Mos, J. Murphy. "Understanding Performance Issues in Component-Oriented Distributed Applications: The COMPAS Framework". In *Seventh International Workshop on Component-Oriented Programming (WCOP 2002) of the 16th European Conference on Object-Oriented Programming*. June 2002, Malaga, Spain
- [9] T. K. Liu, S. Kumaran, Z. Luo. "Layered Queuing Models for Enterprise JavaBean Applications". In *Proceedings of 5th International Enterprise Distributed Object Computing Conference (EDOC)*, Seattle, WA, September 2001
- [10] BEA Systems, Inc. "Scaling EJB Applications". In *BEA WebLogic Enterprise 5.1 Documentation*