

# On Performance of Enterprise JavaBeans

Radek Pospíšil, Marek Procházka, Vladimír Mencl<sup>1</sup>

## Abstract

*Enterprise JavaBeans (EJB) is a new-sprung technology for Java-based distributed software components. During the one year of EJB existence, several implementations have been developed, mainly by the leading corporations of the software industry. The goal of the paper is to identify a set of criteria, which would help developers to evaluate specific EJB implementations. These criteria especially cover a particular implementation's compliance to the EJB specification and performance of the implementation. Experience gained from a project<sup>2</sup> evaluating several EJB implementations is used in this paper.*

## 1 Overview

With the expansion of multitier application architectures, the enterprise software providers focus their attention on the request brokers, transaction monitors, and other middleware services that facilitate building the multitier applications. In response to this demand, Sun Microsystems introduced the *Enterprise JavaBeans* (EJB) standard, a component framework that provides services for transactions, security and persistence in a distributed multitier environment. As its name indicates, the standard is based on the Java programming language and runtime environment.

Since the introduction of the EJB standard in March 1998, a number of companies provided their own implementations. Typically, these implementations are offered as a part of the existing application servers of the respective companies. They differ in many aspects, from the level of compliance to the EJB standard to the class of performance delivered.

The goal of this paper is to identify a set of criteria, which would help to evaluate specific EJB implementations. An extensive part of the paper is dedicated to a thorough evaluation of the individual compliance points as defined by the EJB standard. This is necessary especially because both the EJB standard and the EJB implementations are still under development and thus the compliance points are often not observed.

Apart from the compliance with the EJB standard, the paper also proposes criteria for the performance evaluation of the EJB implementations. The performance tests measure costs of elementary operations (such as an invocation of an empty method), and evaluate characteristics of the respective implementation by observing dependence of both the costs of the elementary operations and the overall behavior on the various factors involved. The characteristics evaluated include scalability, robustness, and overhead caused by the use of various features of the transactional subsystem.

---

<sup>1</sup> Mgr. Radek Pospíšil, Mgr. Marek Procházka, Mgr. Vladimír Mencl, Department of Software Engineering, Faculty of Mathematics and Physics, Malostranské náměstí 25, 118 00 Prague 1, <http://nenya.ms.mff.cuni.cz/thegroup/>, e-mail: {pospisil,prochazka,mencl}@nenya.ms.mff.cuni.cz

<sup>2</sup> EJB Comparison Project, carried out by the Distributed Systems Research Group at the Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic, and sponsored by MLC Systeme GmbH, Ratingen, Germany

The criteria proposed in this paper are based on a project evaluating several EJB server implementations. The results collected in the project have served as a proof-of-the-concept for the criteria described here.

## 2 Enterprise JavaBeans

Sun Microsystems' *Enterprise JavaBeans* (EJB) framework is a component architecture for development of distributed, object-oriented business applications in the Java programming language. The EJB specification [1] [2] defines a standard for the EJB deployment environments, *EJB servers*, and for reusable components that are executed in these environments, *enterprise beans* or beans for short.

A bean implements application-dependent business logic, and is typically structured into *EJB objects*, or bean objects for short. These objects are then associated with the EJB bean. A *container* provides a deployment environment that wraps the beans during their lifecycle; every bean lives within a container. The container provides services that the bean can use, namely transactions, security and persistence. The EJB specification does not state the way these services are to be implemented; it only specifies the interfaces of the EJB container through which the services are made accessible to the bean objects.

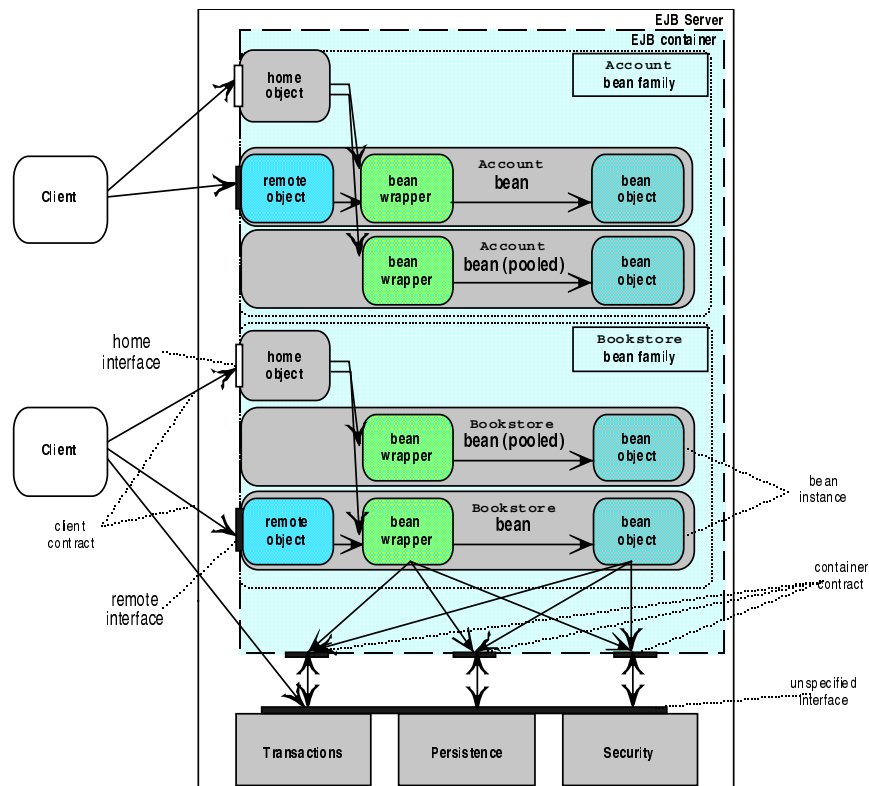


Fig 1. Structure of an EJB server

In general, the protocols that characterize the dialog between a bean object and its container are referred to as the *bean contract*. Furthermore, every bean has a *deployment descriptor*, a description of the bean's characteristics and the bean's usage of the services provided by the container. The container also supports a set of protocols, termed the *client contract*, that characterizes the dialog between the bean clients and the bean. The client contract defines two interfaces that the bean client uses to communicate with the bean: the *home interface* and the *remote interface*. Both interfaces are created at deployment time by special tools supplied by the EJB server provider. The remote interface reflects the functionality of the bean, also called the business methods of the bean. The home interface supports methods for creation and

removal of EJB objects, as well as methods for querying the population of the EJB objects, termed *finder methods*.

To use a business method of a bean, the client has to obtain a reference to the bean's home interface using the *Java Naming and Directory Interface* (JNDI) [3]. Using this reference, the client can create or find a bean object, and obtain a reference to a stub implementing the bean's remote interface. The stub then delegates method calls to the corresponding EJB object.

An EJB server transparently manages the population of the beans residing in the main memory. When the population of the bean objects inside a container grows beyond a certain limit, the container stores some of the not-recently-used bean objects in a secondary memory (the objects are *passivated*). Whenever a method call targets a passivated bean object, the object is brought back into the main memory by the container (the object is *activated*).

There are two types of enterprise beans – *session beans* and *entity beans*. Session beans reflect short-lived objects, which exist on behalf of a single client and do not represent directly any shared and/or persistent data in a database. Compared to an entity bean, a session bean differs in the following:

- Its identity is not exposed to the client; therefore, the bean's home interface cannot provide any finder methods.
- It cannot be reentrant (reentrancy, guarded by the EJB container, is allowed only if enabled in the deployment descriptor; only an entity bean can be reentrant).
- It can be involved in at most one transaction at a time.
- It is transaction-aware; it can be notified about the stage of the current transaction.
- It cannot be persistent.

A session bean has to be serializable to allow the EJB server to passivate/activate the bean's instance to/from a secondary storage. Depending upon its conversational state, a session bean can be *stateful* or *stateless*.

An entity bean, on the other hand, typically represents persistent data, usually stored in a database. An entity bean is transactional, allows shared access from multiple clients, and can be long-lived (living as long as the data in the database, and thus surviving crashes of the EJB server).

Every entity bean is uniquely identified by its primary key (analogous to a primary key in a database). Therefore, the bean's home interface can provide the finder methods. An entity bean supports two types of persistence: *container managed* (CMP) and *bean managed* (BMP) persistence. Transaction support is more complex than with the session beans, because of the support for persistence.

In Enterprise JavaBeans, only distributed flat transactions are supported. The distribution mechanism makes it possible for an application to update data in multiple databases during a single transaction. It is not the concern of an application programmer to ensure transactional semantics, these are guaranteed by the underlying mechanisms. The beans execute the database updates using the standard JDBC API. Behind the scenes, the EJB server registers the database connections as members of a transaction. When the transaction commits, the EJB server and all the databases involved use the two-phase commit protocol to ensure atomic update. Multiple EJB servers can participate in a transaction; data can be modified by enterprise beans installed in different EJB servers in a single transaction.

Every client method invocation on an enterprise bean object is interposed by the container. The interposition allows for delegating the transaction management responsibilities to the container. The container handles transactions according to the *transaction attributes* that are

specified in the corresponding bean's deployment descriptor. The transaction attributes can be associated with the entire bean and apply to all methods, or just with an individual method. The EJB specification version 1.0 [1] supports the `TX_BEAN_MANAGED`, `TX_NOT_SUPPORTED`, `TX_SUPPORTS`, `TX_REQUIRES`, `TX_REQUIRES_NEW`, and `TX_MANDATORY` attributes.

### 3 Compliance with the specification

Most of the EJB servers available are still under development. Typically, the vendors aim at implementing all the features required by the EJB 1.0 specification. However, this goal is often not reached, partially because of misinterpretations of the specification (which is not clear enough in some places), partially because of the implementation errors (which are likely to be fixed in future releases of the server). In this section, criteria for compliance with the EJB specification are described and discussed.

#### 3.1 Essential functionality

By the essential functionality, we understand the functionality that is the key to the use of an EJB server. It includes facilities for creating connections from a client, bean instance creation and removal, accessing the beans' meta-data, and of course invoking the beans' methods. In particular, the following properties have to be evaluated:

- The ability to obtain a reference to the `EJBHome` interface via JNDI.
- The home interface object implementation, i.e. bean creation, removal.
- The functionality of the `EJBMetaData`, `EJBObject`, and `Handle` interfaces.

#### 3.2 Session beans

As stated in the EJB specification, an EJB server has to support the session bean abstraction (unlike the entity beans that are optional). A session bean represents a “function”, being by default temporary (*short-lived*), non-persistent, and thus of limited transactional capabilities (compared to the entity beans). The focus should be put especially on the following points:

- The “one transaction at a time” restriction.
- The implementation of the `SessionContext` interface.
- The functionality of the `SessionSynchronization` interface.
- The correct lifecycle implementation of both the stateful and the stateless session beans.

#### 3.3 Entity beans

By default, an entity bean represents data that are persistent (long-lived) and transactional. In the EJB 1.0 specification, support for entity beans is optional. To test the support for entity beans (if implemented), the following properties should be evaluated:

- The implementation of the `EntityContext` interface.
- Handling of the concurrent access from multiple clients.
- Support for the `findByPrimaryKey()` methods, used for both the container and the bean managed persistence.
- The entity bean lifecycle.
- The support for reentrancy.

#### 3.4 Container-manager persistence

With the *container-managed persistence* (CMP), the state management is delegated to the EJB container (an entity bean does not invoke any database calls to store its state). In the bean's deployment descriptor, all attributes are declared to be managed by the EJB container.

Unlike the *bean-managed persistence* (BMP), CMP is fully automatic; but, due to the limited expressive power of the deployment descriptors, CMP is not as flexible.

In the compliance tests, the support of the EJB container for container managed persistence should be verified.

### 3.5 Security

The EJB security scheme is based on the security model of the Java programming language. Identity of the calling clients is represented by the `java.security.Identity` object. Two security-related methods are provided by the `EJBContext` interface, `getCallerIdentity` and `isCallerInRole`; functionality of these methods as well as the implementation of the *Access Control Lists* should be examined.

### 3.6 Transactions

In a distributed system, a transaction is one of the basic abstractions used to provide transparent concurrency, reliability, and fault tolerance. In an EJB implementation, the following features have to be thoroughly examined:

- The basic functionality, e.g. creating, starting, and finishing a transaction, setting a transaction timeout, marking a transaction as `RollbackOnly`.
- The support for an update of multiple databases.
- The support for transactions distributed onto multiple EJB servers.
- The interoperability with non-Java clients and servers (access to existing enterprise applications, such as transaction monitors, via bridges).
- The support for bean-managed transaction demarcation.
- The support for container-managed transaction demarcation; in particular, the support for the transaction attributes and the handling of the transaction lifecycle.
- The support for transaction isolation levels; a transaction isolation level declared in a deployment descriptor has to be propagated via JDBC connections to the underlying databases.

Note that most of these features are essential for the use of the Enterprise JavaBeans technology in enterprise applications.

### 3.7 Interoperability

From the developer's point of view, interoperability is one of the most important issues. To ensure at least a minimal level of interoperability, the following issues should be examined:

- Platform independence – whether a client can be a standalone Java application with no restrictions to either a specific platform or a Java language implementation.
- Support for transactions that encompass multiple EJB servers of different implementations. Many EJB servers use a proprietary RMI implementation. Since the original Java RMI specification did not say how the RMI implementation should transfer the transactional context, interoperability problems are likely to occur.
- Portability among different EJB servers – assess the difficulty of deploying a bean developed for one EJB server in another one.

## 4 Performance criteria for EJB

In this section, the proposed tests for comparing EJB servers with respect to scalability and performance of the distribution mechanism and transactions are described. In addition, the robustness of the EJB servers is considered.

In the first group of the proposed tests, focus is put on the distribution mechanism. Next, the overall scalability of the EJB server is evaluated, together with the robustness of the server. Finally, the overhead caused by the use of the EJB services is measured, with an emphasis on transactions.

## 4.1 Distribution mechanism

The EJB specification defines the Java RMI as the primary distribution mechanism. It also allows to employ other mechanisms, such as CORBA IIOP. Usually, however, the current EJB servers support RMI only. As the EJB server vendors often choose to use a proprietary implementation of the Java Virtual Machine (and RMI), scalability and performance of the RMI implementation used may vary. Because RMI relies on the Java serialization, performance of the Java serialization has to be taken into account.

### 4.1.1 Serialization

Serialization is the principal point of the RMI performance. Observations have proven that serialization of large method arguments constitutes a significant part of RMI overhead. Therefore, we suggest to measure how the speed of serialization depends on the type and size of the parameters and the way they are passed.

We identified the following measurements as characterizing the serialization performance:

- Measurement of the time spent serializing a single value of a primitive type depending on the way it is stored. Three methods should be tested: passing the value as a primitive type, as a `java.lang` object representing the primitive type, and as a class with a single attribute of the primitive type.
- Measurement of the time spent serializing a fixed amount of primitive type values depending on the way used to pass the values. Three methods should be tested: passing the values as a single array, passing the values as a class with the respective number of attributes, and passing the values as simple parameters of a method.
- Measurement of the time spent serializing a fixed amount of object references depending on the target of the reference. Three types of reference targets should be evaluated – a null reference, a set of reference to different instances of a class, and a set of references to a single instance of a class. The class used should be a simple class with no attributes; this way, the overhead of serializing the instances themselves is minimized.
- Measurement of the cost of the primitive types serialization. Two approaches to this test can be taken; either the time necessary to serialize a single primitive type is measured for all the primitive types found in Java, i.e. `byte`, `char`, `short`, `int`, `long`, `float`, `double`, and `void`, or the overhead of using different data types for passing parameters of a constant total size is measured. Because Java RMI limits the maximum size of the direct method parameters to be 256 bytes, this test is also limited. The test is carried out for all primitive types in Java, with the invoked method taking the appropriate number of parameters of the respective type to make the total size of the parameters constant.

### 4.1.2 RMI

Currently, several implementations of Java RMI exist. For the purposes of evaluating EJB, the most important is Sun JDK 1.1 [4]. As the future EJB servers will probably use Java 2, Sun JDK 1.2 [5] should be considered as well. Furthermore, several EJB servers rely on proprietary RMI implementations, which should also be taken into account.

Although all of the implementations conform to the Java RMI specification, they can have different scalability and performance properties.

To evaluate an RMI implementation, the benchmark tests described below should be used. All the tests perform RMI calls on a set of objects. The time measured is the time necessary to carry out the RMI call only. The target object used for the test should have one method only, with a `void` return type and no arguments. No transaction context should be involved.

To assess the influence of the order of method invocations on the cost of an RMI call, the following tests should be carried out:

- Object instances should be iterated through in an ascending order, with  $N$  RMI calls carried out on each of the objects.
- Object instances should be iterated through in a descending order, with  $N$  RMI calls carried out on each of the objects.
- Object instances should be iterated through in an ascending order  $N$  times, with a single RMI call carried out in each pass on each of the objects.
- Object instances should be iterated through in a descending order  $N$  times, a single RMI call carried out in each pass on each of the objects.
- Object instances should be iterated through in a random order  $N$  times, with a single RMI call carried out in each pass on each of the objects.

The purpose of these tests is twofold, to measure the performance of the RMI mechanism, and to ensure that the performance of the RMI implementation does not depend on the call pattern in a way that would distort the measurement results for other benchmarks.

## 4.2 Scalability

Scalability is one of the most important performance aspects. Here, it is first discussed in general, and then specifically with respect to transactions.

### 4.2.1 Number of clients

The ability of an EJB based application server to serve connections from several clients at the same time is essential. Dependence of the method invocation time on the number of clients connected to the called bean should be examined. The test should start a number of clients, each client executing an infinite loop invoking a method on the bean. At the client side, the time necessary to carry out the method invocation should be measured. In addition, the test should check whether each client gets its share of the access to the bean.

### 4.2.2 Number of bean instances

As data are collected during the application lifetime, the number of object instances tends to grow. The impact of the increasing numbers of objects present in the system on the cost of the elementary operations should be examined to prevent future surprises. As proposed here, the test should create objects in a loop; with the increasing number of objects present, the time of creating a new object instance, the time of performing the first and the subsequent method invocations on the instance, and the time of removing an instance.

### 4.2.3 Transactional context

Typically, reliable applications rely on transactions whenever accessing data. Thus, it is important to know the overhead incurred by the use of transactions. A number of tests dealing with this issue will be discussed in section 4.4; here, only scalability is discussed. The dependence of the *commit* operation time on the size of the transaction context is examined, the size measured in terms of involved objects. The test creates a fixed number of objects in

the EJB server. Then, transactions are carried out with increasing number of objects involved in each.

This test should also be performed with multiple EJB servers involved in the transactions, to assess the cost of using transactions in a distributed environment. The objects should be distributed uniformly across the participating EJB servers.

Furthermore, the test can be performed with a large number of EJB servers involved, checking for a potential dependence of the transactional overhead on the number of EJB servers involved. Unfortunately, the computational resources necessary for performing this test on a reasonable scale are expensive, making the test difficult to carry out.

#### **4.2.4 Number of objects**

In the previous test, the dependence of the *commit* operation time on the number of objects involved in a transaction was examined. However, objects residing on an EJB server are liable to cause an overhead even when they do not participate in the respective transaction. Therefore, this test measures the dependence of the *commit* operation time on the number of objects present in the EJB server. The measurements are carried out as objects are being created on the EJB server. This test is an addition to the test described in section 4.2.2, where the dependence of the object manipulation operation times on the number of objects present in the system was examined.

#### **4.2.5 Number of active transactions**

Dependence of the cost of the *commit* operation on the number of simultaneously open transactions should also be measured. It is necessary to have one thread for each open transaction in this test. Unfortunately, our experience indicates that the limiting factors of the Java Virtual Machine thread system exhibit themselves sooner than the potential dependencies of the EJB server performance on the number of transactions.

### **4.3 Robustness**

By robustness, we understand the ability to perform reliably under demanding conditions. Robustness can be evaluated with respect to many issues. While it is not practical to design tests that would guarantee the suitability of an EJB server for a particular application, we can identify several scales that reveal the most common weaknesses of the server.

#### **4.3.1 Number of bean instances**

The response of an EJB server to an attempt to continuously create EJB instances should be examined. Instances are created until the EJB server in question crashes or reports that it is not possible to create more instances. The test should be performed for both the session beans and the entity beans.

For the entity beans, instances can be passivated and moved to the persistent store during the test. Thus, the result does not represent the maximum number of instances resident in memory. Session bean instances, on the other hand, cannot be passivated, and the result represents the number of instances resident in memory.

#### **4.3.2 Number of transactions**

The maximum number of simultaneously open transactions should be measured. The test should create threads in a loop, each thread opening a transaction and performing a method invocation in the context of the transaction to get the EJB server involved in the transaction.

Note that the results of this test do not necessarily indicate a limit within the EJB server, as the virtual memory of the client's Java Virtual Machine is usually exhausted before the limit of the EJB server is reached.

### 4.3.3 Maximal request size

The maximum size of a request should be determined. Requests with increasing size should be sent to the server until either an error is reported, or the server crashes. To generate an arbitrary request size, an array of that size is passed as the method parameter.

### 4.3.4 Number of clients

An important issue might be measuring the robustness of an EJB server with respect to the number of clients, be it the number of clients connected or the number of clients actively performing method invocations. Unfortunately, the EJB server vendors often limit the number of clients allowed to connect to the EJB server as a part of their licensing conditions, which prohibits carrying out the test.

## 4.4 Transactions

A transaction can add an overhead to the method invocation. Several performance aspects of using transactions can be measured. In section 4.2, the focus was on the behavior of the EJB server when a large number of either objects or transactions is used. In this section, the impact of using various transactional attributes on the method invocation time is discussed.

### 4.4.1 Overhead caused by transactional attributes

With the transactional attributes applied to bean methods, the overhead of transactional context creation, suspension, and/or transfer can occur. Tests should be carried out to measure the exact influence of the transactional context manipulation operations. For each of the transactional attributes, the cost of using the attribute should be determined by measuring the time necessary to complete an invocation of an empty method augmented with the respective transactional attribute. This measurement should be carried out twice, once within a client transactional context, and once without one.

To illustrate the approach, consider a method associated with the `TX_NOT_SUPPORTED` transactional attribute. When called inside and outside a client transaction, the difference in the invocation times represents the overhead of the transaction suspension taking place in the first case. Similarly, if a method associated with the `TX_REQUIRED` transactional attribute is called inside and outside a client transaction, the overhead of the transaction creation is measured. If a method associated with the `TX_REQUIRES_NEW` transactional attribute is called inside a client transaction, the overhead of a client transaction suspension combined with a container-managed transaction creation and completion is measured. Also, note that in the tests mentioned above, when a method is called within a client transaction context, the context has to be transferred from the client to the server.

### 4.4.2 Overhead of bean managed transactions

For bean managed transactions, the time necessary to perform a bean-demarcated transaction should be measured. The measurements have to be done from the client's point of view, and should be taken for all the possible scenarios of distributing a transaction into multiple method invocations:

- Invocation of a single method that does not use bean-managed transactions: The method is augmented with the `TX_BEAN_MANAGED` attribute, but no transaction context is created, and no transaction primitive is called.

- A method which gets a user transaction context and starts and completes a transaction.
- A method which starts and completes a transaction, obtaining the transaction context only once during a preceding method invocation.
- A method that uses a transaction retained on a bean. The transaction is started during one method invocation, is retained after the method completes, and is finished in another method. All bean methods invoked within the transaction's duration are invoked in the context of this transaction.

All these tests should be carried out both inside and outside a client transaction. This should provide results for all possible combinations of transactional attributes, even those that cause unnecessary overhead, as they might be accidentally used by a programmer unaware of a bean's implementation.

All the tests described in this section are performed on methods with both an empty signature and an empty body. A comparison of the overhead of manipulating a transactional context with the overhead of transferring a parameter and a return value can be performed. The results show that overhead of transferring a small number of parameters is insignificant compared to the overhead of manipulating the transactional context.

## 5 Miscellanea

Together with the criteria characterizing performance and standard compliance of a particular EJB server, additional issues, also important to the application developers, should be evaluated:

- Technical requirements (supported hardware platforms, operating systems, Java implementations).
- Developer support (GUI tools, debuggers, logging, quality of documentation).
- Vendor data (a brief company profile, market specialization, flagship products, pricing policy).

## 6 Summary

The Enterprise JavaBeans technology has been introduced in this paper, and criteria suitable for evaluating the EJB server implementations have been proposed. Issues covered by the criteria are compliance with the specification and various benchmark tests that cover the performance, scalability, and robustness characteristics of the distribution mechanism and the transactional support provided by the EJB server implementations.

The criteria described in the paper are based on our research evaluating several EJB server implementations. The criteria have been applied to several existing EJB servers, yielding important compliance and benchmark results. Although the results themselves are not included in this paper, we take them as a proof-of-the-concept for the criteria.

## 7 References

- [1] Vlada Matena, Mark Hapner: Enterprise JavaBeans 1.0 Specification, Sun Microsystems Inc., March 1998, <ftp://ftp.javasoft.com/docs/ejb/ejb.10.ps>
- [2] Vlada Matena, Mark Hapner: Enterprise JavaBeans 1.1 Specification, Public Release, Sun Microsystems Inc., August 10, 1999, <http://java.sun.com/products/ejb/docs.html>

- [3] Java Naming and Directory Interface 1.2 Specification, Sun Microsystems Inc., July 14, 1999, <ftp://ftp.javasoft.com/docs/jndi/1.2/jndi.ps>
- [4] Java Development Kit (JDK) 1.1, Sun Microsystems Inc., <http://java.sun.com/products/jdk/1.1/docs/index.html>
- [5] Java Development Kit (JDK) 1.2, Sun Microsystems Inc., <http://java.sun.com/products/jdk/1.2/docs/index.html>
- [6] Susan Cheung: Java Transaction Service 0.95 Specification, Sun Microsystems Inc., March 1, 1999, <http://java.sun.com/products/jts/index.html>
- [7] Object Management Group: Object Transaction Service, December 1997, <ftp://www.omg.org/pub/docs/formal/97-12-17.ps>
- [8] Susan Cheung, Vlada Matena: Java Transaction API 1.01 Specification, Sun Microsystems Inc., April 29, 1999, <http://java.sun.com/products/jta/index.html>
- [9] Rohit Carg: Enterprise JavaBeans to CORBA Mapping 1.0, Sun Microsystems Inc., March 1998, <ftp://ftp.javasoft.com/docs/ejb/ejb-corba.10.ps>
- [10] Sanjeev Krishnan: Enterprise JavaBeans to CORBA Mapping 1.1, Sun Microsystems Inc., August 11, 1999, <http://java.sun.com/products/ejb/docs.html>
- [11] Satoshi Hirano Yoshiji Yasu, Hirotaka Igarashi: Performance Evaluation of Popular Distributed Object Technologies for Java, HORB Open and Electrotechnical Laboratory, <http://www.horb.org/eval-team/acm98/>
- [12] ECperf Benchmark Specification, Java Specification Request JSR-000004, March 26, 1999, [http://java.sun.com/aboutJava/communityprocess/jsr/jsr\\_004\\_ecperf.html](http://java.sun.com/aboutJava/communityprocess/jsr/jsr_004_ecperf.html)
- [13] CORBA Comparison Project, Final Report, Distributed Systems Research Group, Charles University, Prague, June 1998, <http://nenya.ms.mff.cuni.cz/thegroup/>