

Enterprise JavaBeans Benchmarking¹

Marek Procházka, Petr Tůma, Radek Pospíšil

*Charles University
Faculty of Mathematics and Physics
Department of Software Engineering
Czech Republic
{prochazka, tuma, pospisil}@nenya.ms.mff.cuni.cz*

Abstract

Enterprise JavaBeans (EJB) is an emerging standard of a support platform for distributed multitier applications written in Java. Since the introduction of the standard in 1998, a number of EJB implementations became available, differing in many aspects from the level of standard compliance to the delivered performance. This prompted a demand for evaluation and comparison of the EJB implementations. In response, we have defined an EJB test suite and carried out a comprehensive evaluation of several EJB implementations as a part of the EJB Comparison Project [2]. The purpose of this paper is to share the experience gathered during the evaluation process, and to outline some of the pitfalls that make the evaluation difficult.

Keywords: Enterprise JavaBeans, Java, benchmarking, distributed application performance

¹ This paper is based on the EJB Comparison Project, carried out by the Distributed Systems Research Group at the Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic, for MLC Systeme GmbH, Ratingen, Germany

1 Introduction

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. 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.

Often, it is necessary to evaluate the properties of the EJB implementations, in order to assess their suitability for a specific application, or for a similar decision related to the EJB technology adoption. To provide a framework for such an evaluation, the Distributed Systems Research Group of Charles University in Prague, started the EJB Comparison Project [2]. Commenced in 1999, the project aims at devising a set of criteria for evaluating specific EJB implementations in terms of standard compliance and delivered performance, and at comparing a selected set of EJB implementations using these criteria. This paper provides a brief overview of the EJB implementation evaluation carried out during the project, and outlines the experience thus gathered and some of the pitfalls of the EJB implementation evaluation. The public version of EJB Comparison report [3] is attached to the paper.

2 Enterprise JavaBeans Overview

The Enterprise JavaBeans (EJB) framework, provided by Sun Microsystems, is a component architecture intended for development of distributed, object-oriented business applications in the Java programming language. The EJB specification [9][10] defines interfaces and behavior of the EJB deployment environments, EJB servers, and of the reusable components that execute in these environments, enterprise beans or beans for short.

An enterprise bean implements application-dependent business logic, and is typically structured into bean objects (Figure 1). 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 contained beans 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 container through which the services are made available to the bean objects. 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 client accesses a bean through 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 a particular bean objects, as well as methods for querying the population of the EJB objects, termed finder methods.

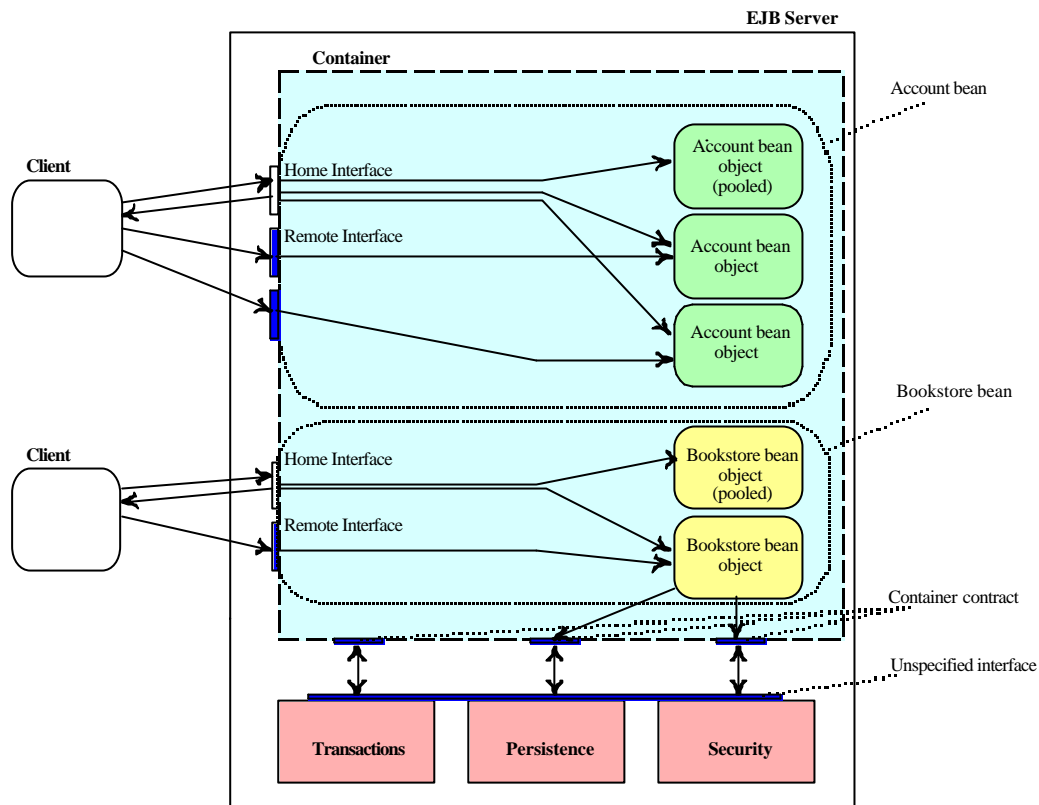


Figure1. EJB architecture

To use a business method of a bean, the client has to obtain a reference of the bean's home interface using the Java Naming and Directory Interface (JNDI). 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 are short-lived objects that exist on behalf of a single client and do not represent directly any shared and/or persistent data in a database. 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.

EJB supports distributed flat transactions defined in JTS [5]. Every client method invocation on a bean is supervised by the bean's container, which makes it possible to manage the transactions according to the transaction attributes that are specified in the corresponding bean's deployment descriptor. Multiple EJB servers or databases can be involved in a single transaction.

The EJB container has to provide security service that control accesses of clients on beans instances, manage user identities and propagate security context along with calls between EJB servers. EJB security is not based on any standard.

3 Test Suite Overview

The test suite used in the EJB Comparison Project consists of two major parts: the compliance tests, aimed at verifying if and how much the EJB implementation subject to testing complies with the EJB specification, and the performance benchmarks, aimed at evaluating the performance delivered by the implementation. The performance benchmarks focus in the basic performance, scalability, and robustness.

Although the EJB comparison project was based on testing of four servers, only the results for the NetDynamics EJB server are published. The results are attached to this report.

3.1 Compliance Testing

One of the principal tasks is to ascertain an EJB implementation's compliance with the EJB specification. Among other things, this is important because the EJB standard is relatively young and rapidly evolving. Consequently, the EJB implementations are new and tend to be incomplete, which makes the question to be answered "how much does it comply with the standard" rather than "does it comply with the standard".

In our project, the compliance tests cover all parts of the EJB specification. In all, the compliance tests include more than 40 individual programs aimed at testing specific points of the specification, or the overall behavior characterized by a major part of the specification. In the tests, we have distinguished an error-free, a partially defective, or a totally defective behavior of the particular implementation. The tests are listed and precisely described in [3]. The results of the compliance tests give a thorough picture of the functionality offered by the EJB implementation, and a useful hint on the quality of implementation.

3.1.1 Essential functionality

The EJB server has to provide services necessary for the EJB framework as described in the EJB specification [9]. These tests have to prove that a server and necessary services (e.g., JNDI server, database connectivity) can be started up. In the next step, basic functionality of both entity and session beans (e.g. lookup of home interfaces, creating/removing of bean instances) is verified.

Most servers passed these tests without problems.

3.1.2 Session and Entity beans

Two parts of the compliance test suite verify the implementation of the session and the entity beans. They are focused on bean lifecycle, a bean specific behavior (e.g., synchronization, reentrancy) and bean context implementation.

All servers tested had problems passing these particular tests, mostly because of pure or incomplete implementation.

3.1.3 Requested services

Every EJB server has to implement three essential services: persistence, security and transaction. The persistence service is used by the entity beans for container-managed persistence (CMP) only. This service provides an implementation-independent-view of the entity bean's state. The security service is used by the session and the entity beans. This service provides basic security functionality such as access restriction and identity propagation. The most important services is the transaction service, based

on JTS [5]. This service is used to add transaction behavior to the session and the entity beans. Usage of the transaction service have effect on the lifecycle state diagram of the session beans (e.g., the bean code can be notified of a transaction state). The entity beans represent data stored in a database, and therefore the transactions are propagated to the underlying database.

All servers tested had problems passing these particular tests due to poor or incomplete implementation.

3.2 Performance Testing

In this section, we outline the approach taken to benchmark the EJB implementations. The benchmarks were executed in different setups, such as with the clients and the server running on the same machine, or the clients being connected to the server via a dedicated network segment. The focus is first on the distribution mechanism, then the transaction management is examined.

3.2.1 Distribution mechanism

The EJB specification defines Java RMI [18] as the primary distribution mechanism, but allows the use of other mechanisms, such as CORBA IIOP [14][19]. As EJB server vendors often choose to use a proprietary implementation of the Java Virtual Machine and/or RMI, scalability and performance of RMI implementations used may vary. Because RMI relies on the Java serialization, performance of the Java serialization has to be taken into account.

There are two groups of the benchmark tests. The first group provides the performance view of the serialization part, i.e., how fast different parameter patterns are passed. The second group provides the performance view of the distribution mechanism, i.e., how fast an invocation is performed.

To measure the speed of Java Serialization, different parameter patterns and types have to be taken into account. There are three areas that have to be measured - primitive data types, overhead of objects and overhead of parameters in a method signature.

These measurements give us several results: a) the average speed of the operation (time of the serialization plus the invocation); b) the scale used to compare different servers (i.e., if a server is fast in serialization of primitive types relative to being slow in serialization of a structure of objects).

To measure the speed of the distribution mechanism, different invocation patterns are used. Because the overhead of Java Serialization is predictable (due to previous measurements), measurements of invocation overhead are done on methods without parameters. The invocation pattern is given by a order of method invocation on a bean instances (e.g., call n-times method on the first instance, then call n-times method on the second instance,...; or the first method call on the first instance,...the first method call on the last instance, the second method call on the first instance,...). The transaction service is not involved in these tests.

4 Scalability

Previous tests give the results based on one client and one bean instance configuration. Using these results, the scalability of large system (more than one client and/or bean instance) is not predictable. To evaluate the scalability of the EJB implementation, the following tests were evaluated:

- measurement of the dependency of the method invocation time on the number of clients connected to the called bean. The test simulated a pool of clients executing requests on a particular bean object. The time necessary to carry out the method invocation was measured. In addition, the test examined the distribution of the bean object response to clients;

- measurement of the dependency of the method invocation time on the number of bean objects. With the increasing number of objects present, we measured the time of creating a new object instance, the time of performing the first and all subsequent method invocations on the instance, and the time of removing an instance. Since some bean instances can be temporarily stored in an external memory or database, policies for pooling, passivating and activating bean instances had to be taken into account.

4.1 Robustness

By robustness, we understand the ability to perform reliably under demanding conditions. Although it is not possible to design tests that would verify the suitability of an EJB server for a particular application, we can identify several benchmarking criteria that are likely to reveal the most common weaknesses of the server. The following tests were provided:

- the limit of the number of bean objects. Instances were created until the EJB server crashed or reported that it was not possible to create more instances. The test was performed for both the session beans and the entity beans. Note that for entity beans, instances can be passivated and moved to the persistent store during the test. Thus, the results did not represent the maximum number of instances resident in memory. Session bean instances, on the other hand, cannot be passivated, and results represent the number of instances resident in memory;
- the limit of the request size. Requests with increasing size were sent to the server until either an error was reported, or the server crashed;
- the limit of the number of clients connected or the number of clients actively performing method invocations.

4.2 Transactions

In a distributed system, a transaction is one of the basic abstractions used to provide transparent concurrency, reliability, and fault tolerance. In the following sections, the tests evaluating the performance, scalability, and robustness of the transaction support are presented.

4.2.1 Performance

With the transactional attributes applied to bean methods, the transactional context creation, suspension, and transfer can add an overhead to each invocation. Tests were 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 was determined by measuring the time necessary to complete an invocation of an empty method augmented with the respective transactional attribute. The measurements were carried out twice, once within and once outside a client transactional context, to evaluate the impact of the clients' transaction context transfer.

For bean-managed transactions, we have measured invocations of a method that did not use bean-managed transactions, a method that always created and finished a bean-managed transaction, and a method that run within a retaining bean-managed transaction.

4.2.2 Scalability

Typically, a large number of transactions can be opened concurrently on an EJB server. To evaluate the scalability of the server with respect to transactions, the following tests were provided:

- measurement of the dependency of the commit operation time on the number of simultaneously

opened transactions. Unfortunately, it is necessary to have one thread for each open transaction in this test. Our experience indicates that the limits of the Java Virtual Machine thread system exhibit themselves sooner than the potential dependencies of the EJB server performance on the number of transactions.

- measurement of the dependency of the commit operation time on the size of the transaction context; i.e., the dependency between the number of involved bean instances and the time to commit were measured. Moreover, the involvement of multiple EJB servers and databases was taken into account.
- measurement of the dependency of the commit operation time on the number of objects present in the EJB server. This test was an addition to the test described in section 4.4.1, where the dependency of the object's method invocation time on the number of objects present in the system was examined.

4.2.3 Robustness

We have examined the maximum number of active transactions on a particular server. Note that the results of this test did not necessarily indicate a limit within the EJB server, as the virtual memory of the client's Java Virtual Machine was usually exhausted before the limit of the EJB server with respect to transactions was reached. Also using a pool of nodes generating new transactions cannot satisfactorily solve this problem because of the overhead caused by the network.

5 Evaluation Experience

Apart from providing interesting data on the standard compliance and performance of the individual EJB implementations, the evaluation results also provided some useful generalizations concerning both the state of the EJB technology and the methodology of EJB benchmarking.

5.1 On EJB Technology

The most important generalization of the evaluation results is the indication of immaturity of both the EJB standard and its current implementations in several areas. As far as the EJB standard is concerned, problems occur mostly in the area of server and client code portability. Our experience indicates the code requires rewriting partially due to differences between Java 1.1 and Java 2, which can be used as the underlying platform for the EJB implementation, and partially due to ambiguities in the EJB standard. They are, however, being fixed very fast by the EJB standardization body (the original EJB 1.0 specification [9] was published in March 1998, the EJB 1.1 version [10] was published in August 1999, the EJB 2.0 version will likely be available by the end of 2000 or in the first half of 2001). We also have to consider the Java Transaction API [6] and Java Transaction Service [5] specifications, which are tightly related to Enterprise JavaBeans.

The EJB implementations themselves carry all the marks of a recently developed system. The level of standard compliance can vary significantly, with some implementations meeting only one half of the requirements of the standard, and even relatively stable implementations failing the requirements in one or two points, e.g., in the areas of retained transactions or transaction isolation levels. Again, this is being remedied very fast, quite often a new version of an EJB implementation occurs before the old one can be thoroughly evaluated.

The individual EJB implementations also differ in the bean deployment mechanism and the text format of the bean deployment descriptor. This might be partially because the text format of the

deployment descriptor was not defined in the EJB 1.0 standard. In EJB 1.1 the deployment descriptor is defined as an XML file.

In terms of performance, the EJB implementations can differ as much as two orders of magnitude in the times it takes to finish basic tasks, such as invoking a method or committing a transaction. Overall, however, the performance of the EJB implementations came as a pleasant surprise, with the faster implementations being close to the performance of, e.g., CORBA ORBs and services implemented in C++.

Quite often, the EJB implementations exhibit performance anomalies that are difficult to explain. These include situations where a method invocation takes longer time to complete when the EJB server handles a smaller number of objects, or situations where invoking a method within a transaction takes longer time than outside a transaction for some servers, and shorter time for others. Hence, it is important to cover all aspects of the EJB implementation behavior with specific benchmarks, as the results are sometimes counterintuitive and trying to deduce behavior in one case from behavior in another can fail.

5.2 On EJB Benchmarking

The process of benchmarking an EJB implementation is hindered by several features of the Java environment. Perhaps the most notable problem here is the disruption of benchmarking results by the garbage collector runs, which needs to be amortized through a measurement of larger tasks or tasks with high repetition counts. Also, the Java Virtual Machine consumes quite a lot of resources, which tends to cause resource sharing delays and swapping, which is also detrimental to the measurement precision.

Another source of disruption is the Java timer accessible through the `java.lang.System.currentTimeMillis()`, whose granularity can be as high as tens of milliseconds, i.e., not enough for a measurement of small tasks that get finished quickly. Note that measuring several repetitions of the same task to improve the precision is not always possible, as certain tasks cannot be carried out repeatedly without an extra operation in between the repetition steps. A solution we have adopted is implementing a more precise timer as a native method through JNI.

The environment itself imposes limits that can prevent carrying out certain types of benchmarks. One of those limits is related to the number of arguments within a Java method signature, which is limited to less than 256. This limits, e.g., measuring the dependency of the method invocation time on the number of its arguments. A similar situation arises when a large number of threads or a large number of transactions needs to be created. This limits, e.g., measuring the scalability with respect to the number of EJB clients served, as it is not possible to run a large number of clients on a small number of machines, and obtaining enough machines to evaluate the EJB implementation scalability for thousands of clients tends to be expensive.

Also, nontrivial is the issue of configuration of the particular EJB implementation and the underlying platform. Things to be configured include default values for timeouts and pooling mechanisms, the settings of the bean passivation and activation criteria within the EJB container, the JVM virtual memory settings, all of which can significantly influence results of some of the benchmarks.

5.3 Toward Standardization

Naturally, it would be desirable to standardize the compliance tests and the benchmarking suite, so that the evaluation results can be published independently for the individual EJB implementations and

then compared. There are, however, several problems which prevent defining such a standard for EJB.

First and foremost, the EJB standard and its implementation do not seem to be stable enough yet to warrant creation of a standardized benchmarking suite. As mentioned earlier, both the client and the server code are not easily portable. Together with documentation that is often lacking, this leads to situations where it is necessary to disassemble the EJB implementation to find out how certain features are implemented and how the benchmarking suite should use them. Also, due to omissions in the EJB standard, some of the compliance tests turn out to be "which of the possible interpretations of the standard does it implement" tests rather than "does it comply with the standard" tests. This will hopefully change as the EJB standardization progresses and the individual EJB implementation vendors improve their adherence to the specification.

It should also be noted that most EJB servers are integrated with a graphical development environment, which does not lend itself easily to a batch-style processing of a large number of benchmarks. Having an unwieldy development environment which makes it difficult to import the benchmarks is another obstacle to the idea of providing a standardized benchmark suite.

Some of the more interesting results were obtained by administering the stub and skeleton code generated by the EJB implementation with timestamp operations. This made it possible to evaluate more precisely the individual phases of the method invocation chain, eliminating, e.g., the network overhead. Unfortunately, patching the code specific to the EJB implementation can hardly be made part of a standardized benchmark suite.

As is the case with most benchmarks, the benchmark results depend not only on the properties of the particular EJB implementation, but also on the underlying database and JVM architectures, on the operating system, and on the hardware used while performing the evaluation. This means that it is difficult to define a precise meaning of the results unless they are very generic, as is the case, e.g., with the TPC benchmark. We believe, however, that even results that depend heavily on the factors mentioned above are useful, as they characterize the behavior of the EJB implementation in specific circumstances.

6 Conclusion

The EJB implementation tests outlined in this paper present a comprehensive test suite capable of evaluating most aspects of an EJB implementation. Applied to several EJB implementations during the course of the EJB Comparison Project [2], the tests indicate that EJB is promising, although not quite fully developed, technology, which can offer a reasonable level of functionality and decent performance. The current EJB implementations leave things to be desired in the area of standard compliance and code portability, but are otherwise functional.

The paper also highlighted several problems related to EJB implementation evaluation, ranging from the problems of benchmark portability to the problems of benchmark result interpretation in the situation where besides the EJB implementation itself, many other factors influence the results. Some of these problems can make introducing a standardized suite for evaluating EJB implementations difficult, and should be remedied by changes to the EJB standard itself (portability, ambiguity), or to the individual EJB implementations (configuration, importing).

References

- [1] CORBA Comparison Project, Final Report, Distributed Systems Research Group, Charles University, Prague, June 1998, <http://nenya.ms.mff.cuni.cz/thegroup/>.
- [2] EJB Comparison Project, Final Report, Distributed Systems Research Group, Charles University, Prague, January 2000, <http://nenya.ms.mff.cuni.cz/thegroup/>.
- [3] EJB Comparison Project, Final Report Public Distribution Version, Distributed Systems Research Group, Charles University, Prague, February 2000, <http://nenya.ms.mff.cuni.cz/thegroup/>.
- [4] R. Carg, Enterprise JavaBeans to CORBA Mapping 1.0, Sun Microsystems Inc., March 1998, <ftp://ftp.javasoft.com/docs/ejb/ejb-corba.10.ps>.
- [5] S. Cheung, Java Transaction Service 0.95 Specification, Sun Microsystems Inc., March 1999, <http://java.sun.com/products/jts/>.
- [6] S. Cheung, V. Matena, Java Transaction API 1.01 Specification, Sun Microsystems Inc., April 1999, <http://java.sun.com/products/jta/>.
- [7] S. C. Dorda, J. Robert, R. Seacord, Theory and Practice of Enterprise JavaBean Portability, Carnegie Mellon University, Software Engineering Institute, Technical Note CMU/SEI-99-TN-005, June 1999.
- [8] S. Krishnan: Enterprise JavaBeans to CORBA Mapping 1.1, Sun Microsystems Inc., August 1999, <http://java.sun.com/products/ejb/docs.html>.
- [9] V. Matena, M. Hapner, Enterprise JavaBeans 1.0 Specification, Sun Microsystems Inc., March 1998, <ftp://ftp.javasoft.com/docs/ejb/>.
- [10] V. Matena, M. Hapner, Enterprise JavaBeans 1.1 Specification, Public Release, Sun Microsystems Inc., August 1999, <http://java.sun.com/products/ejb/docs.html>.
- [11] R. Pospisil, M. Prochazka, V. Mencl, On Performance of Enterprise JavaBeans, In "Proc. Objects'99 Conference," pp. 145-156, Prague, November 12, 1999.
- [12] S. H. Y. Yasu, H. Igarashi, Performance Evaluation of Popular Distributed Object Technologies for Java, HORB Open and Electrotechnical Laboratory, <http://www.horb.org/eval-team/acm98/>.
- [13] Object Management Group, Object Transaction Service, December 1997, <ftp://www.omg.org/pub/docs/formal/97-12-17.ps>.
- [14] Object Management Group, CORBA IIOP, October 1999, <ftp://www.omg.org/pub/docs/formal/99-10-07.ps>.
- [15] ECperf Benchmark Specification, Java Specification Request JSR-000004, March 1999, http://java.sun.com/aboutJava/communityprocess/jsr/jsr_004_ecperf.html.
- [16] Java Development Kit (JDK) 1.1, Sun Microsystems Inc., <http://java.sun.com/products/jdk/1.1/docs/>.
- [17] Java Development Kit (JDK) 1.2, Sun Microsystems Inc., <http://java.sun.com/products/jdk/1.2/docs/>.
- [18] Java RMI, Sun Microsystems Inc, <http://java.sun.com/products/jdk/rmi/index.html>
- [19] Java RMI-over-IIOP, Sun Microsystems Inc, <http://java.sun.com/products/jdk/rmi/index.html>