Distributed Objects Performance Optimization and Modeling

Matjaz B. Juric, Ivan Rozman, Marjan Hericko, Tatjana Welzer, Jozsef Gyorkos

University of Maribor, Institute of Informatics
Smetanova 17, Maribor, Slovenia
E-mail: matjaz.juric@uni-mb.si
URL: http://lisa.uni-mb.si/~juric/

Abstract: *This paper is focused on the performance analysis, comparison and optimization of distributed object middleware for Java 2: RMI (Remote Method Invocation), CORBA IDL (Interface Definition Language) and RMI-IIOP (Remote Method Invocation on Internet Inter-ORB Protocol). The paper presents the following contributions to the research on distributed object performance. First, a detailed performance analysis is provided with the comparison. These results help to understand how the models perform. Second, an overhead analysis has been done, which explains why there are differences in performance. Third, optimizations and improved performance for RMI-IIOP and CORBA IDL are presented. These show considerably better performance in all areas compared to the original versions.*

1. Introduction

Reusable components such as JavaBeans and Enterprise JavaBeans and the global shift from developing applications from scratch to integrating components require high-level support for distributed object communication and interoperability. In version 1.1 Java has been enhanced with a native Java Remote Method Invocation (RMI) mechanism. RMI provides transparent remote method invocation between objects executing in different Java Virtual Machines (JVM). RMI has been designed for Java only and is tightly integrated with JVM. However, it addresses the same problem domain as CORBA (Common Object Request Broker Architecture). An implementation of the CORBA specification has been included in Java 2 (version 1.2) under the name Java IDL (Interface Definition Language). Activities have been initiated to merge RMI and CORBA. RMI that works with the CORBA standardized Internet Inter-ORB Protocol (IIOP) is known as RMI-IIOP and is a part of the Java 2 SDK version 1.3.

With the growing number of applications that use the distributed object middleware as their infrastructure, higher demands are placed on the middleware performance. Although the performance achieved through distributed object models may not be as good as when using low-level approaches, it is still very important that distributed object middleware provides at least acceptable performance for the given problem domain. As Java developers can choose from three middleware implementations that come with the Java 2 SDK, we found it very important to make a performance comparison. We have also done bottleneck identifications and optimizations for RMI-IIOP (in cooperation

with IBM Java Technology Centre) and for IDL. In this paper we present a set of benchmarks that enable quantitative evaluation of distributed object middleware performance in several relevant usage scenarios. Further, they enable the comparison of the results through different middleware architectures, e.g. RMI, and CORBA. Based on the benchmarks we present performance results for RMI, IDL and RMI-IIOP when used with Java 2 and we make a performance comparison. We also provide bottleneck identification and describe the optimizations that have been implemented on RMI-IIOP and IDL. Finally we present the improved performance for both middleware products.

The review of related work has shown that there is no standardized or commonly accepted method for performance assessment of distributed object architectures. Although several papers on distributed object performance have been published, the authors usually use simple benchmarks. The results between authors are not comparable. As far as we know, in the middle of the year 1999, in papers [Ju99a], [NPH00], and [SYH99] three different benchmarks have been proposed. The proposed benchmarks are based on similar concepts. Standardization on this area would be very welcome, as it would enable the comparison of the work, done by different authors. In this paper we present a short description of the benchmarks, proposed by us, and described in [Ju99a] and [Ju99b]. The actual performance comparison of Java distributed object models has been done in [He98], [JZR00], [JRH00], by the authors of this paper, and in [SYH99]. In our earlier work we have presented some performance results for CORBA/Java (using Inprise Visibroker) and RMI version 1.1. We have based our measurements on a modified ATM (Automatic Teller Machine) application. In [NPH00] performance evaluation of RMI can be found. There are also several papers in which different CORBA architectures with the C++ programming language are compared. In [DHE95] the authors report the performance results from benchmarking sockets and several CORBA implementations over Ethernet and ATM networks. In [AD96a] the authors compared the performance of socket-based communication, RPC (Remote Procedure Call), Orbix and ORBeline over ATM network and discovered the sources of overhead. They used a single client and a single server configuration. In [AD96b] the authors measured and explained the overhead of CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface. In [AD97] and [AD98] the authors systematically analyzed the latency and scalability of Visibroker and Orbix and revealed the sources of overhead. Again they used a single client and server configuration over ATM network. They also described techniques to improve the performance and they gave an overview of Tao. In [Sa97] the author described the implementation of a low overhead ORB. He presented some performance results where he used single client server configuration and C++ language. Some performance results in the context of real-time systems are presented in [DA97], [GLD98] and [SSC97]. In [PTB98] has been done a comprehensive comparison of different CORBA compliant ORBs with the C++ language. A common characteristic of all performance studies is that the authors used simple tests to measure performance and they only investigated single client scenarios.

This article is organized as follows: first, we describe the performance evaluation method. Then we present the performance evaluation results and make a comparison. These results represent how the tested systems perform. With the analysis of the results and the code we identify the sources of overhead and point to the bottlenecks. These

results depict why there are differences in performance. Finally, we present performance optimizations and show the performance improvements for RMI-IIOP and IDL.

2. Performance Evaluation

Our goal was to evaluate the performance of Java RMI, RMI-IIOP, and IDL from a developer's standpoint. We wanted to gain results that will make it possible to understand how do the models perform and why there are differences in performance. An important goal was to make the results comparable between the three (and possibly other) models. In this paper we have used a subset of the performance evaluation method, described in [Ju99a]. For the purposes of this article we have measured the round trip times (RTT), throughput and performance degradation (scalability) under single and multiple-client load. The method is independent of the underlying distributed object model and minimizes the influence and the overhead of the performance control component. We have provided results that are directly comparable between the three models. The benchmarks can be downloaded from http://lisa.uni-mb.si/~juric/ web page.

For all the performance measurements the Sun Java 2 SDK, Standard Edition, version 1.2 has been used. Performance for RMI-IIOP are reported for version 0.2. All source code has been compiled and executed using the Java 2 SDK, Standard Edition, version 1.2. Symantec Just-in-Time (JIT) compiler level 3.00.078(x) has been enabled. For the code analysis the Intuitive Systems Inc. OptimizeIt 3.02 Professional profiler has been used. All the computers used Microsoft Windows NT 4.0 Workstation with Service Pack 3 as their operating system. The actual performance measurements have been done for up to eight simultaneous clients. Therefore ten identical Pentium II-333 MHz computers with 128 MB RAM have been used. Eight of them were used for client applications, one of them was the server and one was used for running the synchronization objects. The computers were connected into a 100 Mbps Ethernet network that was free of other traffic.

3. Performance Measurement Results

Due to the limitation if the paper length we have decided to present the performance before optimization and the performance after optimization on the same set of graphs. Therefore, in this chapter focus on the unoptimized performance of RMI-IIOP and IDL. First we have measured the round trip times for methods that return primitive data types. We have observed, that different basic data types (boolean, char, byte/octet, short, int/long, long/long long, float and double) do not show any substantial differences in method invocation times. Therefore we report the averages for basic data types, RTTs for string, testStruct and myObject (Figure 3.1). In all cases RMI shows the best performance. IDL and RMI-IIOP are constantly slower. Although primitive data types as arguments or return values of methods are frequently used, it is also interesting to observe the throughput for arrays and sequences. We have done the measurements for arrays with 1 to 16384 elements. Figure 3.2 shows the comparison of the throughput for basic data types.

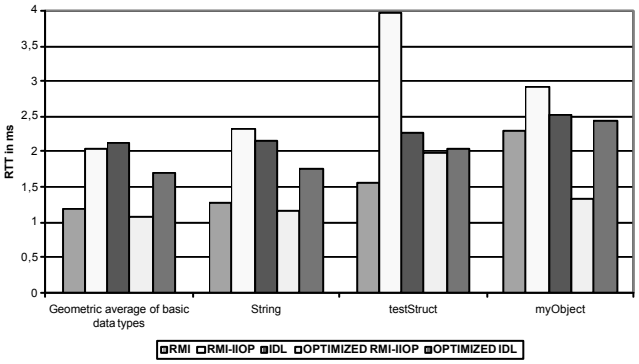

Figure 3.1: Round trip time comparison

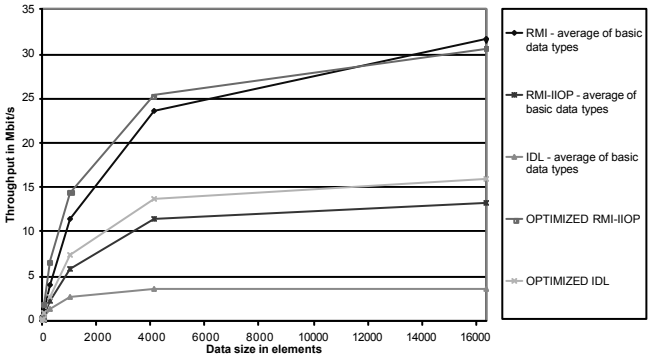

Figure 3.2: Throughput comparison for basic data types

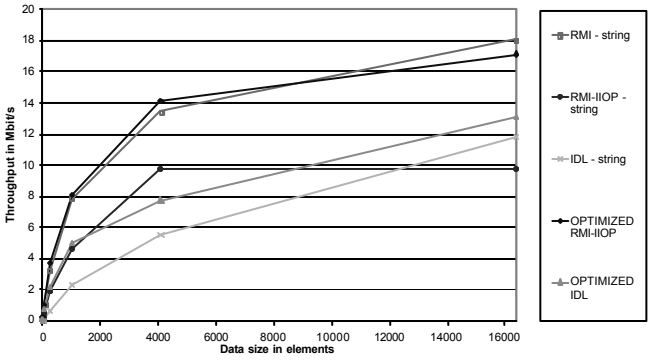

Figure 3.3: Throughput comparison for string

Figure 3.3 shows the throughput for string. It is important to understand, that both RMI and RMI-IIOP used Unicode characters, IDL on the other hand used 8 bit characters. Figure 3.4 shows the RTTs for arrays and sequences of testStruct, respectively. Figure 3.5 shows the RTTs for the object reference myObject. Here all three models were more

comparable. RMI achieved the best results, IDL was ~55% slower and RMI-IIOP 67% slower than RMI.

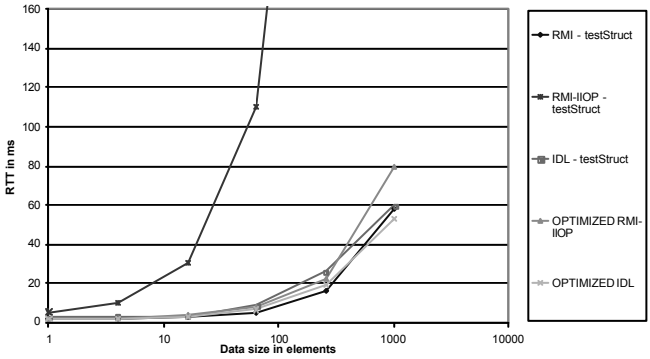

Figure 3.4: RTT comparison for testStruct

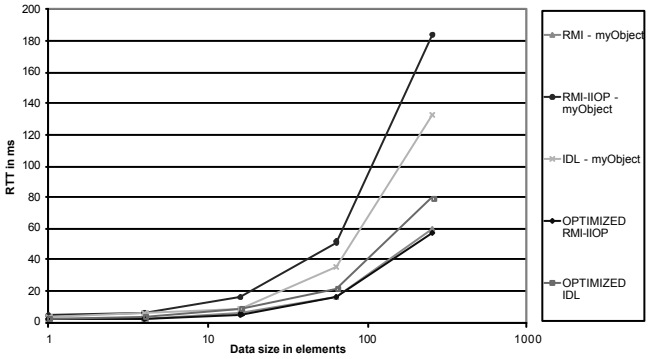

Figure 3.5: RTT comparison for object reference myObject

4. Overhead Analysis

The results show that the best overall performance has been achieved by RMI. The performance of IDL and RMI-IIOP is slower in every tested case. The tested version of RMI-IIOP 0.2 was a technology preview version; therefore it is understandable, that there are several bottlenecks. However, the performance of IDL is inadequate for real-world use. In this paper we present the performance optimizations for RMI-IIOP and IDL. The goal was to match the performance of RMI. While the RMI-IIOP optimizations have been done in cooperation with IBM Java Technology Center in UK, where the RMI-IIOP has been developed, and will be included in the shipping version of RMI-IIOP, the optimizations for IDL have been done as a pure research project at our university. We did not do any optimizations of RMI.

4.1. IDL Overhead Analysis

To process the client requests the IDL server object creates a pool of threads. The Java IDL Listener Thread handles the low-level connection details. For each client a Java IDL Reader Thread is created which handles the connection with a specific client. For each client also a pool of 201 Worker Threads is created. These threads process the IIOP protocol input streams. In the Listener Thread ~5% of the total time is spent, in the Reader ~80% and in a single Worker thread ~15%. On the server side, for the Listener Thread, a lot of time is spent is native methods, like socketAccept (~28% of total time) and getHostByAddr (22%). Around 17% of the total time is spent in thread management. In the Reader Thread, the majority of time is spent in reading the socket (method socketRead, ~56% to ~59%) and in creating new Worker Threads (~25% to ~49%). In the Working Thread, depending on the data size there are a few time consuming methods. The method read_octet reads the subsequent octet into the buffer (initial size 1024 bytes). Before this happens it invokes the alignAndCheck method. This method takes care for the proper data alignment and calls the method grow when the buffer becomes too small. The method grow enlarges the buffer by a factor of two. The method write_octet writes the subsequent octet to the buffer. It uses the alignAndReserve method which supervises the buffer size and calls the grow method to enlarge it. These three methods (alingAndCheck, alingAndReserve and grow) spend up to 42% of execution time. A consequence is data copying. The method TypeCodeImpl.copy copies the input stream to the output stream and with larger data sizes has a significant share in the total execution time (up to ~33%).

On the IDL client side there are two important threads. The Java IDL Reader Thread spends the majority of time in the java.net.SocketInputStream.socketRead method which we have already described. The behavior is very similar to the RMI-IIOP client. Second thread is the main thread, where the majority (over 90%) of the client side execution time is spent. We have observed that with the increasing data size the majority of time is spent in TypeCodeImpl.copy, write_octet, read_octet, alignAndReserve, alingAndCheck, and grow methods (up to ~92%). We have already described the functionality of these methods.

4.2. RMI-IIOP Overhead Analysis

On the server side, RMI-IIOP uses the thread pool with the Listener, Reader, and Worker threads. It allocates 202 Worker Threads per client. The behavior of the Listener and Worker threads is similar as in IDL (described in previous subsection). In the Reader Thread the majority of time is spent in the socket communication (socketRead – up to ~50%) and in the thread creation (Thread.start – up to ~38%). With the increasing data size the initialization of the IIOP input stream becomes time consuming. RMI-IIOP client creates two threads, the Reader Thread which handles the low-level communication and the main thread. The majority of the time in the Reader Thread is spend in the socketRead method (up to ~88%). With the increasing data size the method processInput becomes time consuming too (up to ~11%). The client side overhead for the main thread is comparable to the IDL client side main thread. With RMI-IIOP however more time is spend in writing to the sockets and in particular the output stream grow method has a larger share of up to ~26% of the total time, compared to ~11% by IDL.

4.3. Summary

Based on the overhead analysis, we have identified the following points, where optimizations will be applied. Much of the receiver side (the server object) and some of the sender side (the client object) overhead rises from the inefficient concurrency support – the multithreading exploitation shows comparably large overhead (Thread.start, Thread.init, Thread.run). The other important source of the receiver side overhead can be found in the demultiplexing, demarshalling and presentation layer. IDL and RMI-IIOP use inefficient algorithms for demultiplexing which perform heavy checking (alignAndCheck) and use bad buffering algorithm which requires buffer enlargements (grow) and leads to excessive data copying (copy). An important source of the sender side overhead is inefficient multiplexing, easily identified by IDL and RMI-IIOP. This is partly because of sticking with the GIOP 1.0 specification and partly due to inefficient algorithm which performs aligning and reserving (alignAndReserve), has small initial buffer size and inefficient buffer growth method (grow). Vast amount of the communication overhead resides in the low-level methods that take care for socket based communication (socketWrite, socketRead, socketAccept). These methods are not implemented in Java. Rather they are written natively for each target platform in C language and use the underlying operating system calls. To access them, the Java Native Interface is used, which is not very efficient. Other sources of overhead lie in the excessive data copying and local ORB method invocations, and in generated stubs and skeletons. For example, by demultiplexing strings, they are first copied into a local array and then converted to the actual java.lang.String representation, floats and doubles are first converted to integers (floatToIntBits and doubleToIntBits methods), often redundant arrays are created, etc.

5. Optimizations

Based on the performance analysis and overhead identification we present an overview of the optimizations that have been implemented on RMI-IIOP and IDL. To make the concurrency model more efficient it is necessary to implement highly optimized thread management. For reducing the thread creation time a thread pool can be used. The context switching should be minimized. In the Leader/Follower Thread Pool architecture context switching is successfully minimized because the request is not transferred from one thread to another. Therefore it provides better performance than the Worker Thread Architecture used by IDL and RMI-IIOP, although it is harder to implement.

Demultiplexing overhead is minimized with the use of fast, de-layered and flexible demultiplexing algorithms. Instead of the layered demultiplexing a perfect hashing and active demultiplexing can be implemented for best performance. Presentation layer overhead optimization is achieved by generation of optimized stubs and skeletons. The excessive marshalling and demarshalling overhead in IDL and RMI-IIOP is based on non-optimal buffer allocation and enlargements. We suggest two solutions to this problem. First, the initial buffer size can be adjusted and the enlargement procedure can be optimized. Second, the ORB can be upgraded for support for the GIOP version 1.1.

GIOP 1.1 has introduced a new message type (Fragment) that allows a message to be sent in portions. Therefore the buffer enlargements become superfluous. All three ORBs can be optimized to omit the unnecessary data copying, which would provide improvements most noticeable by larger data sizes. Some optimizations could be achieved with the techniques for optimized range checking. With the careful internal design the count of the unnecessary local ORB method invocation can be minimized. Implementation based optimizations such as minimizing the invocation overhead of frequently called methods with the optimization for the common case, the replacement of large methods with efficient small special purpose methods, avoiding the repeated computation of invariant values and storing redundant data all provide performance improvements. Also, elimination of the run-time checking for the debugging code brings some improvements. Improving the low-level communication overhead requires: (1) optimizations in the native code that handles the communication and (2) optimizations in the JNI. On one side this is the integration with the operating system network features, especially advanced features, such as high-speed network interfaces, and real-time threads. On the other side something can be done with buffer optimizations and their optimal management and with the transport protocol tuning (for example socket lengths can be adjusted). None of the three ORBs evaluated uses an internal buffering for network writing/reading. An optimal buffering architecture would reduce the communication overhead considerably. RMI omits the unnecessary data conversions, found in the IDL and RMI-IIOP communication. Optimizations in the implementation of the GIOP protocol, that prevent unnecessary data conversions when the sender and receiver use the same format speed the communication.

6. Performance Improvements for RMI-IIOP and IDL

On Figure 3.1 the round trip time improvements for primitive data types are presented. Figure 3.2 shows the improvements in throughput for the geometric average of basic data types. The optimized RMI-IIOP achieved throughput that was always comparable to RMI. This is a 2.3 times improvement, compared to the unoptimized version. IDL has also been improved and now achieves peak vales of 16 Mbit/s. This is a 4.5 times improvement. Figure 3.3 shows the throughput improvements for string. Figure 3.4 shows the improvements in round trip times for the arrays and sequences of the testStruct Figure 3.5 shows the improvements for the arrays of object reference. The modifications in the thread management have improved the scalability of RMI-IIOP considerably and is now comparable to RMI. As the actual invocation times for optimized RMI-IIOP have been considerably improved too, and are now better than for RMI, also the actual RTTs in multi-client scenarios have improved. This has been verified by additional performance measurements that are not presented in this paper.

7. Concluding Remarks

The increasing demand for application connectivity and interoperability intensifies the role of distributed object middleware in all areas of application development. With the

use of distributed object models in mission critical applications the performance question becomes very important. In this paper a detailed performance analysis of important distributed object models for Java has been done: RMI, IDL and the newly developed RMI-IIOP, which enables the use of RMI over the IIOP protocol and makes it CORBA compliant. Performance has been measured for relevant usage scenarios which include single and multi-client interactions with up to eight simultaneous clients, different basic and compound data types and different data sizes. In the performance comparison RMI provided the best results in all tests. Although for the basic data types the results differed by a factor of ~2 the larger data sizes showed significant differences. The maximum throughput achieved by RMI was ~30 Mbit/s, which is around a third of the theoretical network throughput. RMI-IIOP achieved only ~13 Mbit/s and IDL only ~3.5 Mbit/s. In multi-client scenarios RMI-IIOP was up to ~90% and IDL up to ~70% slower than RMI.

The overhead analysis showed that the bottlenecks fall in the following categories: (1) the inefficient thread management, (2) ineffective algorithms for demultiplexing and demarshalling, (3) excessive data copying and local method invocations, (4) the overhead of the low-level methods that take care of socket based communication and the JNI. Proposed optimizations were applied on RMI-IIOP and IDL and the performance was remeasured. The presented results for the optimized RMI-IIOP version show performance that is always comparable and some times even superior to the RMI. Although RMI-IIOP uses the standardized IIOP protocol for which it has to make additional transformations and mappings, the careful design and implementation, together with performance analysis and implementation of several optimizations, have enabled good performance. Similar performance can be expected from the final release version, as this has been a joint project with the IBM Java Technology Centre, UK, where the RMI-IIOP has been developed.

In the optimizations of IDL we did not invest as much time and resources. Therefore the improvements were not as big as by RMI-IIOP. However, the optimized IDL performs much better than the original one, with up to 5 times improvements in throughput. However, we believe that neither RMI-IIOP and IDL, nor RMI have reached the limits and there is still a lot of room for further optimizations. This is what our future research work will be based on.

References

[AD96a] Aniruddha S. Gokhale, Douglas C. Schmidt, Measuring the Performance of Communication Middleware on High-Speed Networks, SIGCOMM Conference, ACM 1996, Stanford University, August 28-30, 1996

[AD96b] Aniruddha S. Gokhale, Douglas C. Schmidt, The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks, IEEE GLOBECOM '96 conference, November 18-22nd, 1996, London, England

[AD97] Aniruddha S. Gokhale, Douglas C. Schmidt, Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks, IEEE 17th International Conference on Distributed Systems (ICDCS 97), May 27-30, 1997, Baltimore, USA

[AD98] Aniruddha S. Gokhale, Douglas C. Schmidt, Measuring and Optimizing CORBA Latency and Scalability Over High-Speed Networks, IEEE Transactions on Computers, Vol. 47, No. 4, April 1998

[DA97] Douglas C. Schmidt, Aniruddha Gokhale, Timothy H. Harrison, Guru Parulkar, A High-performance Endsystem Architecture for Real-time CORBA, Distributed Object Computing, IEEE Communications Magazine, Vol. 14, No. 2, February 1997

[DHE95] Douglas C. Schmidt, Harrison Tim, Ehab Al-Shaer, Object-Oriented Components for High-speed Network Programming, 1st Conference on Object-Oriented Technologies, USENIX , Monterey, CA, June, 1995

[GLD98] Christopher D. Gill, David L. Levine, Douglas C. Schmidt, The Design and Performance of a Real-Time CORBA Scheduling Service, August 1998

[He98] Hericko Marjan, Juric B. Matjaz, Zivkovic Ales, Rozman Ivan, Java and Distributed Object Models: An Analysis, ACM SIGPLAN Notices, December 1998

[Ju99a] Juric B. Matjaz, Domajnko Tomaz, Zivkovic Ales, Hericko Marjan, Brumen Bostjan, Welzer Tatjana, Rozman Ivan, Performance Assessment Framework for Distributed Object Architectures, Conference proceedings of ADBIS'99, LNCS series of Springer Verlag, September 1999

[Ju99b] Juric B. Matjaz, the Efficiency of Distributed Object Models, ACM OOPSLA'99, November 1999

[JRH00] Juric B. Matjaz, Rozman Ivan, Hericko Marjan, Performance Comparison of CORBA and RMI, Information and Software Technology Journal, Elsevier, 2000

[JZR00] Juric B. Matjaz, Zivkovic Ales, Rozman Ivan, Are Distributed Objects Fast Enough, *More Java Gems,* Cambridge University Press, March 2000

[NPH00] Nester Christian, Philippsen Michael, Haumacher Bernard, A More Efficient RMI for Java, 1999 ACM Java Grande Conference, San Francisco, California, June 12-14, 1999

[OMG98a] Object Management Group, The Common Object Request Broker: Architecture and Specification, Revision 2.2, February 1998

[OMG98b] Object Management Group, Objects By Value, Joint Revised Submission w/ Errata, OMG TC Document orbos/98-01-18

[OMG99] Object Management Group, Java Language to IDL Mapping, Chapter 28, OMG document ptc/99-03-09

[PTB98] Plasil,F., Tuma,P., Buble,A.: CORBA Benchmarking, Tech. Report No. 98/7, Dep. of SW Engineering, Charles University, Prague

[Sa97] Sai-Ali Lo, The Implementation of a Low Call Overhead IIOP-based Object Request Broker, Olivetti & Oracle Resharch Laboratory, April 1997

[Su98a] Sun Microsystems, Inc., Java Remote Method Invocation Specification, Sun Microsystems, October 1998

[Su98b] Sun Microsystems, Inc., Java Object Serialization Specification, Sun Microsystems, November 1998

[SSC97] Ashish Singhai, Aamod Sane, Roy Campbell, Reflective ORBs: Supporting Robust, Time-critical Distribution, ECOOP'97 Workshop Proceedings, 1997

[SYH99] Satoshi Hirano, Yoshiji Yasu, Hirotaka Igarashi, Performance evaluation of popular distributed object technologies in Java, Concurrency, Practice and Experience, 10, Sept-Nov 1999