CORBA, RMI and RMI-IIOP
Performance Analysis and Optimization

Matjaz B. Juric, Ivan Rozman, Marjan Hericko, Tomaz Domajnko
University of Maribor, Faculty of Electrical Engineering and Computer Science,
Institute of Informatics, Smetanova 17, SI-2000 Maribor, Slovenia, Europe
E-mail: matjaz.juric@uni-mb.si, URL: http://lisa.uni-mb.si/~juric/

ABSTRACT

Java and distributed object models are important for building modern, scalable, interoperable applications. This paper is focused on the performance analysis, comparison and optimization of distributed object middleware for Java 2: RMI (Remote Method Invocation), IDL (Interface Definition Language – CORBA) and RMI-IIOP (Remote Method Invocation over 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 IDL are presented. These show considerably better performance in all areas compared to the original versions.

Keywords: Java, RMI, IDL, CORBA, performance, IIOP

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) that is covered by the activities of Object Management Group (OMG). 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.

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 the 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 and IDL, partly in cooperation with IBM Java Technology Centre, Hursley, UK. 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 architecture, 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 a 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 [1], [2], and [3] three different benchmarks have been proposed. The proposed benchmarks are based on similar concepts. A 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 [1] and [4].

The actual performance comparison of Java distributed object models has been done in [5], [6], [7], by the authors of this paper, and in [3]. 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 [2] 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 [8] the authors report the performance results from benchmarking sockets and several CORBA implementations over Ethernet and ATM networks. In [9] 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 [10] the authors measured and explained the overhead of CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface. In [11] and [12] 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 [13] 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 [14], [15] and [16]. In [17] 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 that they only investigated single client scenarios.

As already mentioned, our work is focused on performance comparison of Java 2 distributed object middleware and on their optimization. The 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

Method

Our goal was to evaluate the performance of Java RMI, RMI-IIOP, and IDL from a developers 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. We have covered relevant aspects such as different basic and user defined data types, different data sizes and multi-client scenarios which are especially important in global networked applications.

In this paper we have used a subset of the performance evaluation method, described in [1]. 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 the purposes of this article we have measured the round trip times (RTT), throughput and performance degradation (scalability) under single and multiple-client load. Round Trip Time (RTT) is the time that elapses between the initiation of a method invocation by the client until the results are returned to the client. Because the methods used for performance evaluation did not do any processing the RTT expresses the overhead of remote method invocation. It is important to understand:

- How different data types as parameters and return values influence the result: RTT for eight simple data types, string, user defined data type and an object reference have been measured.
- How the data size influences the results: Performance results for different data arrays from size 1 up to size 16384 have been gathered.
- How multi-client interactions influence the results: RTTs for one and up to eight simultaneous clients have been measured.

Performance degradation, when multiple clients simultaneously interact with a single server, is defined as a function of number of simultaneous clients. Performance degradation is computed as quotient between the RTT by the given number of simultaneous clients and the RTT by a single client. Performance degradation for up to eight simultaneous clients have been measured.

The comparability of the results between RMI, RMI-IIOP, and IDL is achieved with the identical implementations that differ only in necessary details regarding obtaining the initial references. Further, there is a consistent mapping between Java and CORBA data types and the measurements have been accomplished on identical equipment in an identical environment.

Software and Hardware Testbed Equipment

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. Optimizelt 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

Single Client Scenarios

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 1).

In all cases RMI shows the best performance. IDL and RMI-IIOP are constantly slower. For basic data types, IDL is ~44% slower and RMI-IIOP ~42% slower than RMI. Similar holds true for string, where IDL is ~40% and RMI-IIOP ~44% slower. The largest difference can be observed by the testStruct. Here we can see, that IDL is ~31% and RMI-IIOP ~61% slower than RMI. For the object reference myObject, the times do not differentiate as much. IDL is ~8% and RMI-IIOP ~20% slower than RMI.

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. With larger data sizes the differences in data type sizes were easily identifiable. The byte/octet data type which is 8 bits long when transferred over the wire was the fastest, followed by boolean (16 bits) and so on up to the data type double which is 64 bits long. Both compound data types (testStruct, myObject) showed different behavior and the invocation times (RTTs) raised for them faster than for basic data types. Figure 2 shows the comparison of the throughput for basic data types. We can see, that RMI achieved the best performance. RMI-IIOP and particularly IDL showed a large gap. The average throughput of RMI for basic data types was as high as 31 Mbit/s for 16k arrays. RMI-IIOP achieved 13.2 Mbit/s, and IDL 3.6 Mbit/s only.
Figure 1: Round trip time comparison

Figure 2: Throughput comparison for basic data types

Figure 3 shows the throughput for string. Although RMI still achieved the best results (18 Mbit/s), 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: Throughput comparison for string

Figure 4 shows the RTTs for arrays and sequences of testStruct, respectively. We have already explained that testStruct is handled differently by RMI, IDL and RMI-IIOP. We can see, that RMI and IDL achieved comparable results, although RMI transferred testStruct as object by value. IDL on the other hand handled testStruct as simple data structure. We can also see, that RMI-IIOP handled arrays of testStruct very badly and its performance was hardly acceptable.

Figure 4: RTT comparison for testStruct

Figure 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.
Multi Client Scenarios
In many occasions the distributed objects provide services that are used by several clients simultaneously. To be able to understand the performance levels offered by RMI, IDL and RMI-IIOP in multi-client scenarios we have run several tests where we have gathered the results for two to eight simultaneous clients that invoked the methods without delays. We have found out that different basic data types did not have any relevant impact on the performance. To simplify the results, we have decided to present the geometrical mean of basic and compound data types. In Figure 6 we can see the method invocation times depending on the number of simultaneous clients and in Figure 7 the performance degradation factor.

In absolute times the RMI preserved its advantage over Java IDL and RMI-IIOP. By RMI and RMI-IIOP we can observe a linear increase in method invocation times when the number of clients increases. By IDL the times rise faster for first four clients and than again slower. For example, by four simultaneous clients RMI was 2 times and by eight clients 30% faster than IDL. RMI was also ~75% faster by four clients and ~50% faster by eight simultaneous clients than RMI-IIOP.

When observing the performance degradation factor we can see that all three models are more comparable. By eight clients, IDL’s relative degradation is even lower than RMI’s, although RMI is better for every smaller number of clients. RMI-IIOP’s degradation is always a little higher than RMI’s. By four clients we can expect a performance degradation from three to four and by eight clients from...
five to seven. However keep in mind that in these tests the clients repeated the method invocations without any delays, therefore the presented number of “synthetic” clients is equivalent to a much larger number of real world clients.

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.

The results of RMI-IIOP show, that RTTs for basic data types are ~80% lower than RMI’s. There is an obvious problem by handling of testStruct; response times are 2.5 times slower than RMI’s. The object reference is handled better, although it is still slower; RMI-IIOP also shows deficits by transferring large amounts of data. The achieved throughput is an order of magnitude slower than RMI’s. As the throughput is lower irrespective of data types, this shows on deficits by marshalling and serialization. In multi client scenarios the RMI-IIOP performs relatively good, if we consider the relative performance degradation. It is reasonable to presume, that if we would be able to optimize single client response times, we would also be able to improve the response times in multi-client scenarios.

IDL shows some variations in the results. By basic data types, IDL achieved response times, that are slower than RMI’s, but better than RMI-IIOP’s. However, the throughput, that was achieved by IDL was really bad, particularly by the basic data types. The only cases where IDL achieved acceptable results was by string throughput and testStruct. However it is important to understand that IDL string were half shorter, as the characters were 8 bits long only and that the IDL testStruct is a simple data structure and not an object. Similar as by RMI-IIOP the low throughput shows on deficits by marshalling. IDL however achieved good relative performance degradation factor in multi-client scenarios. This shows, that the multi-threading scheme, implemented by IDL is more efficient that the one, used by RMI-IIOP.

To be able to identify the major sources of overhead, we have done an overhead analysis. To get the insight into multiplexing/demultiplexing, dispatching and demarshalling we have decided to use the class octetSeqTestServer. Data type byte/octet requires the least amount of conversion, therefore it is the data type best suited for studying and comparing the behavior. We have analyzed the method acceptType, which has been invoked 200 times. This method had a parameter – an array/sequence of bytes/octets. Therefore the client object had to marshal the parameter, which was transferred to the server object, where it was demarshaled, demultiplexed and dispatched. The server object returned only an acknowledgement to the client side. Therefore, to study marshalling we will focus on the client side. To study demarshalling we will look at the server object.

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 (alignAndCheck, alignAndReserve 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, alignAndCheck, and grow methods (up to ~92%). We have already described the functionality of these methods.

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.

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 concurrence 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 optimizations are 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 8 the round trip times improvements for primitive data types are presented. With the applied optimizations the response times of the methods that returned basic data types improved. RMI-IIOP now performs in average ~8% better than RMI and is more than 80% faster than the unoptimized RMI-IIOP. Similar improvements have been achieved by string, which is ~9% faster than RMI. The performance of the testStruct did not achieve the RMI’s performance. RMI-IIOP however still performs within 30% of RMI and is more than 2 times faster than the unoptimized RMI-IIOP. The testStruct is also the most complicated compound data type, as it is transferred conforming to the Object By Value specification. The object reference myObject achieved a large improvement with RMI-IIOP times that are over 40% faster than RMI (also more than 2 times faster than the unoptimized RMI-IIOP).

Due to the lack of time and resources, the improvements for IDL did not achieve the level of RMI-IIOP. However, the optimized IDL still performs better than the original. The RTTs for basic data types have been improved for 20%, strings for 18%, the testStruct for 11% and object reference for 3%, compared to the original unoptimized IDL. The performance of RMI has not been achieved. However, there is still room for further optimizations. We believe, that in the future it will be possible for IDL to achieve RMI or even overtake it.

To present the impacts of data size, we have measured round trip times and throughput for different arrays sizes, from 1 to 16384 elements. Figure 9 shows the improvements in throughput for the geometric average of basic data types. As we were able to improve the marshalling and demarshalling process, the improvements, noticed by primitive data types can also be seen here. The optimized RMI-IIOP achieved throughput, that was always comparable to RMI. This is an 2.3 times improvement, compared to the unoptimized version. IDL has also been improved and now achieves peak values of 16 Mbit/s. This is an 4.5 times improvement. RMI however still performs 2 times faster than the optimized IDL.

Figure 10 shows the throughput improvements for string. The optimized RMI-IIOP again achieved the performance of RMI (an 1.8 times improvement over unoptimized RMI-IIOP). The IDL on the other hand has been improved for ~15% only. String already achieved good performance in the original IDL.

![Figure 8: Round trip time improvements for primitive data types](image)
Figure 9: Throughput improvements for basic data types

Figure 10: Throughput improvements for string

Figure 11 shows the improvements in round trip times for the arrays and sequences of the testStruct by RMI, RMI-IIOP and IDL, respectively. By RMI-IIOP the RTTs for 256 element arrays of testStruct have been improved by as much as 19 times. RMI-IIOP now achieves times, comparable to RMI. By 1024 element arrays it is still ~35% slower. IDL on the other hand handles testStruct as a simple data structure. The performance has been improved by more than 15% and in some cases IDL is now even faster than RMI.
Figure 11: Round trip time improvements for testStruct

Figure 12 shows the improvements for the arrays of object reference. By primitive data types the optimized RMI-IIOP achieved over 40% better times than RMI. By arrays it is still faster than RMI, although the difference is only 5%. The optimized RMI-IIOP is however more than 3 times faster than the original. IDL shows 65% time improvement, compared to the original IDL, but is still more than 30% slower than IDL.

Figure 13 shows the performance degradation factor improvements for RMI-IIOP. 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 are a little better than for RMI, also the actual RTTs in multi-client scenarios are better form RMI-IIOP.

As we did not yet improve the thread model of IDL, we have not repeated the measurements for the multi-client scenarios in IDL. However, as we were able to optimize the RTTs, we believe, that the actual RTTs in multi-client scenarios are also better. We will do the optimizations in the thread management in the future.
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 sometimes 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 RMI-IIOP 1.0.1, 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


