RMI, RMI-IIOP and IDL Performance Comparison

Matjaz B. Juric, Ivan Rozman

1. Introduction

Distributed object models are important technologies and are widely used in modern applications. Once, even before Java was born, developers had only one choice, CORBA. CORBA is a language independent technology and a mapping to Java has been defined shortly after Java has been invented. When releasing Java version 1.1, Sun decided not to support CORBA, but to add a Java-specific distributed model, or more exactly, an object request broker – RMI (Remote Method Invocation). There were several reasons for doing so; the most important was that CORBA by the time did not support passing objects by value.

CORBA however, was an important technology, and Sun was (and still is) one of the most important members of the OMG. Therefore in Java 2, version 1.2, Sun added an implementation of CORBA architecture, known as Java IDL. The developers started to wonder, which technology should they use. They could select between RMI, IDL and several commercial CORBA implementations. There were discussions about the advantages and disadvantages of RMI and CORBA, and which one is more suitable for Java developers. In February 2000 issue of Java Report [1] we have done a detailed functional comparison of RMI and CORBA. In the same article we have also done a performance comparison, the second one. The first one has been published in May 1998 [2]. We have found out, that RMI’s performance was in almost all cases superior to IDL’s. This, and the fact, that IDL did not implement some important parts of CORBA specification (like wchar, wstring, and implementation repository), resulted in our recommendation, that RMI was the first choice in distributed object development with Java.

Already in 1998 Sun started an initiative to make RMI compatible with CORBA. In cooperation with IBM, RMI-IIOP has been developed. RMI-IIOP is a part of Java 2 platform from version 1.3. It uses the CORBA standardized protocol, IIOP (Internet Inter-ORB Protocol). This makes RMI-IIOP objects compatible with other CORBA objects. RMI-IIOP almost fully preserves the RMI API, therefore it is relatively easy to port existing RMI applications to RMI-IIOP. To be able to accommodate the additional RMI functionality, CORBA has been extended with Objects by Value specification [3] and Java to IDL Mapping [4].

RMI-IIOP has an important mission to fulfill. It is not the third distributed object model, added to Java 2 platform. It is the ultimate model, which will substitute RMI; and it is the preferred model for distributed objects development in Java. The RMI-IIOP distributed objects are compatible with CORBA objects, yet they are as easy to develop as RMI objects. RMI-IIOP is also an important technology for Java component model – EJB (Enterprise Java Beans).

---

1 Even DCOM was not born yet.
Particularly in distributed systems development, performance is traditionally a major factor. In this article we will focus on performance comparison of RMI, RMI-IIOP, and CORBA IDL. We will show the results of several performance measurements and compare the three models. To get trustworthy results, we have used different method types, different data types, and different data sizes. We have measured the performance on two important platforms, on Windows 2000 Professional and on Linux. We will try to figure out, which distributed model delivers the best performance and try to answer the question, whether RMI-IIOP is suitable to replace RMI and maybe even IDL.

We will also try to find out, whether the performance of RMI and IDL improved with new releases of Java 2 SDK. Our motivation was to give the developers the information to decide, if it is reasonable to upgrade to the latest version of Java 2 platform.

2. Testing Method

For performance measurements, we have used the Performance Evaluation Model for Distributed Objects, developed at the University of Maribor [5]. This is the same model, which has been used in our previous performance comparison [1]. When comparing the results, keep in mind, that this time we performed the tests on different operating systems.

For the performance measurements we have used the Sun Java 2 SDK, Standard Edition. For Windows 2000 Professional, we have used SDK version 1.2, SDK 1.2.2, and SDK 1.3. Versions 1.2 and 1.2.2 used Classic VM, native threads and JIT compiler was enabled. Version 1.3 used HotSpot client VM in mixed mode (default settings). We have used the Linux Gentux 3.0, with bash shell, unoptimized kernel version 2.2.15-3.0. Java 2 SDK version 1.2.2 for Linux used Classic VM with green threads and no JIT compiler. Version 1.3 used HotSpot client VM in mixed mode. We have not tested version 1.2 on Linux.

All measurements have been done on a single Pentium II-350 MHz computer with 128 MB RAM. All irrelevant tasks on both operating systems have been closed. The computer had a 100MBps Ethernet network card, which has been connected to a 100MBps network, free of other traffic. We have repeated each test 20 times and report the average results. To get the necessary timing accuracy each test has been iterated 200 times. To prevent mistakes, we have calculated standard deviation for each test.

For the purposes of this article we have measured the round trip times (RTT) and throughput. Round Trip Time (RTT) is the total time a method invocation takes. The methods used for performance evaluation transferred parameters and return values only. They did not have any processing code. Therefore the RTT expresses the overhead of a remote method invocation. To understand how different data types influence the results, we have measured the performance for eight simple data types, string, user defined data type and an object reference. These data types, together with consistent mappings between RMI, RMI-IIOP and CORBA IDL, are shown in Table 1.
Table 1: Data types used for performance measurements and their mapping

<table>
<thead>
<tr>
<th>RMI and RMI-IIOP</th>
<th>CORBA (RMI/IDL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>char</td>
<td>char and wchar</td>
</tr>
<tr>
<td>byte (signed)</td>
<td>octet (unsigned)</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
</tr>
<tr>
<td>int</td>
<td>long</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>String</td>
<td>string and wstring</td>
</tr>
<tr>
<td>testStruct</td>
<td>testStruct</td>
</tr>
<tr>
<td>(object-by-value)</td>
<td>(IDL struct)</td>
</tr>
<tr>
<td>myObject</td>
<td>myObject</td>
</tr>
<tr>
<td>(object reference)</td>
<td>(object reference)</td>
</tr>
</tbody>
</table>

As you can see, the mapping is quite straightforward. Due to the fact, that Java IDL in version 1.2.x did not implement the wchar and wstring data types, we have used char and string instead (in version 1.2.x only). You should however be aware when comparing the results, that char and string transfer only 8 bits “over the wire”. In version 1.3 wchar and wstring are implemented and we have used them for the measurements.

The testStruct is a user defined data type with data fields only. It maps to a CORBA IDL structure, which is a data only element. In RMI and RMI-IIOP it maps to a class, which means that testStruct object has to be transferred by value. Therefore it has to be serializable, for which we provided custom methods. RMI-IIOP handles the testStruct as a RMI/IDL Value Type and the transfer conforms to the Objects By Value specification [3]. As the sending and the receiving context had identical local implementations no code downloading has been used.

The myObject on the other hand is a user defined class. RMI, RMI-IIOP and IDL transfer it by reference. The difference between myObject and testStruct can be easily observed from the source code. MyObject extends the Remote interface and the testStruct does not.

We have not stuck with simple basic data type measurements. We also wanted to observe, how do these models behave, when larger data quantities are transferred as method parameters and return values. Therefore we have used interfaces, which send and receive arrays and sequences of all the described data types. In RMI-IIOP the RMI/IDL arrays are mapped to value types that contain IDL sequences.

To achieve comparability of the results, we have implemented the Performance Evaluation Model for each model identically. The differences were only in necessary details regarding obtaining the initial references, which do not influence the results. We have already mentioned, that the measurements have been done on identical equipment. To read more about the Performance Evaluation Model please refer to [1] or visit our home page [5], where you can download the source code and documentation.

2 Java IDL in version 1.2.x did not support wchar data type.
3 Java IDL in version 1.2.x did not support wstring data type.
3. Method invocations with primitive data types

We have observed the method invocation times for methods that accept and return primitive data types. To simplify the results we have calculated the geometric average of basic data types (boolean, char, byte/octet, short, int/long, long/long long, float and double). We present the results for string, testStruct and myObject, too.

Figure 1: Method Invocations with Primitive Data Types

Figure 1 shows the average method invocation times (Round Trip Times – RTT) for Java 2 SDK version 1.3. There are two bars for each distributed model; the first shows the performance under Windows 2000 Professional and the second under Linux operating system. First of all, we can see that under Windows 2000, the performance is slightly better. Second, CORBA IDL achieved round trip times, which are more constant, irrespective of the data types, and does not suffer as much from RTT increase by user defined types.

The differences for basic data types and strings are small. In both cases the RMI achieved the shortest times. RMI-IIOP was ~15% and ~30% slower than RMI under Windows 2000 and Linux, respectively. IDL was ~23% slower than RMI under both operating systems.

More interesting are the user defined data types. The testStruct shows considerable differences. We can see, that IDL achieved the fastest times. This is expected, because IDL handles the testStruct as simple data structure. Both RMI and RMI-IIOP transfer the testStruct as object by value. Therefore it is understandable that RMI is ~40% and ~65% slower than IDL under Windows and Linux, respectively. RMI-IIOP is ~60% and ~90% slower.

The object reference, myObject, also shows variations. Although all three models handle it the same way (conceptually), we can see, that RMI is here by far the slowest. It is particularly interesting, that RMI-IIOP achieved such good results and was only ~15% and ~27% slower than IDL under Windows and Linux, respectively. RMI on the other hand was as much as ~90% and ~160% slower.
These results imply that RMI does not have a performance edge over the other models anymore. Even worst, it seems that RMI performs slower than RMI-IIOP and IDL. In our previous article [1], where we have measured the performance of RMI and IDL under Java 2 version 1.2, the situation was totally different. Therefore we have remeasured the performance of Java 2 versions 1.2 and 1.2.2. To further simplify the comparison, we have calculated a weighted average of RTTs: 30% was contributed by average of basic data types, 30% by string, 20% by testStruct and 20% by myObject.

Figure 2 shows the weighted average round trip times for RMI, IDL and RMI-IIOP for Windows 2000 Professional. For RMI and IDL the results for Java 2 versions 1.2 and 1.2.2 are also included. These results confirm our previously published results and show that IDL was particularly slow in version 1.2. It was considerably improved by version 1.2.2 and once again by version 1.3. RMI was improved too, but not as much as IDL. In Java 2 version 1.3 all three models show very comparable performance, with a small advantage of CORBA (Java IDL). RMI-IIOP shows slightly better performance than RMI. Therefore its role to succeed RMI is justified.

Figure 3 shows the performance of RMI, IDL and RMI-IIOP for Linux. This time the results for RMI and IDL in Java 2 version 1.2.2 are included. The picture is similar to
the one under Windows 2000 operating system. Java 2 version 1.3 shows considerable improvements in distributed objects performance for all three models, again led by IDL and followed by RMI-IIOP and RMI.

We have already mentioned that the performance under Windows 2000 was slightly better than under Linux. Figure 4 shows the comparison of weighted average RTTs for Java 2 version 1.3, Windows 2000 and Linux operating system. RMI and RMI-IIOP are on Linux ~20% slower than on Windows 2000, IDL on the other hand only ~4%.

![Figure 4: Comparison of RTTs between Windows 2000 and Linux](image)

**4. Methods that transfer larger data sizes**

The results, presented in previous chapter, are representative for remote method invocations, where parameters and return values are primitive data types. Sometimes, it is also interesting to observe response times for methods that transfer large arrays (or sequences) of data. On one side, such methods are used in particular application domains. On the other side, the tests with larger data sizes place much higher demands on object request brokers. They can show weaknesses in marshalling, multiplexing, dispatching, threading policy, stub and skeleton generation and memory management. They give us a better understanding of the efficiency, such as comparing the throughput with the theoretical network throughput, etc.

Figure 5 shows the average throughput for basic data types for RMI, RMI-IIOP and IDL on Windows 2000 platform. All numbers are shown for Java 2 version 1.3. We can see that there is a difference between the models. RMI shows the best throughput of peak values of almost 30 MBit/s. RMI-IIOP achieved over 26 Mbit/s and IDL almost 24 Mbit/s. The difference between the slowest (IDL) and the fastest (RMI) is in the range of 20%. These results are not bad, particularly for RMI-IIOP and IDL and if we compare them to version 1.2. In versions 1.2 and 1.2.2 RMI achieved results, which are only slightly slower than in version 1.3. IDL, on the other hand, achieved in version 1.2 peak throughputs bellow 8 Mbit/s. We pointed to this in our previous article. Version 1.3 has made considerable improvements. In its beta and pre-release versions, RMI-IIOP has shown low throughputs, too. As you can see, the performance has been dramatically improved.
The situation under Linux operating system is similar, although the gap between RMI on one hand and RMI-IIOP and IDL on the other is larger. IDL shows peak values of almost 38 Mbit/s, while RMI-IIOP and IDL achieve ~27 Mbit/s. This is almost 30% slower. The results are presented in Figure 6.

More interesting is a comparison with Linux Java version 1.2.2. Both RMI and IDL achieved very low throughputs in version 1.2.2. RMI’s peak throughput was 7.5 Mbit/s and IDL’s only 1 Mbit/s. This is a consequence of no just-in-time compiler and green threads, which is the default configuration for Linux Java version 1.2.2, coupled with the ineffectiveness of IDL (as seen under Windows, where the JIT compiler was enabled). The improvements in version 1.3 are therefore even more important.

Figure 7 shows the throughput for methods sending and receiving strings on Windows 2000. We can see, that the results for all three models are practically the same. RMI-IIOP as the fastest model achieved almost 28 Mbit/s and RMI as the slowest almost
26 Mbit/s. The results on Linux are not as uniform. They are shown in Figure 8. IDL, the leader, achieved 35 Mbit/s, RMI and RMI-IIOP around 26 Mbit/s.

Figures 9 and 10 show the response times for methods, which send and receive the testStruct, for Windows and Linux, respectively. Remember, that testStruct is a simple data type in IDL and an object, transferred by value in RMI and RMI-IIOP. RMI-IIOP is in particularly bad position because it has to conform to CORBA Value Types and Objects-by-Value Specification.

On both operating systems the ranking of the models is the same: IDL, RMI, and RMI-IIOP (this time lower numbers mean better results). There are however important differences between Windows 2000 and Linux in terms of response times. IDL, which transferred a data structure only, achieved comparable times on both
operating systems. RMI and RMI-IIOP on the other hand, were ~40% slower on Linux.

Figure 9: Round Trip Times for testStuct on Windows 2000

Figure 10: Round Trip Times for testStuct on Linux

Figures 11 and 12 show the response times for myObject, the object reference, on Windows and Linux, respectively. By this test, the RMI-IIOP and IDL were the fastest and achieved very similar results. RMI was constantly slower. On Windows 2000, RMI was almost 150% slower, and on Linux over 180% slower than RMI-IIOP and IDL. Again, Linux showed considerably slower response times. RMI-IIOP and IDL were ~40% slower and RMI over 60% compared to Windows 2000.
Please be aware, that all results with large data sizes (throughput and RTT) show the best-case scenario. In real-world scenarios the achieved throughput will be lower.

5. Conclusion

Distributed objects and remote method invocations are constituent parts of modern applications. Performance has an important influence on the usability of such applications. Therefore it is useful to know, how well do the three most important distributed models, bundled with Java 2, perform.

In this article we have clarified this question. We have compared RMI, RMI-IIOP, and IDL for Java 2 version 1.3. We have also taken a look at the previous versions. We have measured the performance on Windows 2000 Professional and Linux.

We have learned several things. First of all, in Java 2 version 1.3, there are no order of magnitude differences between the models anymore. In Java 2 version 1.2 IDL performed much worse than RMI. In version 1.3 all three models perform well and show comparable results. Therefore the performance factor alone is not decisive anymore, and this is good news.
Good news is also the performance, achieved by RMI-IIOP. Although RMI-IIOP uses the standardized IIOP protocol for which it has to make additional transformations and mappings, the careful design and implementation of the model, together with careful performance analysis and implementation of several optimizations, have enabled comparable and in some cases even superior performance.

Well, which distributed object model is most suitable for distributed object development in Java? The answer is RMI-IIOP, which is easy to learn and easy to use for Java developers. It has almost the same API as RMI and does not force the developer to learn a separate interface definition language and language mappings. In spite of that it opens the interoperability road to CORBA objects, written in different languages, without a performance penalty.

IDL deserves an honorable mention for its improved performance over version 1.2.x. It offers very good performance and has a permanent place under the developers. It will be used by those developers, which are familiar with CORBA and have used IDL or any other commercial CORBA implementation before. The fact, that IDL now offers good performance, can save costs with third party ORBs, particularly on the client side. We would be completely satisfied with IDL in Java version 1.3, if we would not have difficulties with oneway operations, which resulted in several lockups. We are not sure yet, if this is a bug, but identical code worked fine on IDL 1.2 and on commercial CORBA implementations. Therefore it is not a bad idea to consider a commercial CORBA implementation for the server side objects.

Stays the question, what is the future of RMI. The performance measurements have shown, that RMI is a mature technology, which was able to out pass the other models on some benchmarks. Therefore the only reason for porting existing RMI applications to RMI-IIOP is the need for CORBA interoperability. Due to the already mentioned facts, we would however dissuade from using RMI for new development.

At the end, there is the operating systems question. Good news on this area, too. Whilst in version 1.2.2 Java remote method invocation showed poor performance on Linux, version 1.3 made considerable improvements. On identical hardware configuration, Java on Linux showed comparable performance to Windows 2000. In some cases it was better, but in general slightly slower than on Windows.

Regardless of the distributed object model and the operating system you choose, we wish you and the distributed objects you develop, the best possible performance.

References:

Dr. Matjaz B. Juric is an assistant professor and senior researcher at the University of Maribor, Dr. Ivan Rozman is a full professor at the University of Maribor. Author can be contacted by e-mail at matjaz.juric@uni-mb.si, URL: http://lisa.uni-mb.si/~juric/