Comparison of CORBA and Java RMI
Based on Performance Analysis

Matjaz B. Juric, Ales Zivkovic, Ivan Rozman
Institute of Informatics, Faculty of Electrical Engineering,
Computer and Information Science, University of Maribor
Smetanova 17, SI-2000 Maribor, Slovenia, Europe
E-mail: matjaz.juric@uni-mb.si

Abstract

Distributed object architectures and Java are important for building modern, scalable, web-enabled applications. To meet the requirements of large, mission critical systems they should offer adequate performances especially under heavy multi-client load. This paper is focused on the performance analysis of two distributed object models for use with Java: CORBA and RMI. In the paper we presents performance results based on a real world application for different scenarios that include single client and multi client configurations and different data types and sizes. We evaluate multi threading strategies and analyze code in order to identify the most time consuming methods. We compare the results and give hints and conclusions. We have found out that because of it’s complexity CORBA is slightly slower than RMI in simple scenarios. On the other hand CORBA handles multiple simultaneous clients and larger data sizes better and suffers from far lower performance degradations under heavy client load. The article presents a solid basis for making a decision about the underlying distributed object model.

Key words: Java, CORBA, RMI, distributed objects, performances

1. Introduction

Exponential network growth, global connectivity, new application domains, net-centric operating systems and user interfaces have reached the limits of traditional object oriented programming techniques. The most critical aspects not supported by them are connectivity and interoperability. These two aspects are crucial for building a new generation distributed information systems. Distributed systems require communication between computing entities.

The most promising language for building distributed portable applications capable of executing within a web browser is Java. Java offers basic communication mechanisms with its support for sockets. However, this is a low-level approach. For the new generation information systems development high level approaches are needed. The alternative is the concept, known from procedural languages as remote procedure call (RPC). Paring of RPC concepts and object paradigm results in a distributed object model. For Java there are two suitable distributed object models, Common Object Request Broker Architecture (CORBA) and Remote Method Invocation (RMI).

Both CORBA and RMI hide the communications details of remote method invocations so their basic functionality is similar. But RMI and CORBA were built on different fundaments. RMI has been designed for Java programming language and supports all the details of Java language. It offers the basic functionality of an object request broker (ORB). CORBA on the other hand is a middleware solution defined by the majority of the most important companies in the computer industry [Orfali97]. It is not bound to a particular programming language and offers besides an ORB a rich set of object services, common facilities and domain facilities. In [Juric98b] a feature comparison can be found. It can be seen that although RMI offers some functionality that can not be found in CORBA the later is a more complex architecture.

For a software developer several aspects are important in order to make the right choice of which distributed object model to use. One of them are performances. This paper is focused on the performance analysis of RMI and CORBA/Java. Several testing scenarios that include different data types and single-client and multi-client configurations are defined and results are presented and analyzed. With the use of a profiler tool the most time consuming parts of the code are identified. The testing scenarios are defined so that the results between RMI and CORBA/Java are comparable. With the presented in-depth analysis of CORBA/Java and Java RMI the paper contributes to the understanding of performances of both models.
The paper is organized in eight sections: Section 2 outlines the most important differences between CORBA and Java RMI. Section 3 describes the comparison method and the configuration used for the testing. Sections 4 and 5 present the results of CORBA and RMI performance tests, respectively. Section 6 presents a detailed performance comparison and interprets the results. Section 7 compares the testing method and the results with the related work and Section 8 gives a conclusion.

2. Important differences between CORBA and RMI

Object Management Group’s CORBA is based on object management architecture (OMA) and Core Object Model (COM) [OMAG95]. The five main parts of the architecture are object request broker (ORB) [CORBA98], object services (CORBA services) [COSS97], common facilities (CORBA facilities) [CFA95], domain facilities and application objects [Vinoski97]. CORBA is not bound to a particular programming language. Therefore an interface definition language (IDL) has been introduced. The integral part of the CORBA architecture is the object request broker. CORBA ORB supports two types of method invocation: static (IDL) invocation and dynamic invocation. For remote communication the General Inter-ORB Protocol (GIOP) is defined. The mapping of the GIOP to the TCP/IP protocol is called Internet Inter-ORB Protocol (IIOP).

Remote Method Invocation (RMI) was added to Java in the version 1.1. It offers the basic functionality of the object request broker [RMI97]. Because it was designed for Java it offers some services not found in CORBA [Juric98, Juric98b]. The three independent layers that constitute the RMI system are the stub/skeleton layer, the remote reference layer and the transport layer. For the wire protocol RMI uses Java Object Serialization and HTTP.

Table 1 presents the crucial features supported only by CORBA ORB specification. The most important are language independent wire protocol, dynamic acquiring of object interfaces and the ability to compose and execute method invocations at run-time. They are supported by IIOP, interface repository and dynamic invocation interface, respectively. Other features include different parameter passing modes, persistent naming and persistent object references.

<table>
<thead>
<tr>
<th>Features supported only by CORBA ORB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language independent wire-protocol (IIOP)</td>
</tr>
<tr>
<td>Dynamic acquiring of object interfaces (Interface repository)</td>
</tr>
<tr>
<td>Dynamic method invocations (Dynamic invocation interface)</td>
</tr>
<tr>
<td>Parameter passing modes (in, out, inout)</td>
</tr>
<tr>
<td>Persistent naming</td>
</tr>
<tr>
<td>Persistent object references</td>
</tr>
</tbody>
</table>

Table 1: Features found only in CORBA

In Table 2 the features supported only by RMI are listed. These include dynamic class and stub downloads, object passing by value and URL based object naming.

<table>
<thead>
<tr>
<th>Features found only in RMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic class downloading</td>
</tr>
<tr>
<td>Dynamic stub downloading</td>
</tr>
<tr>
<td>Object passing by value</td>
</tr>
<tr>
<td>URL based object naming</td>
</tr>
</tbody>
</table>

Table 2: Features found only in RMI

CORBA is an older and a more mature technology than Java RMI. First version of CORBA has been released in 1992, the current version is 2.2 and version 3.0 is expected next year. RMI on the other hand is two years old. CORBA is also a better platform for reusing legacy systems [Juric97].

3. Performance comparison

Both CORBA and RMI introduce a certain level of overhead into the method invocations. Therefore one of the goals of the performance comparison was to measure this overhead. The other goal was to investigate the performance degradation under heavy client load. Therefore several scenarios were developed for communication between client and server objects. The requirements for the tests were as follows:
(1) The results from Java RMI and CORBA/Java tests should be comparable.
(2) Single and multiple client scenarios should be simulated.
(3) Different data types should be used.
(4) The hardware equipment that matches typical user environment should be used.

3.1. Testing method
We simulated typical interactions between client and server objects, found in common three-tier applications. Therefore we defined a set of interfaces with methods for Java RMI and CORBA IDL shown in Listing 1 and Listing 2, respectively. As a basis for the tests an Automatic Teller Machine application has been used.

```java
public interface Atm extends java.rmi.Remote {
    … public boolean Working() throws java.rmi.RemoteException;
    public long getAtmNo() throws java.rmi.RemoteException;
    …
}

public interface Account extends java.rmi.Remote {
    … public float getBalance() throws java.rmi.RemoteException;
    public java.lang.String getType() throws java.rmi.RemoteException;
    public double getLimit() throws java.rmi.RemoteException;
    …
}

public interface Card extends java.rmi.Remote {
    … public int getNumber() throws java.rmi.RemoteException;
    …
}
```

Listing 1: Java RMI interfaces used for performance testing

The interfaces were implemented in Java. Attention has been paid to assure that implementations for CORBA and RMI were equivalent. The unavoidable differences were only in the initial convocations to the ORB and a few other details. Therefore the requirement (1) for the tests was met.

```java
interface Atm {
    … boolean Working();
    long long getAtmNo();
    …
};
interface Account {
    … float getBalance();
    string getType();
    wstring getTypew();
    double getLimit();
    …
};
interface Card {
    … long getNumber();
    …
};
```

Listing 2: CORBA IDL interfaces used for performance testing

All the methods returned typed results. They did not accept any parameters and they did not do any processing. The goal was to measure the overhead of the distributed architectures and we wanted to omit any unnecessary influences on the results. The performances were measured for the following data types: integer, long, float, double, boolean and various string sizes, which satisfied the criterion (3).

The server side objects were located on one computer. With the above methods we have covered all the basic data types. You may notice that IDL data types are not the same as Java types. We have used these types to assure consistent mapping from IDL to Java, as shown in Table 3.

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>long</td>
<td>int</td>
</tr>
<tr>
<td>long 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>java.lang.String</td>
</tr>
<tr>
<td>wstring</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>

Table 3: Type mappings from IDL to Java
It is important to understand that Java's `String` class is a null-terminated set of Unicode characters which are 2 bytes long. Native Java does not support 1-byte characters. Although IDL type `string` maps to the Java `String` class, only 1 byte is transferred over the wire. If support for Unicode is needed, then the mapping `wstring` should be used. On the other hand, CORBA objects written in other programming languages typically use `string`. To use the methods of a CORBA object written in Java which uses `wstring`, a conversion in other programs is needed. Therefore we decided to do the performance measurements with both mappings.

On the client side, we designed an applet, which connected to the server objects and invoked the methods. The skeleton of the applet is shown in Listing 3. The two-way static invocation mechanism has been used. To simulate large string transfers, we measured response times for different string sizes. The method `Account.getType()` returned strings of 1, 1000, 2000, 3000, 4000, 5000 and 10000 characters, respectively.

Listing 3: The client-side applet

```java
String acc_type = "1";
my_account.setType(acc_type);
for(int j=0;j<15;j++) {
    long startTime=System.currentTimeMillis();
    for (n=0;n<NO_ITER;n++)
        acc_type=my_account.getType();
    long stopTime=System.currentTimeMillis();
tMessage.append("Elapsed time "+
        (stopTime-startTime)+"\n");
}
```

Time was measured with `System.currentTimeMillis()` method. The method returned elapsed time in milliseconds. To achieve the necessary accuracy, all the invocations were repeated one thousand times. The results reported here are the average values of fifteen repetitions.

For CORBA/Java and Java RMI, the tests were executed in three scenarios:
(a) The server object and the client applet ran on the same computer.
(b) The server object and the client applet ran on two separate computers.
(c) The server object ran on one computer, and the client applets were simultaneously executed from 2, 3, 4, 5, 6, 7, and 8 clients.

The results of (a) and (b) allow the conclusion about the network overhead and the comparison of (b) and (c) shows the performance degradation under heavy client load. While the first result is important for distribution, the second represents typical multi-user interaction. With the described scenarios, the criterion (2) has been satisfied. As you can see in Listing 3, the client applet invoked the methods continually without delays. This does not correspond with typical user interaction, therefore the same results would be achieved with a much larger number of typical clients.

### 3.2. Server-side multithreading strategy

The testing method foresees that multiple clients invoke methods on a single server object. The multithreading strategy that is supported by the server object and the ORB is crucial and has an important impact on the performances. To determine the multithreading strategy used in the scenarios we have performed a simple test. We have defined a server object that had one method. This method delays the program execution for 30 seconds (Listing 4). The method should be invoked by up to eight simulations clients. The execution time should be measured. If the time is around 30 seconds then the server side is capable of handling the requests in parallel. If the execution time is around `no_clients * 30` seconds then the server side serializes the requests.

Listing 4: The server method for determining the multithreading strategy

```java
public void Sec30() {
    System.out.println("Start (30 sec*)");
    try {
        Thread.sleep(30000);
    } catch (Exception e) { ...};
    System.out.println("Stop");
}
```
This test is able to distinguish between the thread-per-servant architecture [Schmidt98] that serializes the requests and architectures that treat the requests concurrently. It is not able to distinguish between several multithreading architectures, but it guarantees at least eight simultaneous requests which is sufficient for the described testing method.

3.3. Software and hardware testbed equipment

The Java source code was compiled and executed within JavaSoft’s Java Development Kit 1.1.4 (JDK), which is the reference platform for Java development. For experiments with CORBA architecture the Visigenic (Inprise) Visibroker for Java 3.0 has been used. The Visibroker is one of the most popular CORBA compliant object request brokers and is integrated into Netscape Navigator 4.0. Therefore it is the most commonly found ORB on the desktop. In certain performance tests with C++ programming language [Gokhale98] it outperformed the Iona Orbix. As a profiler tool the JProbe Profiler 1.1 for KL Group has been used. All the computers used Microsoft Windows NT 4.0 Workstation as their operating system.

The server computer was a Pentium II 233 MHz computer with 64 MB RAM and the clients were Pentiums 200 MHz also with 64 MB RAM. In the today’s Internet applications the bandwidth is crucial. To simulate real world environment we have decided to connect computers into a 10 MBps Ethernet network. The network was free of other traffic. With the described software and hardware configuration the requirement (4) was satisfied.

The reader should be aware that through the article the authors are using the words CORBA and Java/RMI for comparing the performance results. These results apply only to Visibroker for Java 3.0 and JDK 1.1.4 and can not be generalized for other CORBA implementations and other Java Virtual Machines.

4. CORBA performance results

4.1. Single client

In the first scenario a single client applet invoked the methods on the server objects. The tests were done for two configurations: for (a) basic data types and for varying string sizes where mapping to IDL (b1) string and (b2) wstring were used. Two scenarios were included: (1) the server and the client program were located on the same computer and (2) on two different computers.

In Figure 1 the results for basic data types are gathered. The differences in method invocation time between the data types are marginal. In the scenario (1) the average time was 2.17 ms and in the scenario (2) it was 2.30 ms. The network connection had a very small impact on the times, only 6%.

In the second test (b1) the influence of the string size, returned by the method, has been investigated. The results are shown in Figure 2. The overhead of the network grows with the string size from 18% for 1000 bytes string up to 51% for 10000 character string. Similarly as by RMI, the method response time in linearly dependant from the string size. In the first scenario (1) the function is:

\[ t(s) = 0.0018 s + 2.3208 \]

In the network scenario (2) the function is:

\[ t(s) = 0.0029 s + 1.9634 \]
In the third test the wstrings (b2) have been used. The results are shown in Figure 3. The network overhead is around 16%. The linear approximations for single computer and two computers scenarios are:

\[ t(s) = 0.0116 s + 2.6423 \]

and

\[ t(s) = 0.0135 s + 2.1398 \]

4.2. Multiple clients

We have investigated the performance degradation under heavy client load for (a) basic data types and for different string (b1) and wstring (b2) sizes. Figure 4 shows the results for the basic data types.

A single method invocation time grows with the number of clients. The behavior is independent of the data type. By the six concurrent invocations the response time is 2.7 times longer and by eight concurrent
invocations 3.6 times longer than by a single invocation. The exception is the 1 byte wstring data type, where the method invocation time is 2.8 and 4.3 times longer, respectively.

For the string method (b1), the average response time per client is shown in Figure 5. It can be seen, that for the ten times larger string, the method invocation time is prolonged 6.4 times in one client scenario, 4.8 times in four clients scenario and 3.7 times in eight clients scenario. Irrespective of the number of simultaneous clients, the string method invocation time is linearly dependent from the string size.

![Figure 5](image1.png)

Figure 5 CORBA: Multiple clients scenario, different string sizes (b1)

The same analysis was done for the wstring method (b2), where for the ten times larger string you can see 9.1 times longer method invocation in one client scenario, 7.9 times in the four clients scenario and 7.4 times in the eight clients scenario. The average response time for various string size method per client is shown in Figure 6.

![Figure 6](image2.png)

Figure 6 CORBA: Multiple clients scenario, different wstring sizes (b2)

4.3. Multithreading strategy evaluation

When executing the test described in chapter 4.2 we have found out that the Visibroker supports parallel method invocations. The test lasted around 30 seconds regardless of the number of clients. This corresponds with [Visi97] where it is stated that Visibroker supports two thread policies: thread pooling and thread-per-session.
4.4. Code analysis

A profiler has been used for code analysis. To investigate the most time consuming methods the client applet and the server application executing on separate computers have been analyzed. We have decided to analyze the `Account.getType()` method which returns a string. Test examples with IDL string were used. It is interesting to observe the behavior when the string size increases. The majority of time is spend in the methods that handle the communication.

In Table 4 the results for the client side and in Table 5 for the server side are presented. The results in percent show the portion of the whole execution time consumed by a method. On the client side, there were 428 methods involved and on the server side 640 methods.

<table>
<thead>
<tr>
<th>Method name</th>
<th>String size in characters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>.main.</td>
<td>1</td>
</tr>
<tr>
<td>String.&lt;init&gt;(byte[], int, int, int)</td>
<td>1072</td>
</tr>
<tr>
<td>GiopConnectionFactoryImpl.getMessage()</td>
<td>1009</td>
</tr>
<tr>
<td>Socket.getInputStream()</td>
<td>2018</td>
</tr>
<tr>
<td>GiopOutputStreamImpl.write_long(int)</td>
<td>6064</td>
</tr>
<tr>
<td>StringBuffer.append(Object)</td>
<td>9</td>
</tr>
<tr>
<td>Socket.getOutputStream()</td>
<td>5191</td>
</tr>
<tr>
<td>Table 4: CORBA: Client side methods</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method name</th>
<th>String size in characters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>String.getBytes(int, int, byte[], int)</td>
<td>1195</td>
</tr>
<tr>
<td>Hashtable.put(Object, Object)</td>
<td>1082</td>
</tr>
<tr>
<td>GiopInputStreamImpl.read_long()</td>
<td>6085</td>
</tr>
<tr>
<td>GiopOutputStreamImpl.write_long(int)</td>
<td>5167</td>
</tr>
<tr>
<td>Socket.getInputStream()</td>
<td>2017</td>
</tr>
<tr>
<td>Class.forName(String)</td>
<td>47</td>
</tr>
<tr>
<td>SocketInputStream.read(byte[], int, int)</td>
<td>2017</td>
</tr>
<tr>
<td>Table 5: CORBA: Server side methods</td>
<td></td>
</tr>
</tbody>
</table>

5. Java RMI performance results

5.1. Single client

For evaluation of Java RMI the tests were done in the same procedure as for CORBA/Java. Again, two types of results were obtained: (1) the server and the client programs executed on the same computer and (2) the server and the client programs ran on separate network-connected computers. The tests were repeated for (a) basic data types (boolean, integer, long, float, double, 1 character string) and (b) for strings of different sizes.

Figure 7 shows the time of the six methods, needed to return the basic data types (a). The results for the boolean, integer, long, float and double were very close. In the scenario (1) the average time was 1.54 ms. Only the method that returned string took longer, 1.68 ms. In the scenario (2) the average time was 2.11
ms, which gives a 37% performance degradation. The string method took 2.39 ms. Performance degradation was slightly over 42%.

RMI adds a significant overhead to the method invocation. On the same platform the native Java method invocation time for the same methods, which were implemented locally, was on average 400 ns (nanoseconds). However it is important to understand, that RMI enables inter-process communication, that means communication between different Java Virtual Machines (JVMs).

![RMI: Single client scenario, different string sizes (b)](image)

**Figure 8** RMI: Single client scenario, different string sizes (b)

Figure 8 shows the results for varying string sizes (b). The overhead of the two computers scenario (2) over the single computer (1) varies and is around 50%. More interesting is the degradation caused by the string size. In both cases it can be approximated with a linear function. In both cases it can be approximated with a linear function in the form \( t(s) = ks + n \) where \( t \) is time in ms and \( s \) is string size in bytes. In the scenario (1) the function is as follows:

\[
\text{5.} \quad t(s) = 0.0101s + 1.544
\]

For the two computers scenario (2) the function is:

\[
\text{6.} \quad t(s) = 0.0142s + 4.426
\]

5.2. Multiple clients

In the multiple client scenario the goal was to investigate the performance degradation under heavy client load. The tests were done for (a) basic data types and for (b) different string sizes.

![RMI: Multiple clients scenario, basic data types (a)](image)

**Figure 9** RMI: Multiple clients scenario, basic data types (a)

The average response times per client for string method with various string sizes is shown in Figure 10. For example, when the method returns a ten time larger string (10000 characters instead of 1000 characters) the invocation time is 7.5 times larger for one client scenario, 6.3 times for four clients and 5
times for eight clients scenario. Irrespective of the number of clients the response time for string-method invocation is linearly dependent on the string size.

![Graph](image_url)

**Figure 10** RMI: Multiple clients, different string sizes (b)

5.3. **Multithreading strategy evaluation**

The test in chapter 4.2 lead us to the conclusion that RMI supports parallel method invocations on the server side because the test lasted around 30 seconds independently of the number of clients. The RMI Specification [RMI97] guarantees that each method invocation originating from a different client virtual machine will execute in a different thread. This corresponds with our result.

5.4. **Code analysis**

The code analysis was done the same way as by CORBA. In Table 6 the results for the client side are presented. Only the most important methods are listed. In the client applet using RMI there was a total of 40 methods. In Table 7 the results for server side are presented. The server program constituted of 68 methods. Similar as by CORBA code analysis the majority of time was spent in methods that handle the communication.

<table>
<thead>
<tr>
<th>String size in characters</th>
<th>Method name</th>
<th>Calls</th>
<th>1</th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ObjectInputStream.readObject()</td>
<td>1000</td>
<td>6.12%</td>
<td>84.95%</td>
<td>96.25%</td>
<td>97.90%</td>
</tr>
<tr>
<td></td>
<td>.main.</td>
<td>1</td>
<td>21.32%</td>
<td>3.91%</td>
<td>0.91%</td>
<td>0.47%</td>
</tr>
<tr>
<td></td>
<td>UnicastRef.newCall(RemoteObject, Operation[], int, long)</td>
<td>1002</td>
<td>31.41%</td>
<td>3.68%</td>
<td>0.86%</td>
<td>0.44%</td>
</tr>
<tr>
<td></td>
<td>UnicastRef.invoke(RemoteCall)</td>
<td>1002</td>
<td>17.82%</td>
<td>3.22%</td>
<td>0.73%</td>
<td>0.40%</td>
</tr>
<tr>
<td></td>
<td>Naming.lookup(String)</td>
<td>3</td>
<td>11.80%</td>
<td>1.99%</td>
<td>0.48%</td>
<td>0.25%</td>
</tr>
<tr>
<td></td>
<td>UnicastRef done(RemoteCall)</td>
<td>1001</td>
<td>8.96%</td>
<td>1.62%</td>
<td>0.38%</td>
<td>0.19%</td>
</tr>
</tbody>
</table>

**Table 6: RMI: Client side methods**

<table>
<thead>
<tr>
<th>String size in characters</th>
<th>Method name</th>
<th>Calls</th>
<th>1</th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ObjectOutputStream.writeObject(Object)</td>
<td>1000</td>
<td>4.20%</td>
<td>88.40%</td>
<td>97.25%</td>
<td>98.42%</td>
</tr>
<tr>
<td></td>
<td>.TCP Accept-2.</td>
<td>1</td>
<td>34.24%</td>
<td>4.20%</td>
<td>0.96%</td>
<td>0.47%</td>
</tr>
<tr>
<td></td>
<td>StreamRemoteCall.getResultStream(boolean)</td>
<td>1001</td>
<td>27.97%</td>
<td>3.62%</td>
<td>0.90%</td>
<td>0.44%</td>
</tr>
<tr>
<td></td>
<td>StreamRemoteCall.releaseInputStream()</td>
<td>1002</td>
<td>8.72%</td>
<td>1.18%</td>
<td>0.26%</td>
<td>0.13%</td>
</tr>
<tr>
<td></td>
<td>Naming.rebind(String, Remote)</td>
<td>3</td>
<td>7.51%</td>
<td>0.98%</td>
<td>0.21%</td>
<td>0.12%</td>
</tr>
<tr>
<td></td>
<td>.TCP Accept-1.</td>
<td>1</td>
<td>4.90%</td>
<td>0.63%</td>
<td>0.13%</td>
<td>0.08%</td>
</tr>
</tbody>
</table>

**Table 7: RMI: Server side methods**

6. **Interpretation of the results**

6.1. **Single client**

In the single client scenario it can be seen that when using basic data types RMI is faster than CORBA. The difference is especially noticeable when the client and the server are located on the same computer.
For boolean, integer, long, float and double data types CORBA is on average 41% slower. With a one character string, the difference is smaller and is practically the same if using the IDL mapping to string or wstring – CORBA is approximately 31% slower. The disadvantage of CORBA is smaller when the client and the server program are on separate computers. CORBA is then only around 9% slower with basic data types (boolean, integer, long, float and double). For the one character string CORBA is even faster than RMI in this scenario, although only marginal 2.5% regardless if IDL string or wstring was used.

The slower results of CORBA are affected by a much greater complexity of the CORBA architecture. In client-server communication with CORBA 428 methods were involved on the client side and 640 on the server side. RMI required only 40 and 68 methods respectively, which is 10 times less. Fortunately the scenario with the client and the server on the same computer is not as important as the scenario where the client and the server are on separate computers. There the disadvantage of CORBA is much smaller. Also the performance degradation when objects are distributed across a network is much smaller with CORBA architecture and was in our test only 6% compared to the 40% by RMI.

More interesting is the difference for larger strings. When CORBA uses IDL string it is on average 75% faster than RMI when executing on the same computer and 78% faster on separate computers. This result was expected, because IDL char is 1 byte long and Java character is 2 bytes long. When comparing CORBA’s IDL wstring with RMI it can be seen that RMI is 18% faster on single computer, but CORBA is 12% faster when the client and the server are on separate computers. It can be seen that CORBA handles network communication better than RMI. The performance degradation when the client and server are moved to separate computers can be best observed from the coefficients $k$ of the linear approximations. From equations /5/ and /6/ it can be seen that when the string size increases, the degradation with RMI is around 40%, with the CORBA string it is around 61% (equations /1/ and /2/) and with the CORBA wstring it is only 16% (/3/, /4/).

From the code analysis it can be seen that when the string size increased the majority of the time with RMI was spent in methods ObjectOutputStream.writeObject() and ObjectInputStream.readObject() for the server and the client programs, respectively. With CORBA programs, most of the time was spent by methods String.getBytes() and String.<init>(). All these methods are responsible for communication, which is not surprising.

6.2. Multiple clients

An even better picture of the performance issues can be achieved when observing the degradation under a heavy client load. We would like to point out that the testing scenarios were designed so that the clients invoked the methods continuously. For a typical user environment this corresponds to a much larger number of clients. The actual number depends on the average delay between the invocations.

![Figure 11 Actual speed RMI vs. CORBA](image)

Figure 11 shows the actual speed comparison between CORBA and RMI. RMI had an edge over CORBA for up to 4 clients. After that CORBA was faster. Please notice that in single client scenario RMI is faster for basic data types.
The lower complexity of the RMI architecture is nullified with the better multiple client handling of the CORBA architecture. Although both architectures support simultaneous server-side handling of client invocations it is obvious that RMI has less efficient algorithms.

When observing different string sizes, two comparisons should be made. In Figure 12 the performance comparison between CORBA string and RMI/Java string is shown. With the increasing string size and larger number of clients CORBA is over 80% faster than RMI. The comparison between CORBA wstring and RMI/Java string is shown in Figure 13. It can be seen that in most cases CORBA is still faster – up to 33%. On the other hand in some scenarios RMI is up to 20% faster than CORBA.

![Figure 12 CORBA string vs. RMI string](image1)

![Figure 13 CORBA wstring vs. RMI string](image2)

It is obvious that in these scenarios we are faced with a combination of two factors: the ability to efficiently handle multiple simultaneous method invocations and the ability to manage large data (in this case strings). As already mentioned the IDL string is expected to be faster and indeed it is. But it is faster by more than 50% which confirms once again that CORBA handles multiple clients better. The implementation of CORBA we used – Visibroker does not handle the IDL wstring optimally which is probably due to the fact that the wstring specification [CORBA98] has not been finished by the time Visibroker 3.0 was released.

### 6.3. Performance degradation under multiple client load

As a consequence of the factors analyzed in the previous sections the performance degradations of both architectures under multiple client load differ a lot. In Figure 14 the performance degradation for basic data types can be observed. It can be seen that under a heavy client load RMI has larger degradation than CORBA. In the eight clients scenario the degradation is 70%. Both RMI and CORBA support parallel simultaneous multi-client method invocations. Therefore the difference arises from the less optimized RMI code.
Figure 14  Performance degradation under load: basic data types

In Figure 15 the performance degradation for 10000 characters string in the three scenarios is presented. The degradation is slight and is for the eight simultaneous clients lower than 25%. The smallest degradation is achieved by CORBA string, followed by RMI/Java string and CORBA wstring. Again keep in mind that the differences are minor. The degradation of CORBA wstring is only 30% larger than RMI/Java string.

Although IDL string is translated from the 2 bytes representation used by Java into the 1 byte IIOP on-the-wire representation for every request this translation is not time consuming. This is confirmed by the results. On the other hand the IDL wstring is not optimally handled by the Visibroker implementation.

Figure 15  Performance degradation for 10000 bytes string

6.4. Conclusions

First of all it can be seen that neither the tested implementation of CORBA nor RMI in the tested version is considerably faster or slower. But from the test results some conclusions can be derived. In simple scenarios where the number of clients and the amount of data transferred is small both architectures demonstrate comparable results although RMI has an edge over CORBA which is understandable because of the much greater complexity of the CORBA architecture which has to support different programming languages and different platforms. RMI on the other hand is bound to Java and runs inside a Java Virtual Machine. The greater maturity of CORBA can be seen when multiple clients are involved. Then the response times for CORBA were almost always better than for RMI. This is important for the scalability of the applications being developed.

Even more interesting is the comparison of large strings which are a common occurrence in today’s programs. Because of the other programming languages that are supported by CORBA architecture two mappings of Java string to CORBA exist. We have shown that CORBA handles 1 byte character strings very well and is substantially faster than RMI – up to 80%. The 2 bytes wstring is not handled as good as regular string. The results of CORBA and RMI are closer, but CORBA still has an edge over RMI. In the future versions of CORBA the wstring support will certainly be improved. Therefore if there is no need
for Unicode support then string should be used in the applications. Strings should also be used if the connection with legacy systems or objects in other programming languages is planned.

We can conclude that CORBA/Java is more suitable for large fully or partially web-enabled applications where good performances under heavy client load are crucial. Java RMI on the other hand is suitable for smaller fully web-enabled applications where performances especially under multiple client load are not very important.

6.5. Future work

Although the presented results give an insight into the performances of CORBA/Java using the Visibroker 3.0 implementation and Java RMI using the JDK 1.1.4 some questions remain unanswered. First, although we identified the methods that took the majority of the execution time it would be interesting to analyze the source code and suggest improvements. Second, the testing method evaluates as well single client as multiple client scenarios and different data types. Our future work will be on improvements regarding the data types where we will include even larger data sizes in form of arrays, structs and sequences. Third, the presented results reflect only the tested implementations on Windows NT platform. For an in-depth understanding the tests should be repeated with different CORBA implementations and Java Virtual Machines on different platforms, which is what we are also working on. Fourth, we are going to evaluate the influence of the network speed. Therefore we will repeat the tests using networks with different bandwidths. Our goal is to define an ORB performance evaluation model which would be usable with all the most important distributed object models (CORBA, RMI, COM) and would give comparable results.

7. Related work

Existing research in distributed object performance evaluation is limited to the CORBA architecture and C++ programming language. The majority of the work is focused on the latency and scalability investigations, mostly over high-speed networks, where single client and single server configurations are used. The most comprehensive work has been done by authors Douglas C. Schmidt and Aniruddha Gokhale. In [Schmidt95] authors report the performance results from benchmarking sockets and several CORBA implementations over Ethernet and ATM networks. The paper also describes the ACE. The authors showed that the low speed networks can often mask the inefficiency of middleware architectures. Therefore we selected our test cases so that we did not reach the network bandwidth limits. In [Gokhale96b] the authors compared the performance of socked-based communication, RPC, Orbix and ORBeline over ATM network and discovered the sources of overhead. They used a single client and a single server configuration. Our work is based on different platform and extends the results with multiple simultaneous clients scenario. In [Gokhale96] authors measured and explained the overhead of CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface. Because Java RMI does not support dynamic invocations the results are not comparable to our work. In [Gokhale97] and [Gokhale98] 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 performances and they gave an overview of TAO. In [Lo97] 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 context of real-time systems are presented in [Schmidt97], [Gill98] and [Singhai97]. As far as our knowledge our research is the first in-depth performance analysis of Java RMI and CORBA/Java. It provides single client as well as multi client measurements and therefore better reflects real world performances where it is a common occurrence to have multiple clients access a single server.

8. Conclusion

In the paper we have compared two distributed architectures most commonly used with Java. These are RMI and CORBA. Although we have outlined the major differences between the architectures we have also done an in depth quantitative analysis. With the several testing scenarios we have measured performances for methods that returned different data types. We have done the test on a single computer, on two connected computers and under a heavy client load with up to eight simultaneous clients which invoked methods without any delays. We have examined the multi threading strategies. With a profiler tool we have analyzed the code and identified the most time consuming methods. Through the evaluation of the related work we have established that this is, as far as our knowledge, the first in depth analysis of CORBA and RMI in conjunction with Java programming language. Therefore we have provided a solid
foundation to make a responsible decision about which architecture should be used on a particular software project. We have also contributed to the understanding of the performance levels provided by the two distributed object models.

References

[CFA95] Object Management Group, Common Facilities Architecture, Revision 4.0, November 1995
[Juric97] Juric Matjaz Branko, Hericko Marjan, Rozman Ivan: Legacy systems in distributed object architecture, CASSAM - Computer aided software support and maintenance / 5th International conference on re-technologies for information systems, December 1997