Choosing Component Middleware Based on Performance Evaluation
Matjaz B. Juric, 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 models form the basis for distributed component-based applications. This paper is focused on qualitative and quantitative analysis of two most important distributed object models for use with the Java programming language: CORBA and RMI. We compare both models in terms of features, maturity, legacy system support and ease of development. Special attention is paid to performance. We present 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 that because of its 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 degradation under heavy client load. The article presents a solid basis for making a decision about the underlying distributed object model.

Key words: components, distributed objects, CORBA, Java, RMI, performance evaluation

1. Introduction

Global connectivity and exponential network growth provoke architectural changes to the applications. Once monolithic programs are replaced by multi-tier applications which execute in the distributed environment of heterogeneous computers. Each tier is made up of a number of components. The interaction between them is strictly defined and is expressed in terms of interfaces that each component supports. To actually implement the communication a middleware is needed that handles all the details of the communication.

The most prospective programming language for the new-generation applications is Java. For Java there are two suitable middleware solutions, commonly referred to as distributed object models. They are 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. They both support heterogeneous environments.

For a software developer, several aspects are important in order to make the right choice concerning which distributed object model to use. Traditionally one of the most important criteria is performance. The right choice can not be made without a detailed analysis of both models. As far as we know there are no in-depth analyses of the distributed object models and Java language. In this paper a detailed analysis of CORBA/Java and Java RMI can be found. Special attention is paid to performance evaluation. Several testing scenarios that include different data types and single-client and multi-client configurations are defined and the 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 features and performances of both models.

The paper is divided into eight sections: in Sections 2 and 3 a brief outline of CORBA and Java RMI can be found. Section 4 outlines the differences between them. Section 5 describes the comparison method and the configuration used for testing. Sections 6 presents the results of CORBA and RMI performance tests. In Section 7 a detailed performance comparison and the interpretation of the results can be found. Section 8 gives a conclusion.
2. Overview of the CORBA Architecture

Object Management Group’s CORBA is the only component middleware that is as well platform as language independent. It is based on object management architecture (OMA) and Core Object Model (COM)\(^1\). In Figure 1 the five main parts of the architecture\(^2\) are presented:

- **Object request broker** (ORB) is the integral component of the architecture. It hides all the details of the communication between the two objects – the client and the server object. ORB is the object bus.
- **Object services** (CORBAservices) define the system level object frameworks that widen the range of basic functions of the ORB with services such as naming, event, life-cycle, transaction, relation, etc\(^3\).
- **Common facilities** (CORBAfacilities) define the horizontal application frameworks that are used by application objects\(^4\).
- **Domain facilities** define application frameworks for different domains such as healthcare, financial institutions, manufacturing, etc.
- **Application objects** are the applications actually developed by the developers.

![Figure 1](https://example.com/figure1.png)  
**The overview of the CORBA platform**

CORBA is not bound to a particular programming language. To overcome the details of the programming languages two important aspect were introduced:

- The object’s interface was separated from it’s implementation. The interface specifies all the public methods and attributes.
- An interface definition language (IDL) has been introduced. The IDL is used for interface specification in a language independent way.

It is important to understand that IDL is used only for interface definition. For implementation traditional programming languages are used. Therefore a mapping from IDL to the programming language has to be defined. Currently mappings for C, C++, Smalltalk, Java, Ada and COBOL are standardized.

The integral part of the CORBA architecture is the object request broker. On Figure 2 the structure of interfaces of a typical CORBA compliant object request is shown.

![Figure 2](https://example.com/figure2.png)  
**The structure of the object request broker**

The client and the server are two CORBA objects which communicate by means of method invocation. The client invokes a method on the remote server with a simple method invocation, i.e., `object.method(args)`. The servers returns the result as a return value or through arguments.
ORB hides the object location, implementation, execution state and communication mechanisms. CORBA ORB supports two types of method invocation:

- Static (IDL) invocation is used when the object has compile-time knowledge about other objects. The client uses the IDL stub and the server static IDL skeleton.
- Dynamic invocation enables communication between objects without compile-time knowledge of methods. Dynamic invocation interface and dynamic skeleton interface are used by the client and the server, respectively. The information about the object’s interfaces is stored in the interface repository.

The ORB interface is an abstract interface that hides implementation details of an ORB. The object adapter associates a server and the ORB and maps incoming request to the appropriate operations. ORB core is responsible for the communication between the client and the server. For remote communication the General Inter-ORB Protocol (GIOP) should be used. GIOP specifies a high-level protocol and is independent of the underlying transport protocol. The mapping of the GIOP to the TCP/IP protocol is called Internet Inter-ORB Protocol (IIOP).

3. Overview of the Java RMI

In Java component interaction between different Java Virtual Machines (JVM) is handled by a native distributed model called Remote Method Invocation (RMI). RMI was added to Java with the version 1.1. It offers the basic functionality of the object request broker and shares the basic concepts with CORBA. Because it was designed for Java it offers some services not found in CORBA.

Similar to CORBA, RMI utilizes strict separation of the interfaces from the implementation. Therefore a construct interface has been introduced. Because RMI is bound to Java, interfaces are specified in the Java language. Figure 3 shows the three independent layers that constitute the RMI system. These three layers are:

- The stub/skeleton layer is the interface between the application layer and the rest of the RMI system. A stub for a remote object is the client-side proxy which forwards the request to the actual remote object. A skeleton is a server-side entity which dispatches calls to the actual object.
- The remote reference layer is responsible for carrying out the semantics of the invocation and sits on top of the low-level transport layer. It has the client-side and the server-side components.
- The transport layer is responsible for the setup and management of the connection and dispatching the requests to the remote objects within the transport layer’s address space.

![Figure 3 The Java RMI system architecture](image)

The RMI implements a reference counting garbage collection algorithm similar to Modula-3’s Network Objects. With dynamic class loading the classes required to handle method invocations can be loaded at runtime. RMI is not compatible with CORBA’s IIOP protocol. For the wire protocol RMI uses:

- Java Object Serialization for call marshaling and returning data and
- HTTP for posting remote method invocations and obtaining the return data.

4. CORBA versus RMI

In several cases the usage domain of CORBA and RMI overlaps. In the software development process the decision about the underlying distributed component architecture should be made before the implementation phase can begin. The five most important decision criteria are:

- Features.
- Performances and scalability.
- Maturity.
- Support for legacy systems.
4.1. Feature comparison

It has already been stated that CORBA is much more than just an object request broker. It is a complete distributed object platform with support for different programming languages and several services and facilities not found in RMI. Therefore from now on we will compare RMI only with the CORBA object request broker (ORB).

Table 1 presents the crucial features supported only by CORBA ORB. 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>Feature</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. Because RMI was designed for Java from ground up, it supports some features not found in CORBA/Java combination. These include dynamic class and stub downloads, object passing by value and URL based object naming.

<table>
<thead>
<tr>
<th>Feature</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

4.2. Maturity

CORBA is an older and a far more mature technology than Java RMI. The first version of CORBA was released in 1992, the current version is 2.2 and version 3.0 is expected next year. CORBA is widely used by leading companies and has established itself as a mature, secure and scalable architecture. RMI on the other hand is two years old. It is not widely used in mission critical applications yet.

With support for IIOP and several programming languages, CORBA is suitable for building a company’s backbone. RMI on the other hand uses it’s own wire protocol and supports only Java programming language. CORBA supports shared, unshared and per-method activation modes. It supports implementation repository and smart proxies for effective load balancing and fault tolerance. RMI in it’s first version lacks support for such advanced features.

4.3. Support for legacy systems

A common way of reusing legacy applications in a distributed object architecture is through the use of object wrappers. A legacy system is any system that, regardless of age or architecture has existing code and is still useful and in use today. In the context of distributed object architecture, this definition includes not only traditional mainframe-based systems but also systems written in C, C++ or other languages, PC-based and client/server systems and personal productivity tools. Therefore the support for legacy applications can be crucial.

Object wrappers are a natural way of integrating legacy systems with each other and with new systems. They provide access to legacy systems through an encapsulation layer. Once wrapped, legacy systems can participate in distributed object environments using object request brokers.

For effective wrapper building support for different programming languages and different platforms is needed. The most welcome is the ORB’s native support for the programming language of the legacy
system. RMI supports multiple platforms (in fact it is platform independent) but connecting legacy code
with Java language can be painful. CORBA on the other hand supports multiple platforms and several
programming languages and has an edge over Java RMI.

### 4.4. Learning curve and ease of development

RMI has been designed for Java only. Therefore it integrates into the Java environment more smoothly
than CORBA does. There is no need for a separate interface definition language. CORBA on the other
hand offers functionality that cannot be found in RMI. Mastering this functionality requires more
learning. When developing CORBA/Java applications mapping between the data types should be defined.
Therefore we can conclude that CORBA is not as easy to use as RMI.

### 5. Performance Comparison Method

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.

#### 5.1. 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 {
  boolean Working() throws java.rmi.RemoteException;
  long getAtmNo() throws java.rmi.RemoteException;
  ...
}
public interface Account extends java.rmi.Remote {
  float getBalance() throws java.rmi.RemoteException;
  String getType() throws java.rmi.RemoteException;
  double getLimit() throws java.rmi.RemoteException;
  ...
}
public interface Card extends java.rmi.Remote {
  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 insure that implementations for
CORBA and RMI were equal. The unavoidable differences were only in the initial convocations to the
ORB and a few other details. Therefore requirement (1) for the tests was met.

```java
interface Atm {
  boolean Working();
  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 nor did they carry out 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.lang.Strings are null terminated sets of Unicode characters which are 2 bytes long. Native Java does not support 1 byte characters. Although IDL type string maps to the java.lang.String 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 conversion to 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 have measured response times for different string sizes. The method Account.getType() returned strings of 1, 1000, 2000, 3000, 4000, 5000 and 10000 characters, respectively.

```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");
}
```

Listing 3: The client-side applet

Time was measured with the System.currentTimeMillis() method. The method returned elapsed time in milliseconds. To achieve the necessary level of 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, the client applets were simultaneously executed from 2, 3, 4, 5, 6, 7 and 8 clients.

The results of (a) and (b) lead to 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.
5.2. 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 simultaneous 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. Otherwise the server side serializes the requests.

```java
... public void Sec30() {  
    System.out.println("Start (30 sec)");  
    try {  
        Thread.sleep(30000);  
    } catch (Exception e) { ...};  
    System.out.println("Stop");  
} ...
```

Listing 4: The server method for determining the multithreading strategy

This test is able to distinguish between the thread-per-servant architecture 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.

5.3. 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 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 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 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 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 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.

6. Performance Measurements

6.1. 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 (boolean, integer, long, float, double, 1 character string) and for different string sizes (b). Two scenarios were included: (1) the server and the client program were located on the same computer and (2) on two different computers.

As well CORBA as RMI add 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 both CORBA and RMI enable inter-process communication, that means communication between different Java Virtual Machines (JVMs).

In Figure 4 the results for basic data types are gathered. The differences in method invocation time between the data types are marginal. In scenario (1) the average time was 2.17 ms for CORBA and 1.54 ms for RMI. In scenario (2) it was 2.30 ms for CORBA and 2.11 ms for RMI. The network connection influenced the times, 37% for RMI and only 6% for CORBA.
In the second test (b) the influence of the string size, returned by the method was investigated. The results are shown in Figure 5. For CORBA string the overhead of the network grows with the string size from 18% for 1000 bytes string up to 51% for 10000 character string. For CORBA wstring the overhead is around 16% regardless of wstring size. For RMI the overhead is dependant of string size and is around 50%.

In all three cases the method response time is linearly dependant on the string size. 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. For CORBA string in the first scenario (1) the function is:

$1/\quad t(s) = 0.0018 s + 2.3208$

In the network scenario (2) the function is:

$2/\quad t(s) = 0.0029 s + 1.9634$

For CORBA wstring the functions are respectively:

$3/\quad t(s) = 0.0116 s + 2.6423$

$4/\quad t(s) = 0.0135 s + 2.1398$

For RMI string the functions are:

$5/\quad t(s) = 0.0101 s + 1.544$

$6/\quad t(s) = 0.0142 s + 4.426$
6.2. Multiple clients
We have investigated the performance degradation under heavy client load for (a) basic data types and for different string (b) sizes. Figure 6 shows performance degradation for experiment (a). The method invocation time per client grows with the number of clients. Data types boolean, integer, long, float, double and string behave similarly, therefore the average values are presented. The performance degradation for RMI is larger than for CORBA.

![Multiple client scenario, average basic data types (a)](image)

Figure 6    Multiple client scenario, basic data types (a)

For the string method (b), the average response time per client is shown in Figure 7. Irrespective of the number of simultaneous clients, the string method invocation time is linearly dependant from the string size. For the CORBA string it can be seen for example that for the string ten times larger, the method invocation time is prolonged 6.4 times in one client scenario, 4.8 times in the four client scenario and 3.7 times in the eight client scenario.
For RMI, when the method returns a string ten times larger (10000 characters instead of 1000 characters) the invocation time is 7.5 times larger for the one client scenario, 6.3 times for four client and 5 times for the eight client scenario. For CORBA wstring which was ten times larger you can see 9.1 times longer method invocation in one client scenario, 7.9 times in the four client scenario and 7.4 times in eight client scenario.

6.3. Multithreading strategy evaluation

When executing the test described in chapter 4.2 we found that the Visibroker supports parallel method invocations. This corresponds with\(^6\) where it is stated that Visibroker supports two thread policies: thread pooling and thread-per-session.

The same test leads us to the conclusion that RMI supports parallel method invocations on the server side because the test lasted around 30 seconds independent of the number of clients. The RMI Specification\(^5\) guarantees that each method invocation originating from a different client virtual machine will execute in a different thread. This corresponds with our result too.

6.4. Code analysis

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

In Table 4 the results for the CORBA client side and in Table 5 for the CORBA server side are presented. The results in percentages 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.

![Figure 7 Multiple clients scenario, different string sizes (b)](image)

Figure 7 Multiple clients scenario, different string sizes (b)

**Table 4: CORBA: Client side methods**

<table>
<thead>
<tr>
<th>Method name</th>
<th>Calls</th>
<th>1</th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>String.&lt;init&gt;(byte[], int, int, int)</td>
<td>5</td>
<td>2,16%</td>
<td>61,18%</td>
<td>87,51%</td>
<td>93,06%</td>
</tr>
<tr>
<td>Hashtable.put(Object, Object)</td>
<td>5</td>
<td>10,01%</td>
<td>3,65%</td>
<td>1,05%</td>
<td>0,56%</td>
</tr>
<tr>
<td>GiopInputStreamImpl.read_long()</td>
<td>5</td>
<td>5,47%</td>
<td>1,99%</td>
<td>0,58%</td>
<td>0,31%</td>
</tr>
<tr>
<td>Class.forName(String)</td>
<td>5</td>
<td>3,49%</td>
<td>0,97%</td>
<td>0,28%</td>
<td>0,20%</td>
</tr>
<tr>
<td>GiopAdapterThread.doRequest(GiopMessage)</td>
<td>5</td>
<td>3,35%</td>
<td>0,92%</td>
<td>0,05%</td>
<td>0,05%</td>
</tr>
<tr>
<td>GiopInputStream.read(byte[], int, int)</td>
<td>5</td>
<td>3,35%</td>
<td>1,47%</td>
<td>0,47%</td>
<td>0,27%</td>
</tr>
<tr>
<td>Socket.getInputStream()</td>
<td>5</td>
<td>3,40%</td>
<td>1,47%</td>
<td>0,50%</td>
<td>0,61%</td>
</tr>
<tr>
<td>StringBuffer.append(Object)</td>
<td>5</td>
<td>3,40%</td>
<td>1,47%</td>
<td>0,50%</td>
<td>0,61%</td>
</tr>
</tbody>
</table>

**Table 5: CORBA: Server side methods**

<table>
<thead>
<tr>
<th>Method name</th>
<th>Calls</th>
<th>1</th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>String.getBytes(int, int, byte[], int)</td>
<td>5</td>
<td>2,16%</td>
<td>61,18%</td>
<td>87,51%</td>
<td>93,06%</td>
</tr>
<tr>
<td>Hashtable.put(Object, Object)</td>
<td>5</td>
<td>10,01%</td>
<td>3,65%</td>
<td>1,05%</td>
<td>0,56%</td>
</tr>
<tr>
<td>GiopInputStreamImpl.read_long()</td>
<td>5</td>
<td>5,47%</td>
<td>1,99%</td>
<td>0,58%</td>
<td>0,31%</td>
</tr>
<tr>
<td>Class.forName(String)</td>
<td>5</td>
<td>3,49%</td>
<td>0,97%</td>
<td>0,28%</td>
<td>0,20%</td>
</tr>
<tr>
<td>SocketInputStream.read(byte[], int, int)</td>
<td>5</td>
<td>3,35%</td>
<td>0,92%</td>
<td>0,05%</td>
<td>0,05%</td>
</tr>
<tr>
<td>GiopAdapterThread.doRequest(GiopMessage)</td>
<td>5</td>
<td>3,35%</td>
<td>1,47%</td>
<td>0,47%</td>
<td>0,27%</td>
</tr>
</tbody>
</table>
In Table 6 the results for the RMI 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 RMI server side are presented. The server program constituted 68 methods. Similar to CORBA the majority of time was spent in methods that handle the communication.

<table>
<thead>
<tr>
<th>Method name</th>
<th>Calls</th>
<th>1</th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<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>.main.</td>
<td>1</td>
<td>21.32%</td>
<td>3.69%</td>
<td>0.91%</td>
<td>0.47%</td>
</tr>
<tr>
<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>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>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>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>Method name</th>
<th>Calls</th>
<th>1</th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<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>.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>StreamRemoteCall.getResultStream(boolean)</td>
<td>1001</td>
<td>27.97%</td>
<td>3.84%</td>
<td>0.90%</td>
<td>0.44%</td>
</tr>
<tr>
<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>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>.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

7. Interpretation of the Results

7.1. 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 as if using the IDL mapping to string or wstring – CORBA is approximately 31% slower. The disadvantage of CORBA is lessened 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 marginally 2.5% regardless of either IDL string or wstring was used.

The slower results of CORBA are affected by the 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 executed 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 a 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 time with RMI was spent in methods ObjectOutputSteam.writeObject() and ObjectInputSteam.readObject() for the server and the client programs, respectively. With CORBA programs, most of the
time was used by methods `String.getBytes()` and `String.<init>()`. All these methods are responsible for communication, which is not surprising.

### 7.2. 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 8 Actual speed RMI vs. CORBA](image)

Figure 8 shows the actual speed comparison between CORBA and RMI. RMI had the edge over CORBA for up to 4 clients. After that CORBA was faster. Please notice that in the single client scenario RMI is faster for basic data types.

The lower complexity of the RMI architecture is nullified by the superior 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 9 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 10. 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 9 CORBA string vs. RMI string](image)

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 stated 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 has not been finished by the time Visibroker 3.0 was released.

Figure 10  CORBA wstring vs. RMI string

7.3. 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 great deal. In Figure 11 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 11  Performance degradation under load: basic data types

Figure 12  Performance degradation for 10000 bytes string
In Figure 12 the performance degradation for the 10000 character 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 bear 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.

8. Conclusion

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. 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 applications. 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 well 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.

In this paper we have made a comparison of the two most important distributed object middleware architectures for Java. We have compared several important aspects such as features, maturity, support for legacy systems and ease of learning and ease of use. We have paid special attention to the in-depth quantitative analysis. With several testing scenarios we have measured performances for methods that returned different data types. We have carried out 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. As far as we know this is 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 performance levels provided by the two distributed object models.

References
4. Object Management Group, Common Facilities Architecture, Revision 4.0, November 1995