Java and Distributed Object Models: An Analysis


*University of Maribor, Faculty of Electrical Engineering and Computer Science, Smetanova 17, SI-2000 Maribor, Slovenia, Europe
**University of Ljubljana, Faculty of Computer Science, SI-1000 Ljubljana, Slovenia
E-mail: matjaz.juric@uni-mb.si

Abstract

Java has an important role in building distributed object oriented web enabled applications. In the article an analysis of two distributed object models in context of Java language is presented. Several aspects of RMI and CORBA such as features, maturity, support for legacy systems, learning curve and ease of development are compared. A special emphasis is given to the performances. Different testing scenarios give a complete overview about real world performances of both architectures. Based on the comparison results, recommendations for selecting the most appropriate architecture for a given problem domain are presented. Therefore the paper contributes to the understanding of the distributed object architectures and to the study of Java RMI and CORBA performances.

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

1. Introduction

Java has established itself as a modern object oriented programming language especially suitable for building web enabled applications. Modern web applications are commonly build out of components or distributed objects and therefore a special demand is placed to connectivity and interoperability. Java offers basic communication mechanisms with its support for sockets. However, this is a low-level approach. For new generation of application 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).

Although CORBA and RMI offer the same basic functionality – hiding the communications details of remote method invocations - they were built on different fundamentals. The choice of the distributed object model should be made in an early stage of the software development project. The transition from one model to the other is painful and has bad impact on the costs and the schedule of the project. The right choice can not be made without a detailed analysis of both models. Traditionally performance issues have a high impact on the decision. To our knowledge there are no in depth performance analyses of RMI and Java/CORBA. In [Gokhale96], [Gokhale97], [Gokhale98], and [Lo97] performance measurements between a client and a server object over high speed ATM networks can be found. CORBA with C++ programming language has been used.

In this paper a detailed analysis of CORBA and RMI can be found. A special attention has been paid to the performance issues where we have made a series of experiments and tests which give an insight into the performance levels that can be expected from Java applications when used in conjunction with CORBA on one and RMI on the other hand. In Section 2 an overview of the CORBA architecture is given and in Section 3 the Java RMI model is outlined. In the next section a feature comparison is made with emphasis on Java development. The next section outlines the comparison method and the configuration used for testing. Sections 6 and 7 present the results of CORBA and RMI performance tests, respectively. A detailed performance comparison and interpretation of the results is given at the end.

2. Overview of the CORBA architecture

Object Management Group’s CORBA is the most important middleware project ever undertaken by industry [Orfali97]. It is based on object management architecture (OMA) and Core Object Model (COM) [OMAG95]. The integral part of the CORBA architecture is the object request broker. In Figure 1 the structure of a typical CORBA compliant object request is shown. In addition to the ORB the CORBA specification [CORBA98] defines object services, common facilities and domain facilities.

CORBA is not bound to a particular programming language. To overcome the details of the programming languages two important aspects were introduced. The object’s interface was separated from its implementation and an interface definition language (IDL) has been introduced. It is important to understand that IDL is used only for interface definition.
For implementation traditional programming languages are used. Therefore mappings from IDL to the programming languages have been defined. Currently mappings for C, C++, Smalltalk, Java, Ada and COBOL are standardized [CORBA98].

The client and the server are CORBA objects which communicate in means of method invocation. The client invokes a method on the remote server with a simple method invocation, i.e., object.method(args). The server returns the result as a return value or through arguments. ORB interface is an abstract interface that hides implementation details of an ORB. 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 usually the General Inter-ORB Protocol (GIOP) is 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

Java programming language lacks the ability of easy communication between objects in different Java Virtual Machines (JVM). Although Java can be used with the industry standard distributed object architecture CORBA, a native distributed model called Remote Method Invocation (RMI) was added to Java in the version 1.1. RMI offers the basic functionality of the object request broker and shares the basic concepts with CORBA. RMI utilizes strict separation of the interfaces from the implementation. Because RMI is bound to Java, interfaces are specified in the Java language. Figure 2 shows the three independent layers that constitute the RMI system. These three layers are [RMI97]:

- The stub/skeleton layer is the interface between the application layer and the rest of the RMI system.
- The remote reference layer is responsible for carrying out the semantics of the invocation and sits on top of the low-level transport layer.
- The transport layer is responsible for setup and management of the connection and dispatching the requests to the remote objects within transport layer’s address space.

RMI implements reference counting garbage collection algorithm similar to Modula-3’s Network Objects [Birell94]. 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, although plans have been announced to make them compatible. 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 object architecture should be made before implementation phase can begin. The five most important decision criteria are features, performances and scalability, maturity, support for legacy systems and learning curve as well as ease of development.

We will compare RMI only with CORBA object request broker (ORB), although CORBA is much more than just an object request broker [CROSS97, CFA95]. 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. Other features include different parameter passing modes, persistent naming and persistent object references.
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. It can be seen that CORBA is a more complex architecture with more features.

<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>
<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. CORBA is widely used by the leading companies and has established itself as a mature, secure and scalable architecture. RMI on the other hand is not even two years old. Until now it has not been widely used in mission critical applications.

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.

A common way of reusing legacy applications in a distributed object architecture is through object wrappers [Juric97]. A legacy system is any system that, regardless of age or architecture has existing code and is still useful and in use today. Object wrappers 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.

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 and dealing with remote objects is nearly the same as dealing with local ones. CORBA is more complex and therefore requires more learning and is harder to use.

5. Performance comparison

The goal of the performance comparison was to measure the overhead CORBA/Java and Java RMI introduced into method invocation and 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. Real world usage patterns 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, which we developed for one of our banks, has been used. With the use of such an application we satisfied the criterion (2).

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.

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). To assure consistent mapping from IDL to Java we have used the type mappings shown in Table 3.

```
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**

```
Listing 2: CORBA IDL interfaces used for performance testing
```

```java
public interface Atm {
    boolean Working();
    long getAtmNo();
}

public interface Account {
    float getBalance();
    wstring getType();
    double getLimit();
}

public interface Card {
    long getNumber();
}
```

**Listing 2: CORBA IDL interfaces used for performance testing**

It is important to understand that `java.lang.String` s 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` a conversion in other programs is needed. Therefore we decided to do the performance measurements with both mappings.

<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**

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 bytes, respectively.

```
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 `System.currentTimeMillis()` method, which 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, 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. 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. 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 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 from 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.

6. Java RMI performance results
In the first scenario a single client applet invoked the methods on the server objects. The tests were done for two configurations: (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 3 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%.

Figure 4 shows the results for varying string sizes (b). The overhead of the two computers scenario (2) over the single computer (1) 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 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:

\[
\text{Single computer} (1) \quad t(s) = 0.0101 s + 1.544
\]

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

\[
\text{Two computers} (2) \quad t(s) = 0.0142 s + 4.426
\]

In the multiple client scenario the goal was to investigate the performance degradation under heavy client load. The tests were done for basic data types and the results are shown in Figure 5. The method invocation time per client grows with the number of clients. By six clients the average invocation takes more than three times longer and by eight clients more than five times longer than by a single client. Data types boolean, integer, long, float and double behave similar, only the one-character string shows marginally larger times.
7. CORBA performance results

For evaluation of CORBA/Java the tests were done in the same procedure as for Java RMI. Again, two types of results were obtained, for (a) basic data types and for varying string sizes where mapping to IDL (b1) string and (b2) wstring were used. The performances were evaluated for two scenarios: (1) the server and the client program were located on the same computer and (2) on two different computers.

In Figure 6 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 7. The overhead of the network grows with the string size from 18% for 1000 bytes string up to 51% for 10000 bytes 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 8. The network overhead is around 16%. The linear approximations for single computer and two computers scenarios are:

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

---

**Figure 5** RMI: Multiple clients scenario, basic data types

**Figure 6** CORBA: Single client, basic data types (a)

**Figure 7** CORBA: Single client, different string sizes (b1)

**Figure 8** CORBA: Single cl., different wstring sizes (b2)
and

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

We have investigated the performance degradation under heavy client load for basic data types. Figure 9 shows the results. 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.

**Figure 9** CORBA: Multiple clients scenario, basic data types

### 8. Interpretation of the results

In the single client scenario it can be seen that on 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. By the 1 byte string, the difference is smaller and is regardless of the IDL mapping to string or wstring practically the same – 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 on basic data types (boolean, integer, long, float and double). For the 1 byte string CORBA is even faster than RMI in this scenario, although only marginal 2.5% regardless of IDL string or wstring being used. The slower results of CORBA are effected by a much greater complexity of the CORBA architecture. Fortunately the scenario with the client and the server on the same computer is not as important as the scenario where client and server are on separate computers. There the disadvantage of CORBA is much smaller. Also the performance degradation when objects are distributed across 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 is expected, because IDL string is 1 byte long and Java string 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 client and 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 on separate computers can be best observed from the coefficients \( k \) of the linear approximations. From equations /1/ and /2/ it can be seen that when the string size increases, by RMI the degradation is around 40%, by CORBA string it is around 61% (equations /3/ and /4/) and by CORBA wstring it is only 16% (/5/, /6/).

An even better picture of the performance issues can be achieved when observing the degradation under heavy client load. We would like to point out that the testing scenarios were designed so that the clients invoked 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.

In Figure 10 the performance degradation for basic data types can be observed. It can be seen that under heavy client load RMI has larger degradation than CORBA. In the eight clients scenario the degradation is 70% larger. In Figure 11 the actual speed comparison between CORBA and RMI is shown. In the actual speed of the test examples 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.

![Performance degradation under multiple client load](image1)

![Actual speed RMI vs. CORBA](image2)

It can be seen that neither CORBA nor RMI is essentially 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 which is important for the scalability of the programs being developed.

Even more interesting is the comparison of large strings which are a common occurrence in today’s programs. We have shown that CORBA handles 1 byte string very well and is a lot faster than RMI – up to 80%. The 2 bytes wstring is not handled as good as string is. The results of CORBA and RMI are closer. In the future versions of CORBA the wstring support will certainly be improved. Therefore if there is no need for Unicode support string should be used in applications. String should also be used if the connection with legacy systems or objects in other programming languages is planned.

9. Conclusion

In the paper we have evaluated the role of Java language in building distributed applications. Therefore we have compared two distributed architectures most commonly used with Java, RMI and CORBA. We have done a qualitative analysis by comparing several important aspects like features, maturity, support for legacy systems and ease of learning and ease of use. 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 completed the measurements on single computer, on two connected computers and under a heavy client load with up to 8 simultaneous clients which invoked methods without delays.

We have provided a solid foundation to make a substantiative decision on which architecture should be used on a particular software project. The general conclusion of this paper is that CORBA/Java is more suitable for large fully or partially web-enabled applications where legacy support is needed and good performances under heavy client load are crucial. Java RMI on the other hand is suitable for smaller fully web-enabled applications where legacy support can be managed by already existing bridges or is not necessary at all and where the ease of learning and the ease of use are more critical than performances are.
References


