Java and distributed object models are important for building modern, scalable, interoperable applications. This paper is focused on the performance analysis, comparison and optimization of the most important distributed object models for Java: RMI (Remote Method Invocation) and RMI-IIOP (Remote Method Invocation over Internet Inter-ORB Protocol). The paper presents the following contributions to the research on distributed object performance. First, a detailed performance analysis of both models is provided with the in-depth comparison. These results help to understand how the models perform. Second, an overhead analysis and the bottleneck identification is presented with the explanation why there are differences in performance. Third, possible optimizations and the results for performance improved post-beta RMI-IIOP versions are presented. These show considerably better performance in all areas compared to the original beta release, with RMI-IIOP having equivalent or better performance to RMI in almost all cases.

Keywords: Java, RMI, performance, IIOP

1. Introduction

Java has established as the most pervasive object oriented programming language. It has introduced several new concepts and is inseparably connected to the networking computing paradigm. Therefore Java had to support distributed applications in a easy, high-level way that would hide the details of remote communication from the developer. This necessity has been emphasized by the introduction of reusable components such as JavaBeans and Enterprise JavaBeans and the global shift from developing applications from scratch to integrating applications using components.
In its first version Java provided support for sockets only. Although sockets allow easy communication between computing entities this approach is low-level and too time consuming to be acceptable for modern application development. Therefore in version 1.1 Java has been enhanced with a native Java Remote Method Invocation (RMI) mechanism that fulfilled the stated needs. RMI provides transparent remote method invocation between objects executing in different Java Virtual Machines (JVM). It hides all the details of remote communication such as locating the remote object, transferring the request and returning the result [1]. RMI has been designed for Java only and is tightly integrated with JVM.

RMI addresses the same problem domain that is covered by the activities of Object Management Group (OMG). Under the auspices of OMG there have been increasing efforts to define a standard software infrastructure that permits seamless application integration through heterogeneous environments [2]. These efforts have resulted in a standard software infrastructure known as Common Object Request Broker Architecture (CORBA). An implementation of the CORBA specification has been included in Java 2 (version 1.2) under the name Java IDL (Interface Definition Language). Activities have been initiated to merge RMI with CORBA. Therefore RMI is enhanced to work with the CORBA standardized Internet InterORB Protocol (IIOP) and is known as RMI-IIOP.

Selecting the appropriate distributed object model for the target application domain is a multi-criteria decision problem. Usually one of the most important criteria is performance [3]. The performance achieved through distributed object models may not be as good as when using low-level approaches. The main reasons are marshalling and demarshalling overhead, demultiplexing and dispatching, data copying and additional remote invocations for resolving object references.

Evaluating the performance of distributed object models is a difficult task. The review of related work has shown that there is no standardized or commonly accepted method for performance assessment of distributed object architectures. The research on performance is limited mostly to the CORBA architecture and C++ programming language. The majority of the work is focused on latency and scalability investigations, mostly over high-speed networks, where single client and single server configurations are used. In [5] the authors report the performance results from benchmarking sockets and several CORBA implementations over
Ethernet and ATM networks. In [6] the authors compared the performance of socket-based communication, RPC (Remote Procedure Call), Orbix and ORBeline over ATM network and discovered the sources of overhead. They used a single client and a single server configuration. In [7] the authors measured and explained the overhead of CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface. In [8] and [9] the authors systematically analyzed the latency and scalability of Visibroker and Orbix and revealed the sources of overhead. Again they used a single client and server configuration over ATM network. They also described techniques to improve the performance and they gave an overview of Tao. In [10] the author described the implementation of a low overhead ORB. He presented some performance results where he used single client server configuration and C++ language. Some performance results in the context of real-time systems are presented in [11], [12] and [13]. A common characteristic of all performance studies is that the authors used very simple tests to measure performance and that they only investigated single client scenarios. In our earlier work [14, 15, 16] we have presented some performance results for CORBA/Java (using Inprise Visibroker) and RMI version 1.1. We have based our earlier performance measurement on a modified ATM (Automatic Teller Machine) application. Although we have considered multi-client scenarios the performance assessment procedure was not as advanced as the one presented in this article.

In this article we focus on performance assessment, comparison and optimization of the Java 2 RMI and RMI-IIOP distributed object models. First we globally describe the remote method invocation mechanism, compare RMI to CORBA, JRMP (Java Remote Method Protocol) to GIOP (General Inter-ORB Protocol), identify the goals and possible solutions for RMI-IIOP, and explain the role of concurrency models supported by ORBs (Object Request Brokers) (Section 2). We show, that the selection of appropriate performance metrics for distributed object systems enables a quantitative evaluation and comparison of target models and the identification of bottlenecks. Therefore we define the performance criteria for some relevant usage patterns that include single and multi-client interactions, different basic and compound data types and data sizes. With careful selection of criteria we achieve comparability among different distributed object models (Section 3). We define the performance measurement procedure and report and compare the results for RMI and RMI-IIOP (Section 4). These results represent how the tested systems perform. With the analysis of the results and the code we identify the sources of overhead and point to the bottlenecks (Section 5). These results depict why there are differences in performance. Finally, we present general performance
optimizations and specific optimizations for RMI-IIOP. We show the improved performance for RMI-IIOP post-beta versions and the final release version (Section 6). This article forms a basis for building an analytical performance model for distributed object models. In this paper however the analytical modeling is only slightly mentioned because of the intended audience that is interested into information of practical use to the reader rather than theoretical discussions [4].

2. Method Invocation in Distributed Object Models

Distributed object models allow clients to invoke methods on distributed objects without concern for the following [2]:

- **Object location transparency:** the details about location of client and server objects are managed without affecting the implementation.

- **Platform and programming language transparency:** the client does not have to know on which platform runs the server nor which programming language it is implemented in.

- **Communications protocols, interconnections and hardware transparency:** distributed object models provide an abstraction layer. Therefore the application does not have to cope with different communication protocols, data type sizes, ranges, storage layouts and other hardware differences.

On Figure 1, client is a distributed object that performs application tasks by obtaining object references to remote server objects and invoking methods on them. Server (object implementation) is an instance of a class, associated with a public remote interface, that implements the services provided by the object. The client/server roles can be (and often are) exchanged during the run-time. Client and server objects communicate with messages. For sending and receiving messages the objects do not have to take up with communication details. Rather they use an abstraction layer – the object request broker (ORB). ORB is the integral part of a distributed object model [1, 2, 17]. It is responsible for handling the details of the communication (Figure 1):

- marshalling the client’s request,
- locating the appropriate target object,
transferring the request,
receiving the message request,
demultiplexing, demarshalling and dispatching the request,
performing an operation upcall,
returning the result.

Figure 1  Remote Method Invocation Path

To provide the communication, the ORB core uses the operating system services. For message interchange between distributed objects a wire protocol is used. This can be a proprietary protocol such as Java Remote Method Invocation Protocol (JRMP) or the General Inter-ORB Protocol (GIOP) which is standardized by Object Management Group (OMG). Distributed method invocation is very complex and introduces many layers of overhead.

2.1. Java Remote Method Invocation (RMI) and Java Remote Method Protocol (JRMP)

Java RMI utilizes strict separation of interfaces and implementation, although the interfaces are specified and the functionality is implemented in Java. The three independent layers that constitute the RMI system are [1]:
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. Stubs and skeletons are generated by the rmic compiler. In version 1.2 skeletons are no longer required for remote method call dispatch. Instead generic code is used for these duties\(^1\).

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 set-up and management of the connection and dispatching the requests to the remote objects within the transport layer’s address space.

RMI was developed for Java only and is fully integrated with the JVM. When compared with the CORBA, RMI offers some unique features like:

- passing objects by value,
- distributed garbage collection algorithm that is similar to Modula-3’s Network Objects [18],
- URL based object naming,
- dynamic class downloading,
- dynamic stub downloading.

On the other hand RMI lacks some functionality offered by CORBA compliant ORBs. RMI supports only two-way static synchronous method invocation. There is no support for interface or implementation repository, for dynamic method invocation, one-way operations, or dynamic server implementations. RMI also lacks a language independent wire protocol. Therefore interoperability with other languages is difficult\(^2\).

For the wire communication RMI uses the Java Remote Method Protocol (JRMP). JRMP makes use of two other protocols [1]: Java Object Serialization and HTTP (Hypertext Transfer Protocol). Java Object Serialization is used for call marshaling and returning results. HTTP is used to send remote method

\(^1\) For performance evaluation the 1.2 compliant stubs have been used.

\(^2\) However the interoperability can be achieved using Java Native Interface for example.
invocation data and obtain results. Usually the RMI transport layer opens direct sockets to hosts. Such
collection has been used in this performance evaluation. To bypass firewalls two alternative HTTP based
mechanisms are available. Both send RMI data encapsulated into a HTTP POST request. HTTP connections
are at least an order of magnitude slower than those sent through direct sockets [1]. RMI transport protocol
defines six messages: \texttt{Call}, \texttt{Ping}, \texttt{DgcAck}, \texttt{ReturnData}, \texttt{HttpReturn} and \texttt{PingAck}.

2.2. \textit{Common Object Request Broker Architecture (CORBA), General and Internet Inter-ORB Protocol (GIOP/IIOP)}

CORBA (Common Object Request Broker Architecture) is based on the Object Management Architecture
and the Core Object Model [17]. The CORBA specification covers the following main parts: the object
request broker (ORB), object services (CORBA services), common facilities (CORBA facilities), domain
facilities and application objects. In this article we will focus on the CORBA object request broker only.
CORBA is independent of a programming language. Similar to RMI, CORBA utilizes strict separation of the
interface and the implementation. Interfaces are specified in a proprietary IDL (Interface Definition
Language) while implementation is done in any programming language for which a mapping from IDL
exists\textsuperscript{3}.

The architecture of a typical CORBA compliant object request broker includes:

- Static stubs and skeletons: stubs provide the static invocation interface and are responsible for
  marshalling the requests. Skeletons demarshall the low-level message representation into typed data.
  Static stubs and skeletons are generated at compile time by an IDL compiler.

- Dynamic invocation interface: allows the dynamic request generation which is useful when the client has
  no compile time knowledge of the interfaces it is accessing.

\textsuperscript{3} Currently C, C++, Smalltalk, Java, Ada, COBOL mappings are standardized by OMG. For other languages non
standardized mappings exist.
• Dynamic skeleton interface: makes it possible to implement object implementations that have no static knowledge about the interface they are implementing. The client does not know whether the server is using the static skeletons or the dynamic skeleton interface.

• Object adapter: associates an object implementation with an ORB, demultiplexes and dispatches the requests. Examples are Basic Object Adapter and recently defined Portable Object Adapter.

• ORB Core: is responsible for the communication between the client and the server object. For low-level communication operating system services are used.

For inter ORB communication CORBA standard specifies the General Inter-ORB Protocol (GIOP), which is a language independent wire protocol. It defines the common data representation (CDR), GIOP message formats and GIOP transport assumptions. For all IDL types there is a CDR mapping defined. GIOP defines seven message formats: Request, Reply, CancelRequest, LocateRequest, CloseConnection, MessageError and Fragment (added in version 1.1).

With these seven messages all the functionality of CORBA is supported. More about GIOP/IIOP can be found in [2]. Internet Inter-ORB Protocol (IIOP) is a specialization of the GIOP to the TCP/IP transport protocol. Additionally to GIOP it specifies how agents open TCP/IP connections and use them for GIOP message transfer [2].

2.3. Remote Method Invocation over Internet Inter-ORB Protocol (RMI-IIOP)

We have already mentioned that today Java developers have two choices for distributed object functionality. They can use RMI, the native distributed model bundled with Java from version 1.1. They can also use the CORBA model by using one of the several implementations. In Java SDK 2 a CORBA object request broker implementation is included under the name Java IDL.

A common goal was to merge the two distributed object models, RMI and CORBA. RMI clients will be able to access CORBA servers and CORBA clients will be able to access RMI servers transparently. This will
spare the developers to make a choice between the usability of RMI and the interoperability of CORBA or to deal with two different incompatible protocols. Therefore the RMI application programming interface (API) should remain unchanged as much as possible and comparable performance should be delivered. There are two possibilities to achieve the stated goals:

- Build a bridge between both protocols or
- Provide a consistent mapping from Java to IDL (and therefore to IIOP).

The second approach has numerous advantages and was therefore chosen. Whilst the mapping of basic data types is trivial there are many ambiguities with complex data types. The OMG defined mapping from IDL to Java is not bijective. This means that if you map the IDL to Java and then from Java back to IDL you do not get the same result. For example, IDL data types `enum`, `struct`, `union` and `typedef` map to a Java class. The mapping from Java to IDL is not trivial and can have significant impact on performance, as we will show in the next section. There is also a problem with handling IDL `out` and `inout` parameters and providing consistent handling of IDL data type `any`.

The subset of Java that is fully compatible with current RMI semantics and with IDL, IIOP and the CORBA object model is referred to as RMI/IDL. It is defined in [19] and will be included in the CORBA 2.3 as a chapter “Java Language to IDL Mapping”. The conforming RMI/IDL types are all standard Java primitive types (void, boolean, byte, char, short, int, long, float, double), remote interfaces that follow certain restrictions, RMI/IDL value types, arrays of conforming types, exceptions, CORBA object references and IDL entity types.

Value types represent classes whose values are moved between systems using the copy semantics. The actual state of the object is transmitted between systems rather than transmitting the object reference. We have already mentioned that CORBA 2.2 provides support for transmitting object references only. RMI on the other hand supports both types of transmission: by reference and by value. Therefore CORBA 2.3 specification will be enhanced with the Objects By Value standard [20]. This standard extends CORBA and

---

4 By the time of writing this paper CORBA 2.3 has not been approved by OMG yet.
OMA (Object Management Architecture) with the notion of Value which provides semantics that bridge CORBA structs and CORBA interfaces and allows transferring object by value. To achieve this, several extensions to the IDL have been introduced like: `value, ValueBase, custom, safe, abstract, public, init,` and `supports`. Also the GIOP (General Inter-ORB Protocol), IIOP and the on-the-wire format are extended. Support for transmission of the object state and the repository type information has been added. Also several other extensions have been introduced like extensions to the Interface Repository and new Type Codes.

RMI-IIOP has been jointly developed by IBM and Sun and will be included in the next major release of Java. It is based on the recently adopted specifications and provides transparent interoperability between RMI and CORBA distributed object model. RMI-IIOP can work with any CORBA implementation that supports these new specifications.

2.4. Concurrency Models

Distributed object applications are commonly used in multi-client environments. This means that several clients simultaneously invoke methods on a single shared object implementation (server). From the performance viewpoint it is crucial how the ORB handles incoming requests. In general it can serialize them or it can process them in parallel. This is known as the ORB concurrency model. In different ORBs several models are implemented like blocking server, thread-per-server, thread-per-connection, thread-per-request or different thread pool architectures like worker thread pool or leader/follower thread pool architecture. All but the blocking server and the thread-per-server process incoming requests in parallel. Usually the leader/follower thread pool or thread-per-request architectures are deployed as they are relatively easy to implement [21] and offer acceptable performances [22]. The thread pool architecture concurrency is limited by the number of threads in pool. The thread-per-request ultimately limits the number of clients a server can support to the operating system upper thread limit.

When evaluating performance the concurrency model can have a significant influence on the results. Therefore we have examined the architectures provided by Java RMI and RMI-IIOP. They all handle incoming requests in parallel. We have verified this with a test described in [23]. According to the
documentation the RMI uses the thread-per-request and the RMI-IIOP the thread pool architecture. Concurrency models used by RMI and RMI-IIOP are described in detail in section 5.

3. Performance Evaluation

3.1. Goals

Our goal was to evaluate the performance of Java RMI and RMI-IIOP from a developer’s standpoint. We wanted to gain results that will make it possible to understand how do the models perform and why there are differences in performance. An important goal was to make the results comparable between the two (and possibly other) models. We wanted to cover all the relevant aspects such as different basic and user defined data types, different data sizes and multi-client scenarios which are especially important in global networked applications. The goal of performance evaluation has not been going into low level details of the communication. Therefore we have identified performance criteria that: (1) help in the selection and (2) ensure the suitability of ORB middleware for the target application domain, (3) enable the identification of strong and weak point of each model and (4) provide a basis for analytical modeling.

The performance evaluation method used in this paper is independent of the underlying distributed object model and minimizes the influence and the overhead of the performance control component. We have provided results that are directly comparable between models. We have also provided implementations for both target environments.

3.2. Performance Criteria

For a developer the most important performance criterion is the total time taken by a remote method invocation. This time is defined as the round trip time (RTT). RTT is the time that elapses between the initiation of a method invocation by the client until the results are returned to the client. In our case it is
measured in milliseconds (ms). Because the methods used for performance evaluation did not do any processing the RTT expresses the overhead of remote method invocation. It is important to understand:

- How different data types as parameters and return values influence the result:
  
  RTT for eight simple data types, string, user defined data type and an object reference have been measured.

- How the data size influences the results:
  
  Performance results for different data arrays from size 1 up to size 16384 have been gathered.

- How multi-client interactions influence the results:
  
  RTT for one and up to eight simultaneous clients has been measured.

Round Trip Time is certainly one obvious and interesting metric. In these test cases, the network is fast and has a high bandwidth. The Client and Server Systems Under Test are running tightly controlled repeated loops of remote method requests. The response time in this distributed configuration is approximately Client CPU time + network time (outbound) + Server CPU time + network time (inbound) - assuming single message in-out per method cost once in a steady-state. Since the same processing takes place in the Client and Server processes, whether they are on the same physical machine or two distributed machines – any single machine transport optimizations are at a very low level, and remain constant for a given transport type – it can be the case that a test will achieve a faster RTT running distributed than when on a single machine. CPU consumption of client and server machines is important in determining the machine capacity. When analyzing results from a single computer, uniprocessor configuration, clearly both the server and client pathlength are being measured together. There will be some low-level transport saving opportunities as these are loop-back rather than network measurements, though note that these will be constants in the results for any given protocol.

*Throughput* is defined as the number of method invocations in a given time interval (usually a second). Throughput and round trip time are inversely proportioned.

*Performance degradation* when multiple clients simultaneously interact with a single server. It is defined as a function of number of simultaneous clients. Performance degradation as a base index is computed as
quotient between the RTT by the given number of simultaneous clients and the RTT by a single client. Performance degradation as a chain index is computed as quotient between the RTT by the given number of simultaneous clients and the RTT by a given number of clients minus one. RTTs for up to eight simultaneous clients have been measured. The application driver framework manages physically separate, identical client computers, synchronizing measurement runs. Each client generates a new object request as soon as it receives the previous response, without delay. The test is intended to show the effects upon response time by increasing workload, as opposed to realistically simulating a specific number of end-users. In these tests, a single client machine does not typically saturate the server CPU capacity. This, together with batching efficiencies, mean that the total amount of completed object requests per unit time (server capacity), increases somewhat as the number of client machines is increased. See [23] for a further discussion. Evaluating multi-client scenarios is especially important because they correspond to the typical usage pattern found in today’s Internet applications.

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

The main observations from this workload have been measured Round Trip Time responses for remote objects, either on physically remote computers or logically remote JVMs. This application would be very suitable for the use of queueing analysis techniques. In Kendall notation these systems are denoted as M/M/1/8 - it is a variation on the 'machine repairman' model, though since there is no client-delay ('the completely ineffective machine repairman'), we only observe the queueing time effects at close to 100% server utilization. The main difference between this model and open systems such as M/M/1 is that here queueing time increases as a linear function rather than an exponential one when the server is fully utilized [24]. An analytical model based on the experiment results presented in this paper provides a solid basis for further theoretical work in this area. As already mentioned, the model has not been included into this paper because of the practical nature of the Systems Journal.
3.3. Implementation Details

To measure RTT, interfaces (Listing 1) with corresponding classes have been defined on the server side for each basic data type listed in Table 1 and for two compound data types: testStruct and myObject. Each interface provides two methods: one that accepts a data type as a parameter and has no return value, and one that has no parameters but returns the data type as a return value. In both methods the processing overhead has been omitted.

```java
// TEMPLATE FOR INTERFACES FOR BASIC DATA TYPES, FOR TESTSTRUCT AND MYOBJECT
public interface <data_type>TestServer extends Remote {
    void acceptType(<data_type> Value) throws RemoteException;
    <data_type> char returnType() throws RemoteException;
}

// COMPOUND TYPE TESTSTRUCT
public final class testStruct implements java.io.Serializable {
    public boolean b; // instance variables
    public byte o;
    public short s;
    public int l;
    public long ll;
    public float f;
    public double d;

    public testStruct() { } // constructors
    public testStruct(boolean __b, byte __o, short __s, int __l, long __ll, float __f, double __d) { ... }

    private void writeObject(java.io.ObjectOutputStream out) throws java.io.IOException {...};
    private void readObject(java.io.ObjectInputStream in) throws java.io.IOException, ClassNotFoundException {...};
}

// OBJECT REFERENCE MYOBJECT
public interface myObject extends Remote {
    byte[] a() throws RemoteException;
    void a(byte[] arg) throws RemoteException;
}
```

Listing 1: Java RMI and RMI-IIOP server side interfaces

In addition to simple data types two user defined data types are introduced. First, a data-only class named testStruct is presented in Listing 1. As it is transferred by value it has to implement the serialization interface. By RMI-IIOP the testStruct is handled as a RMI/IDL Value Type [19] and the transfer conforms to the Objects By Value specification [20]. As the sending and the receiving context had local implementations of the class that were identical no code downloading has been used.

Second, a user defined class is introduced, named myObject. The definition is shown in Listing 1. Both RMI and RMI-IIOP transfer myObjects by reference. For RMI-IIOP myObject is a RMI/IDL Remote Interface and has to conform to the CORBA IOR (Interoperable Object Reference) specification. The difference
between `myObject` and `testStruct` can be easily observed from the listing as `myObject` extends the `Remote` interface and the `testStruct` does not.

To be able to measure RTT for different sizes of method parameters and return values new interfaces are introduced. In contrast to the interface presented in Listing 1 the new interfaces deal with arrays of all the described data types. In RMI-IIOP the RMI/IDL arrays are mapped to value types that contain IDL sequences.

Table 1 presents the equivalent mapping between Java RMI and RMI/IDL (CORBA IDL used by RMI-IIOP) basic data types. The mapping of Java RMI boolean, short, int, long, float and double is straightforward because they have exact IDL analogous types. The Java RMI signed data type byte is mapped to the unsigned IDL type octet. Both are 8 bits long. The Java RMI Unicode char is mapped to the IDL wchar type.

<table>
<thead>
<tr>
<th>Java RMI data type</th>
<th>RMI-IIOP/IDL data type</th>
<th>Length in bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>boolean</td>
<td>8</td>
</tr>
<tr>
<td>char</td>
<td>wchar</td>
<td>16</td>
</tr>
<tr>
<td>byte</td>
<td>octet</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>long</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>long long</td>
<td>64</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>64</td>
</tr>
</tbody>
</table>

*Table 1: Simple data types used for performance evaluation*

For implementing the multi-client scenarios special synchronization objects have been added to the system. These object guarantee that the multi-client invocations are done synchronously. Synchronization is achieved using semaphores. The synchronization is done in the intervals between performance measurements. For the actual implementation the concurrency control services provided by each architecture have been used. The synchronization objects are placed on a separate processing unit. Therefore synchronization has negligible impact on the performance results. The details of synchronization as well as the performance evaluation method are described in detail in [23].

---

5 When transferred over the wire.
The actual measurements are done on the client side. A client implements the following activities:

- binds to the server side object implementations used for synchronization,
- binds to the server side object implementations used for performance measurements,
- opens an output table where the results are written to,
- allocates memory for storing the temporary results,
- performs the performance measurements,
- calculates the results,
- writes the results to the table.

The performance measurements are performed for each interface described. Let $X$ denote the performance measure of interest. To obtain the $i$-th observation $X_i, i=1, 2, ..., n$, the following steps are necessary:

- the client waits to get the synchronization acknowledgement,
- it reads the system time,
- it performs the method invocations. To get the desired resolution of the result the test is performed $r$ times,
- it reads the system time again and calculates the $X_i$-th performance observation.

After completing the $n$ observations, sample mean $\bar{X}$, the variance $s^2$ and standard deviation $s$ are calculated.

### 3.4. Software and Hardware Testbed Equipment

For all the performance measurements the Sun Java® 2 SDK, Standard Edition, version 1.2 has been used. Performances for RMI-IIOP are reported for beta version 0.2 (dated Dec 18, 1998), for post-beta builds 7b, 8c and fcsb2 and for the final release version. All source code has been compiled and executed using the Java 2 SDK, Standard Edition, version 1.2. Symantec Just-in-Time (JIT) compiler level 3.00.078(x) has been enabled. For the code analysis the Intuitive Systems Inc. OptimizeIt® 3.02 Professional profiler has been used. All the computers used Microsoft Windows® NT 4.0 Workstation with Service Pack 3 as their operating system.
The actual performance measurements have been done for up to eight simultaneous clients. Therefore ten identical Pentium II-333 MHz computers with 128 MB RAM have been used. Eight of them were used for client applications, one of them was the server and one was used for running the synchronization objects. The computers were connected into a 100 Mbps Ethernet network that was free of other traffic.

All the performance measurements have been repeated 20 times and the average values are reported. Additionally each method has been invoked 200 times to achieve the necessary timing accuracy.

4. Performance Results for RMI and RMI-IIOP

For the presentation of performance results we have proceeded from the following presumptions:

- In the majority of applications it is most likely that the distributed object communication will take place between client and server objects executing on distinct computing entities.
- The majority of applications that will take advantage of distributed object interoperability can be separated into two parts: applications that will use one client object to one server object communication and applications that will use many client objects to one server object communication.
- In the many-to-one scenarios it is unlikely that the client objects would send requests to the server object continuously without delays. Therefore the performance levels presented in this paper are achieved by a much larger number of real-world clients.

In this section we present the observed external performance results for RMI and RMI-IIOP beta, first for the single-client scenarios and then for the multi-client scenarios. In each section we give a short explanation of the results. Where possible, same scales for graphical presentation have been used. For the primitive data types byte, int and long the notation RMI data type/IDL data type is used: byte/octet, int/long and long/long, respectively.
4.1. Single Client Scenarios

First we compare the performances of primitive data types. Then we compare primitive data types with arrays containing one element. We compare RTTs and throughput for different data sizes for basic and compound data types. Finally we investigate the scenario with the client and the server objects executing on a single computer versus the scenario where client and server objects are on two distributed computers.

4.1.1. Analysis of Primitive Data Types

The graph on Figure 2 presents the average Round Trip Times for methods accepting and returning primitive data types (scalars). The results are reported for scenario where client and server object are executed on separate computers.

![Comparison of RTT for Primitive Data Types](image)

Distributed method invocation is significantly slower than a normal method call within a single JVM. A normal Java method invocation takes around 300 nanoseconds as measured by our tests. A comparable RMI invocation on the same computer (between two JVMs) takes around 1.2 milliseconds (ms) and a RMI-IIOP invocation almost 2 ms. This is mainly because of the complexity of distributed data flows (marshalling, demarshalling, demultiplexing) and the necessary low-level transport and context switching activities.
In Figure 2 we can see that RMI achieved the fastest times. For basic data types (boolean, char, byte/octet, short, int/long, long/long long, float, double) RMI-IIOP is 66% slower. As already mentioned the class testStruct is handled differently by RMI and RMI-IIOP. RMI-IIOP transfers it as CORBA RMI/IDL value type conforming to the Objects by Value specification. We can see, however, that RMI-IIOP is almost 150% slower than RMI. For the object reference (myObject) the results are closer. RMI-IIOP is 20% slower. If we take a look at the string we can see that RMI-IIOP is around 70% slower than RMI.

We can observe similar behavior for all basic data types although their byte size when transferred over-the-wire differs. It is clear that the remote method invocation overhead is larger than the time needed to transmit a single instance of a data type. It is also evident that RMI-IIOP introduces more layers (as it is based on the CORBA implementation Java IDL), is not so well integrated into Java, and considering it is a beta version offers room for optimization (described in sections 5 and 6). The much slower result for the testStruct in RMI-IIOP is a consequence of the transmission technique used. This is also a point where major improvements will be introduced in the final release version. The behavior of the object reference does not differ very much although RMI-IIOP has to conform to the CORBA IOR (Interoperable Object Reference).

4.1.2. Primitive Data Types vs. Arrays with One Element

Figure 3 shows the comparison between primitive data types (scalars) and arrays with one element. In previous subsection we have seen that the basic data types behave very similar. Therefore times for a geometric mean of basic data types (boolean, char, byte/octet, short, int/long, long/long long, float, double), for testStruct and myObject are compared. This comparison shows the overhead of the data presentation layer for handling the arrays. The presented data holds for single client scenario where server and client objects are located on separate computers. Data type string is omitted because the use of an array of one character strings does not make any sense.
With RMI the overhead for handling arrays is 8% for basic data types, 5% for `testStruct` and 2.5% for object reference. RMI-IIOP shows 15% overhead for basic data types, but 41% and 57% for `testStruct` and `myObject`, respectively. RMI-IIOP’s overhead with the later two is too large, therefore optimization will be done on this area for the final release.

The overhead is explained if we look at the multiplexing process. In RMI the arrays are serialized so that first the array length is written and then each element is serialized. In RMI-IIOP the arrays are represented as RMI/IDL value types with IDL sequences. When serialized they must be marshaled with a repository ID indicating their runtime type.

### 4.1.3. Impact of Data Size on RTT

Figure 4 and Figure 5 show the RTT for different data sizes for RMI and RMI-IIOP, respectively. The results are reported for arrays sized from 1 to 16384 elements. Results for single client scenario with client and server on separate computers are shown. Please note the logarithmic x axis in both graphs.
As expected RMI-IIOP is slower than RMI. For example the average RTT for basic data types (boolean, char, byte/octet, short, int/long, long/long long, float, double) by arrays with 4096 elements in 2 times slower by RMI-IIOP. More interesting is the behavior for 4096 characters string where RMI-IIOP is only 38% slower than RMI.

To understand the comparison we should be aware how RMI and RMI-IIOP transfer the data. RMI-IIOP handles java.lang.String as RMI/IDL value type WstringValue, which includes a IDL wstring attribute. In
RMI the arrays are serialized according to the serialization specification [25]. By RMI-IIOP arrays are handled as RMI/IDL boxed value types containing IDL sequences. When marshaled a repository ID has to indicate their runtime type. Because they are value types they are transferred according to the Objects By Value specification. Handling RMI-IIOP arrays involves more overhead and is more time consuming than RMI arrays.

According to the Table 1 we would foresee that the types with the same byte size (when transferred over-the-wire) would behave approximately the same. This is the case with basic data types, but we can see that the float and double are always a little slower than the int/long and long/long long, respectively. The answer once again is hidden in the (de)multiplexing process where floating point numbers are first converted to integers (using methods floatToIntBits and doubleToIntBits for multiplexing and intBitsToFloat and intBitsToDouble for demultiplexing). Then they are handled in the same way as int/long and long/long long.

All the data types show almost linear dependence between data size and RTT. Data types myObject and testStruct show a much larger RTT increase than the basic data types. In RMI the testStruct is an object that has to be passed by value. Serialization methods are provided (see section 3). In RMI-IIOP however the testStruct is transmitted according to the CORBA object-by-value specification. This is the main reason why RMI-IIOP is much slower. The transformation is time consuming. For object reference RMI-IIOP has to use the CORBA compliant IOR (Interoperable Object Reference). By the performance of RMI-IIOP arrays for testStruct and myObject major improvements have been achieved since the beta version was released. We will present the improvements in the section 6.

To get a better understanding for basic data types it is interesting to observe the throughput. The highest throughput by far is achieved with RMI; its peak value is around 37.4 Mbps and is more than a third of the theoretical network throughput. RMI-IIOP achieved throughput with peak values around 18.5 Mbps. Both RMI and RMI-IIOP achieve their peak values at arrays of characters, which are 16 bits long and have the lowest overhead of multiplexing and demultiplexing. Arrays of bytes do not achieve this throughput because
they are only 8 bits long. Java.lang.Strings on the other hand are objects. Transferring objects between different JVMs is more costly than arrays of characters.

Figure 6 presents the geometric average throughput for basic data types (boolean, char, byte/octet, short, int/long, long/long long, float, double, and string). RMI-IIOP’s average throughput is ~2.3 times lower than RMI’s.

Figure 7 shows the throughput comparison for the structure `testStruct` and Figure 8 for the object reference `myObject` (please note the secondary axis for `myObject`). For the `testStruct` it is especially noticeable that the throughput of RMI-IIOP is an order of magnitude lower than RMI. The reason is the following: RMI and RMI-IIOP treat the `testStruct` as an object which is transferred by value. RMI-IIOP has to conform to the OMG Objects By Value [20] specification. This means that major transformations are needed. Improving the RMI-IIOP performance in this point is an important task before bringing out the final version. Object reference is handled by both architectures similar. You should be aware however that whilst RMI has to deal with native Java object references, RMI-IIOP has to use the CORBA IORs instead.

### 4.1.4. Analysis of Single Computer Versus Two Distributed Computers Results

The graphs on Figure 6, Figure 7, and Figure 8 show the throughput comparison for single computer scenario versus two distributed computers. In single computer scenario the client and the server object are executed on one computer in two JVMs (separate processes). Figure 6 presents the average throughput for basic data types (boolean, char, byte/octet, short, int/long, long/long long, float, double, and string), Figure 7 for `testStruct` and Figure 8 for the object reference (`myObject`).
Figure 6  Average Throughput Comparison for Basic Data Types: Two Distributed Computers and Single Computer Scenarios

Figure 7  Throughput Comparison for testStruct: Two Distributed Computers and Single Computer Scenarios
For primitive data types (scalars) the difference in average throughput for the single computer and two distributed computers is minimal. Throughput for RMI is around 3% slower on two computers while RMI-IIOP is faster on two computers – from 3% to 5% on average. When comparing arrays of different sizes we can observe an obvious trend that shows better throughput when the client and the server are placed on two distributed computers. To explain this behavior we have measured the CPU usage in both scenarios. In the single computer scenario (two JVMs on a single computer) we have observed a 100% CPU utilization when the method invocation took place. Between distinct method invocation a short time CPU utilization drop has been observed. The two JVMs occupied all the available CPU resources.

(Note to the editors: Only the CPU Usage history is relevant in the context of this article. The whole screen shot is included to show that such behavior has actually been measured.)
Figure 9  CPU Usage for the Server Side when Client and Server are Placed on Separate Computers
In Figure 9 and Figure 10 we can observe the CPU usage when the client and the server objects are moved to the separate computers. Effectively we have 2 x 333 MHz processing power, connected via a high speed network, compared to 1 x 333 MHz on a single computer. Both single computer and distributed implementations are using the full complexity of the distributed message paths – no optimizations such as fast local inter-process communication for a single machine in the products being compared, although the network subsystem will run local loop-backs. Neither client nor server computer reached peak CPU usage of 100%. The JVM processing utilization seems to be the answer why the scenarios where client and server are distributed are faster.

4.2. Multi Client Scenarios

4.2.1. Primitive Data Types

In Figure 11 and Figure 12 we can see the RTT for primitive data types in multi client scenarios for up to eight simultaneous clients. The RTTs shown in graphs present the average times per client. Both models execute each client’s request in a separate thread (see section 2.6). Therefore we can observe very similar behavior of basic data types. TestStruct and myObject show deviations where by RMI the object reference myObject shows the largest times, and by RMI-IIOP the testStruct.
By eight simultaneous clients RMI-IIOP is around 100% slower than RMI. For the testStruct RMI-IIOP is 3 times slower and for the object reference around 10% slower than RMI. Although the RMI-IIOP beta version achieves comparable results, optimizations will be applied especially to the testStruct handling performance.

Figure 12  RMI-IIOP: RTT per Client for Primitive Data Types in Multi-Client Scenario

Figure 13  Comparison of the Geometric Average Performance Degradation
In Figure 13 the geometric average performance degradation for basic data types, `testStruct` and `myObject` is compared between RMI and RMI-IIOP. Although the degradation by RMI is lower than by RMI-IIOP both are comparable. This leads us to the conclusion that when the RMI-IIOP single client performance gets improved will the improvement influence the actual multi-client performance as well.

For basic data types the cumulative degradation for the eight client scenario is lowest for RMI (5.9), followed by RMI-IIOP (7.4). Note however that RMI had the lowest degradation for up to six concurrent clients. The results for the `testStruct` are similar, except that both models showed comparable results up to seven clients. It seems that by the eight concurrent clients we have reached the point of RMI-IIOP model where the performance drops rapidly. The degradation for the object reference (`myObject`) for seven simultaneous clients is lowest by RMI (4.7), followed by RMI-IIOP (5.7). The eight concurrent client however contributed more than 40% degradation for RMI. Again, it seems that we have reached the point where the performance degradation increases rapidly for RMI. Therefore by the eight client scenario the RMI-IIOP leads, followed by RMI.

When observing primitive basic data types under heavy client load we have seen that the response times (and performance degradation) are not subordinated by the data type but only by the number of simultaneous clients. By compound data types there are minor differences as an effect of different handling of those data types, described in previous subsection.

### 4.2.2. Impact of Data Size on RTT and Performance Degradation

In this section we will investigate the impact of data size on the response times and on the performance degradation. Although we have measured the results for all basic and compound data types we have decided to present only geometric mean RTT for basic data types (boolean, char, byte/octet, short, int/long, long/long long, float, double, string) as we think that they sufficiently describe the common behavior observed in all cases.
Figure 14  RMI: Average RTTs for Arrays of Basic Data Types in Multi-Client Scenario

Figure 15  RMI-IIOP: Average RTTs for Arrays of Basic Data Types in Multi-Client Scenario

Figure 14 and Figure 15 show the geometric mean response (RTT) times for basic data types for RMI and RMI-IIOP, respectively. Note: the scales are identical for easier comparison. The absolute performance is again the best by RMI. For large data sizes and many concurrent clients RMI-IIOP is around 58% slower.

We have already observed the rapid performance degradation for arrays of compound data types (testStruct and myObject) in the single client scenario. The multi-client results confirm this. We have already identified the RMI-IIOP ineffectiveness by handling arrays of testStructs, mainly because of using the CORBA object-by-value specification.
The analysis of data for multi-client and large data sizes shows that the correlation between the number of clients and the relative performance degradation is strong (which is obvious). It also shows, that the dependence between the data size and the degradation is minimal. In other words, the performance degradation for different sized arrays is similar and is mainly subordinated to the number of concurrent clients.

5. Overhead Identification and Optimizations

In section 4 we presented the external performance results for RMI and RMI-IIOP. In this section we explain how the three evaluated models work and why there are differences in performance. We will point to the source of the performance overheads. Finally we will present optimizations and show how to improve performance.

5.1. Overhead Analysis

For analyzing the overhead we have used a modified version of the performance evaluation application described in section 3. To get the insight into multiplexing/demultiplexing, dispatching and demarshalling we have decided to use the class octetSeqTestServer. Data type byte/octet requires the least amount of conversion, therefore it is the data type best suited for studying and comparing the behavior. We have analyzed the method acceptType, which has been invoked 200 times. This method had a parameter – an array of bytes. Therefore the client object had to marshal the parameter, which was transferred to the server object, where it was demarshaled, demultiplexed and dispatched. The server object returned only an acknowledgement to the client side. Therefore, to study marshalling we will focus on the client side. To study demarshaling we will look at the server object.
5.1.1. RMI

The analysis of server side overhead shows that RMI uses the thread-per-request concurrency model. Table 2 shows the server and the client side overhead for RMI for different parameter sizes – arrays of byte. The numbers shown in table reveal the percentage of the total execution time spent in the corresponding method. Only the most important methods are listed.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>% of time spent in the method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data size in elements:</td>
<td>1</td>
</tr>
<tr>
<td><strong>SERVER SIDE</strong></td>
<td>&lt; 1.00</td>
</tr>
<tr>
<td>java.net.SocketOutputStream.socketWrite</td>
<td>3.36</td>
</tr>
<tr>
<td>java.io.ObjectOutputStream.&lt;init&gt;</td>
<td>5.36</td>
</tr>
<tr>
<td>java.io.IOExceptionReader.read</td>
<td>2.21</td>
</tr>
<tr>
<td>java.io.ObjectInputStream.allocateNewArray</td>
<td>&lt; 1.00</td>
</tr>
<tr>
<td>java.net.SocketOutputStream.socketWrite</td>
<td>19.95</td>
</tr>
<tr>
<td>java.net.InetAddressImpl.lookupAllHostAddrs</td>
<td>5.04</td>
</tr>
<tr>
<td>sun.rmi.server.UnicastRef.invoke</td>
<td>4.12</td>
</tr>
<tr>
<td>java.lang.System.currentTimeMillis</td>
<td>2.52</td>
</tr>
<tr>
<td>java.io.BuffferedOutputStream.write</td>
<td>&lt; 1.00</td>
</tr>
</tbody>
</table>

Table 2: Analysis of the server side and the client side overhead for RMI

For single element array the server spends ~19% of its time in the `socketWrite` method of the `SocketOutputStream` class. This method writes an array of bytes to the socket and is natively implemented for the target Java environment (for example Solaris or Win32). Similar holds for the `socketRead` method of the `SocketInputStream` which becomes more and more important with increasing data size. With the increasing data size the method `allocateNewArray` consumes up to 24.60% of the total execution time. This method allocates a new array for the specified class and is implemented natively since the type of the array needs to be set to the corresponding class. Method read of the `BufferedInputStream` class reads bytes from a byte-input stream into the specified byte array, starting at the given offset, and consumes up to ~5% of the total execution time.

For the client side we can observe that the majority of time is spent (~20% to ~23% depending of the data size) in the `socketWrite` method, which is a natively implemented method for writing to a socket. Method
lookupAllHostAddr finds an internet address for a given hostname and is implemented natively too. Method UnicastRef.invoke returns the result of a method invocation on the remote object which resides on the remote host. Therefore it has to create the call context, marshal the parameters and demarshal the result. To create an unique identifier, the class UID uses the method currentTimeMillis\(^6\), which returns the current time in milliseconds, and pairs it with the Internet address.

5.1.2. RMI-IIOP Beta

RMI-IIOP beta uses the CORBA (Java IDL) infrastructure for implementing the distributed method invocations\(^7\). RMI-IIOP uses the thread pool with the Listener, Reader, and Worker threads. The Listener Thread handles the low level connection details. For each client a Reader Thread is created which handles the connection with a specific client. For each client also a pool of 202 Worker Threads is created. These threads process the IIOP protocol input streams.

Table 3 shows the server side overhead in the Reader Thread and the client side overhead in the Reader and the Main Thread. On the server side the most important thread is the Reader Thread. The majority of time is spent in the socket communication (socketRead) and in the thread creation (Thread.start). With the increasing data size the initialization of the IIOP input stream becomes time consuming.

On the server side there are also the Listener and the Worker Thread. The behavior of the Listener Thread concentrates mainly in the java.net.PlainSocketImpl.socketAccept method, which is a native method of the default socket implementation. The Worker Thread shows a few time consuming methods. The method CDRInputStream.read_octet reads the subsequent octet into the buffer. Before this happens it invokes the CDRInputStream.alignAndCheck method. This method takes care for the proper data alignment and calls the method grow when the buffer becomes too small. The method grow enlarges the buffer by a factor of two.

\(^6\) For the overhead of the currentTimeMillis method is fully responsible the UID class. Please note that the tests in this section have been modified and do not measure the RTT.

\(^7\) Although RMI-IIOP builds on CORBA it does not use the same classes as Java IDL does.
The method `CDROutputStream.write_octet` writes the subsequent octet to the buffer. It uses the `CDROutputStream.alignAndReserve` method which supervises the buffer size and calls the `grow` method to enlarge it. A consequence is data copying. The method `TypeCodeImpl.copy` copies the input stream to the output stream and with larger data sizes has a significant share in the total execution time (~33%).

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Data size in elements:</th>
<th>1</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERVER SIDE – Reader Thread</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.sun.rmi.iop.IIOPInputStream.&lt;init&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.net.SocketInputStream.socketRead</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.Thread.start</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLIENT SIDE – Reader Thread</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.net.SocketInputStream.socketRead</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.sun.rmi.iop.IIOPConnection.createInputStream</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.sun.rmi.iop.IIOPConnection.processInput</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.sun.rmi.util.Lock.unlock</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.sun.rmi.iop.ReaderThread.run</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLIENT SIDE – Main Thread</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.sun.rmi.iop.CDROutputStream.grow</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.net.SocketOutputStream.socketWrite</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.ClassLoader.defineClass0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.lang.ClassLoader.findBootstrapClass</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sun.misc.Resource.getBytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.sun.rmi.iop.CDROutputStream.write_char</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.sun.rmi.iop.CDROutputStream.write_repositoryId</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.sun.rmi.iop.CDROutputStream.alignAndReserve</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Analysis of the client and the server side overhead for RMI-IIOP beta

On the client side the RMI-IIOP client creates two threads, the Reader Thread which handles the low-level communication and the Main Thread. The majority of the time in the Reader Thread is spend in the `java.net.SocketInputStream.socketRead` method. With the increasing data size the method `com.sun.rmi.iop.IIOPConnection.processInput` becomes time consuming too (~11%).

The majority of the client side execution time is spent in the Main Thread. There are two important methods.

The `java.net.SocketOutputStream.socketWrite` consumes ~13%. With the increasing array size the `CDROutputStream.grow` method’s share increases up to ~26% of the total execution time. The method `grow` is responsible for enlarging the buffer size used for marshalling the requests. It does this by enlarging the buffer by factor of two, therefore several calls to the `grow` methods were necessary by larger data sizes.
5.2. Bottleneck Identification and Optimization

In the previous section we have identified the sources of the overhead for RMI and RMI-IIOP. We have seen that the RMI-IIOP uses a similar multithreading model to Java IDL. This is not surprising because RMI-IIOP uses the CORBA communication infrastructure and makes RMI compatible with CORBA and the IIOP protocol. The RMI-IIOP multithreading model differs significantly to the RMI multithreading model. While RMI uses a single thread for each request (in this case the client, which was single-threaded), RMI-IIOP uses more than 200 threads per client. The threads should be well exploited in order to achieve good performance.

The common bottlenecks that we have identified is summarized in the following:

- Vast amount of the communication overhead resides in the low-level methods that take care for socket based communication (socketWrite, socketRead, socketAccept). These methods are not implemented in Java. Rather they are written natively for each target platform in C language and use the underlying operating system calls.

- Much of the receiver side (the server object) and some of the sender side (the client object) overhead rises from the inefficient concurrency support – more exactly the multithreading exploitation shows comparably large overhead (Thread.start, Thread.init, Thread.run). RMI-IIOP beta performs here worse than RMI which is an outcome of the threading architecture.

- The other important source of the receiver side overhead can be found in the demultiplexing, demarshalling and presentation layer. RMI-IIOP uses an inefficient algorithm for demultiplexing which performs heavy checking (alignAndCheck) and uses bad buffering algorithm which requires buffer enlargements (grow) and leads to excessive data copying (copy). RMI on the other hand shows a bottleneck by the allocation of arrays for new classes (allocateNewArray).

---

8 Actually is this a weakness of the Java IDL implementation bundled with the Java DSK 1.2. Similar behavior can be observed by IDL tests as well.
An important source of the sender side overhead is inefficient multiplexing, easily identified by RMI-IIOP beta. This is partly because of the GIOP 1.0 specification and partly due to inefficient algorithm which performs aligning and reserving (alignAndReserve), has small initial buffer size and inefficient buffer growth method (grow).

Other sources of overhead lie in the excessive data copying and local ORB method invocations, in stub and skeleton generation, and no caching architecture.

The presented results form a solid basis to provide optimizations for identified bottlenecks. The following is the overview of the optimizations that improve performance:

- Improving the low-level communication overhead requires optimizations in the native code that handles the communication. On one side this is the integration with the operating system network features, especially advanced features, such as high-speed network interfaces, and real-time threads. On the other side something can be done with buffer optimizations and their optimal management and with the transport protocol tuning (for example socket lengths can be adjusted). None of the ORBs evaluated uses an internal buffering for network writing/reading. An optimal buffering architecture would reduce the communication overhead considerably. For low-speed communication infrastructure and large data transfers data compression algorithms could speed up the transfer.

- To make the concurrency model more efficient it is necessary to implement highly optimized thread management. For reducing the thread creation time a thread pool can be used instead of the thread-per-request architecture. The context switching should be minimized. In the Leader/Follower Thread Pool architecture the context switching is successfully minimized because the request is not transferred from one thread to another. Therefore it provides better performance than the Worker Thread Architecture used by RMI-IIOP, although it is harder to implement.

- Demultiplexing overhead is minimized with the use of fast, de-layered and flexible demultiplexing algorithms [9]. Instead of the layered demultiplexing a perfect hashing and active demultiplexing can be
implemented for best performance. Presentation layer overhead optimization is achieved by generation optimized stubs and skeletons.

- The excessive marshalling and demarshalling overhead in RMI-IIOP is based on non-optimal buffer allocation and enlargements. We suggest two solutions to this problem. First, the initial buffer size can be adjusted and the enlargement procedure can be optimized, which would improve the testcase performance. Second, the ORB should be upgraded for support for the GIOP version 1.1. GIOP 1.1 has introduced a new message type (Fragment) that allows a message to be sent in portions. Therefore the buffer enlargements become superfluous.

- RMI omits the unnecessary data conversions, found in the RMI-IIOP communication. Optimizations in GIOP protocol, that would prevent data conversions when the sender and receiver use the same format (and programming language) would speed the communication especially between the client and the server implemented in the same programming language. The optimizations could be implemented following the example of big-endian and little-endian conversion, already optimized by GIOP.

- Both ORBs can be optimized to omit the unnecessary data copying, which would provide improvements most noticeable by larger data sizes. Some optimizations could be achieved with the techniques for optimized range checking, described in [26]. With the careful internal design the count of the unnecessary local ORB method invocation can be minimized.

- Implementation based optimizations such as minimizing the invocation overhead of frequently called methods with the optimization for the common case, the replacement of large methods with efficient small special purpose methods, avoiding the repeated computation of invariant values, inlining and storing redundant data all provide performance improvements.

These and additional optimizations have been applied to the RMI-IIOP post-beta builds. In the next section we will show the performance improvements achieved. Unfortunately it was not possible to give a detailed description of the optimizations, since the RMI-IIOP project is a joint project between IBM and Sun. We do
not however present the improved results for RMI because our domain was the RMI-IIOP development only. It would however be interesting to apply these optimizations in future and present the improved results.

6. Performance Improvements of RMI-IIOP Post-Beta Releases

The beta release of RMI-IIOP had acceptable performance for a technology preview, but as has been described in detail in the previous sections, improvements were needed for the production release. In this section we present the improved performances of the post-beta builds 7b, 8c, FCS2b and the final release version 1.0.

The test cases split into two groups - the primitive data types, where performance in the beta is of the order of twice as slow as RMI-JRMP, and the arrays and sequences where there is a strong correlation between data length and increasing degradation. RMI-IIOP beta also showed an over-average performance degradation for compound data types, especially for arrays of `testStruct`. All areas have improved significantly since the beta, as the results below will demonstrate. In the case of the basic data types, performance of RMI-IIOP is generally at least as good as RMI-JRMP. For the complex data types, larger lengths show greatly improved performance.

6.1. Post-Beta Performance Measurements

These measurements were made on a faster (400 MHz) Pentium II processor with 196 MB RAM, and Windows NT 4.0 Workstation operation system, in a different test environment that the beta measurements, and so should not be compared directly with the earlier measurements. For ease of comparison, the results presented here are for a single client running on the same machine as the server. This removes network effects, and readily illustrates the combined effects of improvements in both the client and server components. To illustrate the dynamic effects of developments since the beta, measurements from several builds are included. The comparison point, and base JVM for all measurements is the Java 2 SDK, Standard Edition, v1.2.
6.2. Post-Beta Performance Analysis

Earlier data demonstrated that in the single machine measurement, these test cases drive the CPU to 100% utilization, so it is reasonable to equate reductions in RTT with improvements in efficiency that have reduced instruction pathlength.

The basic data type test cases all show very similar RTTs, and similar RTT improvements. This is most evident in moving from the beta release to build 7B, with a further small improvement in subsequent versions in most cases. Figure 16 shows the geometric means of basic data types and RTTs for compound types \texttt{testStruct} and object reference. For basic data types RMI-IIOP post-beta 8C performs on average around 5%, FCS2B around 11% and the final version around 7% better than RMI. The compound data types have shown variations. For \texttt{testStruct}, build 7B gave a large improvement, with a minimal further setback in 8C. Build FCS2B and the final version improved the performance again to give performance within 25% of RMI. For \texttt{myObject}, build 7B clearly regressed performance by around 4 fold from the beta, but this was recovered in the subsequent versions to give RMI-IIOP times that are over 40% faster than RMI.

![Figure 16](image-url)

\textit{Figure 16}  Performance Improvements for Scalars and Arrays with One Element for Average of Basic Data Types, testStruct and Object Reference
In previous section we have investigated the performance for one element arrays and have compared them to primitive data types. Especially one-element arrays of \texttt{testStruct} and \texttt{myObject} showed poor performance. Figure 16 also shows performance improvements for handling one element arrays. For basic data types there is a large improvement from beta to version 7B. Again, version 8C shows a little drawback and FCS2B and final version another small improvement. The performances compared to RMI-IIOP beta are improved for more than 30% and the final version of RMI-IIOP is only $\sim$15% slower for single element arrays of basic data types than RMI. The greatest improvements have been achieved by the \texttt{testStruct}, where the build 8C is only 12% and the final version only 10% slower than RMI. The performance of \texttt{myObject} is improved too, but not as much as the \texttt{testStruct}. Although the final version is $\sim$15% faster than beta, it is still $\sim$40% slower than RMI.

To present the impacts of data size Figure 17, Figure 18, and Figure 19 present the throughput for different array sizes, up to 16384 elements. Figure 17 shows the geometric average throughput for arrays of basic data types. Post-beta 7B already improved performance over the beta RMI-IIOP. Version 8C and FCS2B achieved better performance than RMI. Although the final version is slightly slower than FCS2B it is still faster than RMI. Figure 18 shows the throughput for \texttt{testStruct}, where the post-beta versions continuously improved performance. This time however the RMI performance is not achieved – RMI-IIOP is less than 30% slower. We have already discussed the problems with the \texttt{testStruct}, which has to be transferred as CORBA object-by-value, in previous sections. Figure 19 presents the throughput for arrays of object references. We have already identified a performance drawback by the 7B build and the slower results for the RMI-IIOP final version for one element arrays. The 8C, FCS2B and the final version however manage to offer slightly better performance than RMI for large arrays. Because the RMI-IIOP has to transfer the repository id the single element arrays shows the overhead that is compensated with the larger number of elements.
Figure 17  Comparison of Average Throughput for Basic Data Types in Dependence to Array Size

Figure 18  Comparison of Average Throughput for the testStruct in Dependence to Array Size
7. Concluding Remarks

The increasing demand for application connectivity and interoperability intensifies the role of distributed object models in all areas of application development. With the use of distributed object models in mission critical applications the performance question becomes more and more important. In this paper a detailed performance analysis of important distributed object models for the Java has been done: RMI and the newly developed RMI-IIOP, which enables the use of RMI over the IIOP protocol and makes it CORBA compliant. Performance has been measured for relevant usage scenarios which includes single and multi-client interactions with up to eight simultaneous clients, different basic and compound data types and different data sizes. In the performance comparison RMI provided the best results in all tests\(^9\). Although for the basic data types the results differed by a factor of \(~2\) the larger data sizes showed significant differences. The maximum throughput achieved by RMI was \(~38\text{Mbps}\), which is more than a third of the theoretical network throughput. RMI-IIOP beta achieved only \(~12\text{ Mbps}\). In multi-client scenarios RMI-IIOP beta was from \(~75\%\) to \(~100\%\) slower than RMI.

\(^9\) But note the significant improvements in later versions of RMI-IIOP gave equivalent or better performance than RMI in most cases.
The overhead analysis showed that the bottlenecks fall in the following categories: (1) the overhead of the low-level methods that take care of socket based communication, (2) the inefficient thread management, (3) ineffective algorithms for demultiplexing and demarshalling, (4) excessive data copying and local method invocations. Proposed optimizations were applied and performances remeasured for the post-beta RMI-IIOP builds and the final release version. The presented results for the final RMI-IIOP version show performance that is always comparable and in many times even superior to the RMI. Although RMI-IIOP uses the standardized IIOP protocol for which it has to make additional transformations and mappings, the careful design and implementation of the model, together with careful performance analysis and implementation of several optimizations, have enabled excellent performance. We have seen that the remote method invocation performance is critical in large distributed systems. RMI-IIOP provides a valuable alternative which will make Java objects interoperable with any CORBA object while providing excellent performance and relative ease of development.

* Trademark or registered trademark of International Business Machines Corporation.
** Trademark or registered trademark of Sun Microsystems, Inc., The Object Management Group, The Open Group, etc.

Acknowledgements

We are grateful to Simon Nash, (IBM Hursley Park) Technical Leader of the IBM-Sun RMI-IIOP project for suggesting this paper, to Stephen Pipes (IBM Hursley Park) for carrying out the measurements on the post-beta builds, to Bostjan Brumen (University of Maribor), Tomaz Domajnko (University of Maribor) and Ales Zivkovic (University of Maribor) for helpful suggestions and for help with the measurements.

References


[22] Object-Oriented Concepts, ORBacus For C++ and Java, 1999

[23] Juric B. Matjaz, Domajnko Tomaz, Zivkovic Ales, Herricko Marjan, Brumen Bostjan, Welzer Tatjana, Rozman Ivan, Performance Assessment Framework for Distributed Object Architectures, accepted for publication in Conference proceedings of ADBIS'99 in the LNCS series of Springer Verlag as full paper


Matjaz B. Juric, University of Maribor, Institute of Informatics, Faculty of Electrical Engineering, Computer and Information Science, Smetanova 17, 2000, Maribor (electronic mail: matjaz.juric@uni-mb.si) Mr. Juric is a researcher at the University of Maribor. His interest include many areas of object technology with emphasis on distributed object models, performance analysis and optimization, the Java platform, and network computing. He is the author of more than 80 bibliographic units that include journal and conference articles, books, and educational material.

Alan P. Stevens, IBM United Kingdom, Hursley Park, Winchester, Hants S021 2JN, United Kingdom (electronic mail: alan_stevens@uk.ibm.com) Mr. Stevens is the Performance Technical Planner with responsibility for Java Performance at the IBM Java Technology Centre. He is involved in planning performance for IBM Java products developed in Hursley, working with a variety of IBM platform performance and research teams. His interests include performance tools and instrumentation, and he worked with colleagues from Sun Microsystems on the definition of the Java Virtual Machine Profiling Interface (JVMPI). Mr. Stevens joined IBM in 1988 and prior to his Java involvement, was a performance analyst.
specialising in the CICS transaction processing monitor on PC and UNIX systems. He received a B.Sc in Psychology from Bristol University in 1980.

**Ivan Rozman, University of Maribor, Institute of Informatics, Faculty of Electrical Engineering, Computer and Information Science, Smetanova 17, 2000, Maribor (electronic mail: i.rozman@uni-mb.si)** Prof. Rozman is an university professor at the University of Maribor. His interest cover the software engineering management, software development methodologies, software process and software quality issues. He is author or co-author of two books and ten original scientific papers published in international scientific journals, many conference papers, professional papers and project reports.

**Marjan Hericko, University of Maribor, Institute of Informatics, Faculty of Electrical Engineering, Computer and Information Science, Smetanova 17, 2000, Maribor (electronic mail: marjan.hericko@uni-mb.si)** Dr. Hericko is an assistant professor at the University of Maribor. Dr. Hericko is an assistant professor at the University of Maribor. His research interests include all aspects of object technology with emphasis on development methods and tools, quality assurance and object oriented metrics.