Java 2 Remote Method Invocation Performance Analysis, Comparison and Optimization for RMI, RMI-IIOP, and IDL

Matjaz B. Juric, Alan P. Stevens, Ivan Rozman, Marjan Hericko

Abstract

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 three most important distributed object models for Java: RMI (Remote Method Invocation), RMI-IIOP beta (Remote Method Invocation over Internet Inter-ORB Protocol), and IDL (Interface Definition Language). The paper presents the following contributions to the research on distributed object performance. First, a detailed performance analysis of the three 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 are presented and the results for performance improved post-beta RMI-IIOP versions. These show considerably better performance in all areas compared to the original beta release, with RMI-IIOP having equivalent or better performance to RMI-JRMP in almost all cases.

Keywords: Java, RMI, IIOP, CORBA, performance

1. Introduction

Java has established itself 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 away from programming 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 too 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.

Unfortunately 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 Inter-ORB 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.

In this article we focus on performance assessment, comparison and optimization of the three distributed object models suitable for use with Java. 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. We define the performance measurement
procedure and report and compare the results for RMI, RMI-IIOP beta\(^1\) and IDL. 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. These results depict why there are differences in performance. Finally, we present general
performance optimizations for all three models and specific optimizations for RMI-IIOP. We show the improved
performance for RMI-IIOP post-beta versions. This article forms a basis for building an analytical performance
model for distributed object models. In this article 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 \[4\].

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. The
paper also describes Ace. 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.

\(^1\) RMI-IIOP beta 0.2 (dated Dec 18, 1998) has been used.
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. This paper focuses on distributed object models for Java 2 platform: RMI (Remote Method Invocation), IDL (Interface Definition Language) and RMI-IIOP. The results give an insight into performances of evaluated models and present a solid basis for the developer to make a responsible decision about which distributed object model to use. The explanation of the results reveals the sources of the overhead and gives optimization solutions.

As a consequence of this performance analysis and bottleneck identification several performance improvements were made to RMI-IIOP after the beta version has been released. In this paper we present the improvements collected in the RMI-IIOP post-beta builds 7b, 8c, and FCS2b. It is expected at the time of writing that final released version will be based on post-beta FCS2b. (Note to reviewers - the final version will be built in the next few days - it would be very helpful to be able to update this with data from the released code, given that this article, if accepted, will not appear until many months after the RMI-IIOP release).

The contributions of this paper are: (1) Detailed performance analysis of RMI, IDL, and RMI-IIOP covering the majority of usage scenarios among other with different basic and user defined data types, different data sizes and multi-client scenarios. (2) Comparison of the three models and the identification of points where the performance should be improved (especially regarding, but not limited to RMI-IIOP). (3) Detailed explanation of performance results and identification of the source code that caused a certain behavior that has reflected in performance analysis. (4) Optimization techniques for improvement of identified bottlenecks, and (5) the presentation of an performance-improved post-beta version of RMI-IIOP as a consequence of performance analysis and bottleneck optimizations.

The paper is organized as follows: Section 2 globally describes the remote method invocation mechanism, compares RMI to CORBA, JRMP (Java Remote Method Protocol) to GIOP (General Inter-ORB Protocol), identifies the goals and possible solutions for RMI-IIOP, and explains the role of concurrency models supported by ORBs (Object Request Brokers). Section 3 presents the performance evaluation method, with the goals, performance criteria, theoretical background, implementation details, and test-bed equipment. Section 4 gives a
detailed overview of the performance results for RMI, IDL, and RMI-IIOP, compares the results, and identifies their source. Section 5 presents a detailed overhead analysis, bottleneck identification and techniques for performance improvement. Section 6 presents the performance of post-beta RMI-IIOP after implementing the optimizations. Sections 7 gives the concluding remarks.

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

- **Platform and programming language**: 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**: 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.

![Remote Method Invocation Path](image_url)

*Figure 1  Remote Method Invocation Path*
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]. ORB 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.

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)

Java RMI utilizes strict separation of interfaces and implementation, although the interfaces are specified and the functionality is implemented in Java. Because RMI is fully integrated with JVM some unique features are supported 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\(^2\). When compared with CORBA it can be seen that RMI provides only basic ORB functionality.

Figure 2 shows the three independent layers that constitute the RMI system [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\(^3\).

• 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.

\(^2\) However the interoperability can be achieved using Java Native Interface for example.

\(^3\) For performance evaluation the 1.2 compliant stubs have been used.
2.2. Java Remote Method Protocol (JRMP)

For the wire communication Java Remote Method Protocol (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 invocation data and obtain results. Usually the RMI transport layer opens direct sockets to hosts. Such connection 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:

- Call,
- Ping,
- DgcAck,
- ReturnData,
- HttpReturn and
- PingAck.

2.3. Common Object Request Broker Architecture (CORBA)
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:

- **Object request broker** (ORB) is the integral component of the architecture. It hides all the details of the communication between the two objects – the client and the server object. ORB is the object bus.

- **Object services** (CORBA services) define the system level object frameworks that widen the range of basic functions of the ORB with services such as naming, event, life-cycle, transaction, relation, etc.

- **Common facilities** (CORBA facilities) define the horizontal application frameworks that are used by application objects.

- **Domain facilities** define application frameworks for different domains such as healthcare, financial institutions, manufacturing, etc.

- **Application objects** are the applications actually developed by the developers.

In this article we will focus only on the CORBA object request broker. 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\(^4\).

CORBA supports static and dynamic method invocation, one and two way operations and synchronous and deferred synchronous invocation mechanisms. A specification for asynchronous invocation is also being prepared. It supports different parameter passing modes (in, out, and inout), dynamic server implementations, persistent naming and persistent object references. It has an interface and an implementation repository and an independent wire protocol. However some features characteristically for Java are not supported by CORBA, for example passing objects by value\(^5\), dynamic class downloading and dynamic stub downloading. In CORBA there is also no distributed garbage collection or URL based naming.

\(^4\) Currently C, C++, Smalltalk, Java, Ada, COBOL and PLI mappings are standardized by OMG. For other languages non-standardized mappings exist.

\(^5\) An object-by-value specification is on the way and will be included in CORBA 3.0. The main reason for this specification has been enabling the RMI to work over the IIOP.
Figure 3 shows the architecture of a typical CORBA compliant object request broker:

- 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.

- 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.

2.4. General and Internet Inter-ORB Protocol (GIOP/IIOP)
The General Inter-ORB Protocol (GIOP) is a language independent wire protocol for inter ORB communication. 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].

### 2.5. RMI-IIOP

A common goal was to merge the two distributed object models, RMI and CORBA, in a way that RMI clients would be able to access CORBA servers and CORBA clients would be able to access RMI servers transparently. The language syntax should remain unchanged as much as possible. There are two possibilities to achieve the stated goals:

- Build a bridge between both protocols or
- Provide a consistent mapping from Java to IDL.

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 when the objects are transferred as objects-by-value, as we will show in the next section. There is also a problem with handling IDL “in” and “inout” parameters that are not supported in RMI. Providing consistent handling of IDL data type “any” is also not easy. RMI on the other hand supports transferring objects by value, but CORBA 2.2 does not. Therefore CORBA 3.0 will be extended by an object-by-value specification.

2.6. 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 [19] and offer acceptable performances [20]. 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, IDL and RMI-IIOP. They all handle incoming requests in parallel. We have verified this with a test described in [21]. Unfortunately in the documentation we were unable to find the details of the threading model, therefore we assume they all use the thread-per-request. Concurrency models used by RMI, RMI-IIOP, and IDL are described in detail in section 5.

3. Performance Evaluation

3.1. Goals
Our goal was to evaluate the performance of Java IDL, 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 three (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 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 all three 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 analysing 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 [21] 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, RMI-IIOP and IDL is achieved with the identical implementation that differs 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.

### 3.3. Theoretical Background

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. Depending on the concurrency model used by the underlying distributed object model the systems fall in two general groups: the systems that process incoming requests parallel and the systems that serialize them (see section 2.6).

The system used here processes the requests in parallel and the clients wait for the response before submitting new request. 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 (see [22]).

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.

### 3.4. Implementation Details

To measure RTT, interfaces (Listing 1 and 2) with corresponding classes have been defined on the server side for each data type listed in Table 1. Each interface provides two methods. In both methods the processing overhead has been omitted.

```java
public interface <data_type>TestServer extends Remote {
    void acceptType(<data_type> Value) throws RemoteException;
    <data_type> char returnType() throws RemoteException;
}
```
Listing 1: Template for Java RMI server side interfaces

```java
interface <data_type>TestServer {
    void acceptType(in <data_type> Value);
    <data_type> returnType();
};
```

Listing 2: Template for CORBA/Java IDL server side interfaces

<table>
<thead>
<tr>
<th>Java RMI data type</th>
<th>Length in bits&lt;sup&gt;6&lt;/sup&gt;</th>
<th>CORBA IDL data type</th>
<th>Length in bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>8</td>
<td>boolean</td>
<td>8</td>
</tr>
<tr>
<td>char</td>
<td>16</td>
<td>char</td>
<td>8</td>
</tr>
<tr>
<td>byte</td>
<td>8</td>
<td>octet</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>short</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>long</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>long long</td>
<td>64</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>float</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>double</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 1: Simple data types used for performance evaluation

Table 1 also presents the equivalent mapping between Java (RMI) and CORBA (IDL) data types. Although the over-the-wire length of Java RMI char and IDL char is different we have decided to compare them. This is because Java IDL does not support the wchar data type and native Java does not support 1 byte characters. In Java, strings are represented using the `java.lang.String` class. Internally they are represented as sets of Unicode 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, which is not supported by Java IDL. Therefore we have decided to compare them although we should note that they are not strictly equivalent.

In addition to simple data types two user defined data types are introduced. First, the IDL structure `testStruct`, which is presented in Listing 3. In RMI the structure is implemented as a final class. It implements the serialization interface (Listing 4) which is why it can be transferred by value. It is important to understand that for IDL the `testStruct` is a simple data structure while for the RMI it is an class (object). Therefore the RMI has to use the object-by-value transfer of parameters. Even more work has to be done in RMI-IIOP which has to adapt the object to the CORBA object-by-value specification which it already supports.

---

<sup>6</sup> When transferred over the wire.

<sup>7</sup> Java IDL supports only 8 bit characters. Although CORBA specifies a wchar type which can hold 16 bit characters it is not implemented in Java IDL. When transferred over the wire the 16 bit Java Unicode characters are truncated to 8 bits.
struct testStruct {
    boolean b;
    octet o;
    short s;
    long l;
    long long ll;
    float f;
    double d;
};

Listing 3: User defined structure testStruct defined in IDL

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 {
        out.writeBoolean(b);
        out.writeByte(o);
        out.writeShort(s);
        out.writeInt(l);
        out.writeLong(ll);
        out.writeFloat(f);
        out.writeDouble(d);
    }

    private void readObject(java.io.ObjectInputStream in) throws java.io.IOException, ClassNotFoundException {
        b = in.readBoolean();
        o = in.readByte();
        s = in.readShort();
        l = in.readInt();
        ll = in.readLong();
        f = in.readFloat();
        d = in.readDouble();
    }
};

Listing 4: testStruct implemented in Java RMI

Second, a user defined class is introduced, named myObject. The definition is shown in Listing 5 and 6 for RMI and IDL respectively. myObjects are transferred by reference in IDL, RMI, and RMI-IIOP. They all have to send the object reference. RMI-IIOP has to change it’s internal reference to the CORBA IOR (Interoperable Object Reference).

public interface myObject extends Remote {
    byte[] a() throws RemoteException;
    void a(byte[] arg) throws RemoteException;
}

Listing 5: myObject definition in RMI

interface myObject {
    attribute octetSeq a;
};

Listing 6: myObject definition in IDL

To be able to measure RTT for different sizes of method parameters and return values new interfaces are introduced. In contrast to the interfaces presented in Listing 1 and 2 the new interfaces deal with arrays and
sequences of all the described data types for RMI and IDL, respectively. We have not used the IDL arrays because they are of fixed size. Both IDL arrays and IDL sequences map to Java arrays. Because Java arrays are also variable length it is a fair comparison. The sequences for CORBA IDL and arrays for RMI are defined as follows:

```c
typedef sequence<data_type> (data_type)Seq;  // IDL
(data_type[]) ValueSeq;  // RMI
```

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 [21].

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 \( X_i, i=1, 2, \ldots, n \), the following steps are necessary:

- the client waits to get the synchronization acknowledgement,
- it reads the system time,
- it performs the method invocation. To get the desired resolution of the result the test is performed \( r \) times,
- it reads the system time again and calculates the \( X_r \)-th performance observation.
After completing the $n$ observations, sample mean $\bar{X}$, the variance $s^2$ and standard deviation $s$ are calculated. A skeleton of the client side application is shown in Listing 7.

```java
if(size>0) {
    testPattern= new myObject[size];
    for(n=0;n<size;n++) {
        testPattern[n]=new myObjectImpl();
    }
}
for (i=0;i<noIter;i++) {
    if(size==0) {
        startTime=System.currentTimeMillis();
        for (r=0;r<noRep;r++) {
            obj.acceptType(dummy);
        }
        stopTime=System.currentTimeMillis();
    } else {
        startTime=System.currentTimeMillis();
        for (r=0;r<noRep;r++) {
            objs.acceptType(testPattern);
        }
        stopTime=System.currentTimeMillis();
    }
    data[i][0]=(stopTime-startTime);
}
for (i=0;i<noIter;i++) {
    if(size==0) {
        startTime=System.currentTimeMillis();
        for (r=0;r<noRep;r++) {
            dummy=objs.returnType();
        }
        stopTime=System.currentTimeMillis();
    } else {
        startTime=System.currentTimeMillis();
        for (n=0;n<noRep;n++) {
            testPattern=objs.returnType();
        }
        stopTime=System.currentTimeMillis();
    }
    data[i][1]=(stopTime-startTime);
}
data[i][2]=(data[i][0]+data[i][1])/2
```

Listing 7: The skeleton of the client side application

### 3.5. 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) and for post-beta builds 7b and 8c. 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 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, IDL, 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, IDL, and RMI-IIOP. First we present results for the single-client scenarios and then for the multi-client scenarios. Each series of results is presented in the following order: RMI results, IDL results, and RMI-IIOP results. Where possible, same scales for graphical presentation have been used. In each section we give a short explanation of the results.

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

![Figure 4](image)

**Figure 4  Comparison of RTT for primitive data types**

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). This is mainly because of the complexity of RMI data flows (marshalling, demarshalling, demultiplexing) and the necessary low-level transport and context switching activities.

In Figure 4 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) IDL is 80% slower and RMI-IIOP 66% slower than RMI. RMI-IIOP is, however, 8% faster than IDL. As already mentioned the testStruct is handled differently by RMI, IDL, and RMI-IIOP. For RMI and RMI-IIOP it is a class and for IDL it is a data structure. We can see, however, that IDL is 45% slower than RMI and RMI-IIOP almost 150% slower. RMI-IIOP is also almost 70% than IDL. For the object reference (myObject) the results are closer. IDL is 10% and RMI-IIOP 20% slower than RMI. If we take a look at the string we can see that IDL and RMI-IIOP are 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 IDL and RMI-IIOP introduce more layers and are not so well integrated into Java, therefore they are slower. The much slower result for the testStruct in RMI-IIOP is a consequence of the transmission technique used. In RMI-IIOP testStruct is transmitted as (CORBA) object-by-value and in IDL as a structure. 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 obtain the CORBA IOR (Interoperable Object Reference).

4.1.2. Primitive data types vs. arrays with one element

Figure 5, Figure 6, and Figure 7 show the comparison between primitive data types and arrays/sequences with one element. 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.

![Figure 5](image_url)  
 Figure 5  RMI: RTT for primitive data types vs. arrays with one element
With RMI the overhead for handling arrays is 8% for basic data types, 5% for testStruct and 2.5% for object reference. IDL shows 15% overhead for basic data types, 21% for testStruct and 29% 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 as well as in IDL the arrays are serialized so that first the array length is written and then each element is serialized. In all three models a “for” loop is used. For demarshalling a local array is created which leads to a data copy process.
4.1.3. Impact of Data Size on RTT

Figure 8, Figure 9, and Figure 10 show the RTT for different data sizes for RMI, IDL and RMI-IIOP. 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 all three graphs and a different y scale for IDL results.

Figure 8  RMI: Influence of data size on RTT

Figure 9  IDL: Influence of data size on RTT
First we can see that RMI achieved the fastest RTTs. 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:

- IDL is 5.6 times slower than RMI and
- RMI-IIOP is 2 times slower.

More interesting is the behavior for 4096 characters string where IDL is 23% and RMI-IIOP 38% slower than RMI.

The comparison is not fair, however. The IDL uses strings with 8-bit characters while RMI and RMI-IIOP use strings with Unicode characters. The reason why IDL is not even faster can be discovered if we take a look at the multiplexing and demultiplexing code. In both cases in IDL range checking is implemented which slows down the performance. In the demultiplexing process first an array of bytes is read. Then the Java string is constructed. Because of that we have excessive data copying which influences the performance especially with larger data sizes.

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

---

8 Java IDL does not support IDL data types wchar and wstring.

9 An exception is raised if a character is out of range [0, 255].
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 IDL the testStruct is a structure and only the attribute values have to be transmitted. This is why the results are comparable. 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 both IDL and RMI-IIOP have to use the CORBA compliant IOR (Interoperable Object Reference). Here another transformation takes place. 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 we shall look at Figure 11, Figure 12, and Figure 13 where the throughput in Mbits/sec for basic data types is presented. 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 takes the second place with peak values around 18.5 Mbps and IDL reaches only 11.8 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. IDL on the other hand shows the highest throughput at strings, but you should be aware that IDL strings are composed of 8-bit characters.
Figure 11  RMI: Influence of data size on throughput for basic data types

Figure 12  IDL: Influence of data size on throughput for basic data types
The reason for IDL to achieve such a low throughput for large data sizes is the following: at the multiplexing process a buffer of an initial size 1024 bytes is used. This buffer is then enlarged by factor two (class CDROutputStream, method grow). By aligning the data types to the corresponding boundaries prescribed by GIOP protocol [2] every time buffer size checking is performed (method alignAndReserve).

Although IDL achieves the best throughput by strings, a disadvantage can be observed by demultiplexing strings. Strings are first read as arrays of bytes and then converted to the actual java.lang.String representation. With larger string this means excessive data copying and memory overhead which has bad influence on performance and is also the reason why IDL does not achieve higher throughput with strings. This is an additional burden for the garbage collection. For all three models the garbage collection is performed on local objects, for RMI the Distributed Garbage Collection algorithm takes care for distributed object as well.

RMI and RMI-IIOP java.lang.Strings on the other hand are objects. Transferring objects between different JVMs is more costly than arrays of characters and this is the reason why by RMI and RMI-IIOP arrays of characters achieve the maximum throughput.

Figure 14 presents the geometric average throughput for basic data types (boolean, char, byte/octet, short, int/long, long/long long, float, double, and string). Results for RMI, IDL, and RMI-IIOP are presented. IDL peak throughput
is more than 7 times lower than RMI and more than 3 times lower than RMI-IIOP. RMI-IIOP throughput is 2.3 times lower than RMI.

Figure 14  Comparison of geometric average throughput for basic data types

Figure 15 shows the throughput for the structure testStruct for RMI, IDL and RMI-IIOP. It is especially noticeable that the throughput of RMI-IIOP is an order of magnitude lower than RMI and IDL. 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 object-by-value specification\textsuperscript{10}. This means that major transformations are needed. IDL on the other hand treats testStruct as a simple data structure and has to transfer only the data. Improving the RMI-IIOP performance in this point is a major task before bringing out the final version.

\textsuperscript{10} OMG Object-by-Value specification is still in progress by the time of writing this article.
Figure 15 Comparison of throughput for the testStruct

Figure 16 shows the throughput for object reference myObject for RMI, IDL and RMI-IIOP. Object reference is handled by all the architectures similar. RMI achieves more than two times larger throughput than IDL. You should be aware however that RMI has to deal with native Java object references, IDL and RMI-IIOP have to use the CORBA IORs instead.

Figure 16 Comparison of throughput for myObject

4.1.4. Analysis of Single Computer Versus Two Distributed Computers Results
The graphs on Figure 17, Figure 18, and Figure 19 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 17 presents the average throughput for basic data types (boolean, char, byte/octet, short, int/long, long/long long, float, double, and string), Figure 18 for testStruct and Figure 19 for object reference (myObject).

![Figure 17](image1.png)  
**Figure 17**  
*Average Throughput Comparison for Basic Data Types: Single Computer vs. Two Distributed Computers*

![Figure 18](image2.png)
For primitive data types 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 IDL and RMI-IIOP are faster on two computers – from 3% to 5% on average.

When comparing arrays and sequences 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. The results presented here show the IDL scenario; the results for RMI and RMI-IIOP are similar. In Figure 20 we can see the CPU usage for client and server placed on single computer. We can see that the peak is at 100%. Distinct method invocations can clearly be seen. The two JVM occupy all the available CPU resources.
Figure 20  CPU Usage when Client and Server Objects are Located on a Single Computer
In Figure 21 and Figure 22 we can observe the CPU usage when the client and the server objects are moved to the separate computers. 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. 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.

4.2. Multi Client Scenarios

4.2.1. Primitive Data Types
In Figure 23 to Figure 25 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. All three models use the same threading model and execute each client’s request in a separate thread (see section 2.6). By all three models we can observe very similar behavior of basic data types. This is especially noticeable by RMI and RMI-IIOP. TestStruct and myObject show deviations where by RMI and IDL the myObject shows the largest times, and by RMI-IIOP the testStruct.

Figure 23  RMI: RTT per client for primitive data types in multi-client scenario

Figure 24  IDL: RTT per client for primitive data types in multi-client scenario
RMI achieves the fastest times, followed by IDL and RMI-IIOP. By eight simultaneous clients, IDL is for basic data types 75% slower than RMI and RMI-IIOP is around 100% slower than RMI. RMI-IIOP is only 15% slower than IDL. For the testStruct IDL is 33% and RMI-IIOP 3 times slower than RMI. For the object reference the IDL and RMI achieve almost the same results, RMI-IIOP is around 10% slower. Although the RMI-IIOP beta version achieves comparable results, optimizations will be applied especially to the testStruct handling performance.

Better insight into performance degradation is presented in Figure 26, Figure 27, and Figure 28. In Figure 26 the performance degradation for basic data types is compared between RMI, IDL and RMI-IIOP. It is interesting to observe that cumulative degradation for the eight client scenario is lowest for IDL (5.4), followed by RMI (5.9) and RMI-IIOP (7.4). Note however that RMI had the lowest degradation for up to six concurrent clients. On a secondary axis the chain index for performance degradation in percent is shown.
In Figure 27 the performance degradation for the testStruct is shown. The results are similar as before, except that all three models showed comparable results up to seven clients. By the eight concurrent clients the RMI-IIOP had the largest degradation (see the chain index on the secondary axis), followed by RMI. It is interesting to observe that the degradation of RTTs for IDL is practically the same for seven and eight simultaneous clients. It also seems that by the eight concurrent clients we have reached the point of RMI-IIOP model where the performance drops rapidly.
In Figure 28 the degradation for object reference (myObject) is shown. We can see that for up to seven simultaneous clients RMI had the lowest degradation (by seven simultaneous clients 4.7), followed by RMI-IIOP (5.7) and IDL (5.9). The eight concurrent client however contributed more than 40% degradation for RMI. Again, it seem 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 IDL and RMI.

![Comparison of Performance Degradation for myObject](image)

*Figure 28  Comparison of Performance Degradation for myObject*

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), and results for testStruct and myObject.
Figure 29  RMI: Average RTTs for arrays of basic data types in multi-client scenario

Figure 30  IDL Average RTTs for sequences of basic data types in multi-client scenario

Figure 31  RMI-IIOP: Average RTTs for arrays of basic data types in multi-client scenario
Figure 32  RMI: RTT for arrays of test structure in multi-client scenario

Figure 33  IDL: RTT for sequences of test structure in multi-client scenario
Figure 34  RMI-IIOP: RTT for arrays of test structure in multi-client scenario

Figure 35  RMI: RTT for arrays of object reference in multi-client scenario

Figure 36  IDL: RTT for sequences of object reference in multi-client scenario
Figure 35, Figure 36, and Figure 37 show the geometric mean response (RTT) times for basic data types for RMI, IDL, and RMI-IIOP, respectively. Note: the scales are identical for easier comparison. The absolute performance is the best for RMI, followed by RMI-IIOP and IDL. For large data sizes and many concurrent clients IDL is more than two times slower and RMI-IIOP is around 58% slower than RMI. It is interesting to observe that IDL gets consistently slower than RMI-IIOP for all data sizes larger than 1024 bytes (measuring their binary length as transferred over-the-wire). This result confirms the buffer-sizing bottleneck already identified in previous subsection.

We have already observed the rapid performance degradation for arrays of compound data types (testStruct and myObject) in the single client scenario. The results on graphs from Figure 32 to Figure 37 confirm this in the multi client scenario as well. Therefore the results are reported for smaller data sizes only. For the testStruct we can however see (Figure 32, Figure 33, Figure 34), that the IDL reached the overall fastest response times, followed by RMI and RMI-IIOP. We have already identified the RMI-IIOP ineffectiveness by handling arrays of testStructs, manly because of using the CORBA object-by-reference specification. The explanation, why IDL is faster than RMI hides in the fact, that RMI handles the testStruct as a class, therefore it has to transmit the objects by value. For IDL, the testStruct is a simple data structure. It is also important to notice, that the response times were measured for arrays of testStruct for up to 1024 elements only, therefore not too much buffer resizing took place by IDL.
In Figure 35, Figure 36, and Figure 37 the response times for arrays of object references are shown. The disadvantage of RMI-IIOP is clearly noticeable. Note: the performance improvements will be presented in the next section. The performance of RMI and IDL is comparable. In single-client scenario we have seen that RMI handled large arrays of object references better than IDL. In multi-client scenario we have seen that one instance of object reference is handled similar by RMI and IDL and the relative performance degradation is comparable, although RMI showed a considerable worsening when going from seven to eight concurrent clients. This situation is reflected in this scenario as well. On Figure 35, and Figure 36 we can observe faster RTTs for RMI by few concurrent clients, comparable by 5, 6, and 7 concurrent clients and slower by 8 concurrent clients.

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 by RMI and RMI-IIOP 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. By IDL the relative performance degradation drops with the larger data sizes.

5. Overhead Identification and Optimizations

In section 4 we presented the external performance results for RMI, IDL, 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/a sequence of bytes/octets. Therefore the
client object had to marshal the parameter, which gets transferred to the server object, where it is 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 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></td>
<td>Data size in elements:</td>
</tr>
<tr>
<td>java.net.SocketOutputStream.socketWrite</td>
<td>19.04</td>
</tr>
<tr>
<td>java.io.ObjectOutputStream.&lt;init&gt;</td>
<td>4.34</td>
</tr>
<tr>
<td>java.io.ObjectInputStream.read</td>
<td>3.54</td>
</tr>
<tr>
<td>java.io.BufferedInputStream.read</td>
<td>3.36</td>
</tr>
<tr>
<td>sun.rmi.transport.tcp.TCPTransport.handleMessages</td>
<td>3.27</td>
</tr>
<tr>
<td>java.lang.StringBuffer.append</td>
<td>2.92</td>
</tr>
<tr>
<td>java.lang.String.getChars</td>
<td>2.30</td>
</tr>
<tr>
<td>java.lang/System.arraycopy</td>
<td>2.21</td>
</tr>
<tr>
<td>java.io.BufferedInputStream.ensureOpen</td>
<td>2.12</td>
</tr>
<tr>
<td>java.lang.reflect.Method.toString</td>
<td>1.68</td>
</tr>
<tr>
<td>java.io.DataInputStream.readInt</td>
<td>1.59</td>
</tr>
<tr>
<td>java.io.ObjectInputStream.allocateNewArray</td>
<td>&lt; 1.00</td>
</tr>
<tr>
<td>java.net.SocketInputStream.socketRead</td>
<td>&lt; 1.00</td>
</tr>
<tr>
<td>java.net.PlainSocketImpl.socketAvailable</td>
<td>&lt; 1.00</td>
</tr>
</tbody>
</table>

Table 2: Analysis of server 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.
Table 3 presents the client side overhead for RMI. Again the percentages of the total execution times spent in the corresponding methods are shown. Data sizes 1, 1024, 4096, and 16384 are presented.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>% of time spent in the method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data size in elements:</td>
</tr>
<tr>
<td>java.net.SocketOutputStream.socketWrite</td>
<td>19.95</td>
</tr>
<tr>
<td>java.net.InetAddressImpl.lookupAllHostAddr</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.lang.String.charAt</td>
<td>2.06</td>
</tr>
<tr>
<td>java.io.ObjectOutputStream.&lt;init&gt;</td>
<td>1.60</td>
</tr>
<tr>
<td>java.io.ObjectOutputStream.setBlockData</td>
<td>1.60</td>
</tr>
<tr>
<td>java.io.ObjectOutputStream.writeObject</td>
<td>1.60</td>
</tr>
<tr>
<td>java.io.BufferedOutputStream.write</td>
<td>&lt; 1.00</td>
</tr>
<tr>
<td>java.io.DataOutputStream.writeUTF</td>
<td>&lt; 1.00</td>
</tr>
</tbody>
</table>

Table 3: Analysis of client side overhead for RMI

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\(^\text{11}\), which returns the current time in milliseconds, and pairs it with the Internet address.

5.1.2. IDL

To process the client requests the IDL server object creates a pool of threads. The Java IDL Listener Thread handles the low level connection details. For each client a Java IDL Reader Thread is created which handles the connection with a specific client. For each client also a pool of 201 Worker Threads is created. These threads process the IIOP protocol input streams. The server side overhead is shown in Table 4, Table 5, and Table 6. Table 4 shows the server side overhead for the Listener Thread, Table 5 for the Reader Thread, and Table 6 for the

---

\(^{11}\) 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.
Worker Thread. In the Listener Thread ~5% of the total time is spent, in the Reader ~80% and in a single Worker thread ~15%.

### Method Name | % of time spent in the method
--- | ---
java.net.PlainSocketImpl.socketAccept | 27.77 39.21 45.33 57.84
java.net.InetAddressImpl.getHostByAddr | 22.22 18.35 11.69 8.50
java.lang.String.charAt | 11.11 7.32 4.61 2.10
java.lang.Class.newInstance | < 1.00
java.net.Socket.getPort | 11.11 3.10 < 1.00 < 1.00
com.sun.CORBA.iiop.ListenerThread.run | 5.55 4.77 3.92 3.08

Table 4: Analysis of server overhead for the Listener Thread for IDL

The method socketAccept which spends ~28% of total time, is a native method of the default socket implementation. Another native method is the getHostByAddr. Note that around 17% of the total time is spent in thread management.

### Method Name | % of time spent in the method
--- | ---
java.net.SocketInputStream.socketRead | 58.86 45.94 57.6 56.04
java.lang.Thread.start | 39.03 48.96 40.35 24.95
com.sun.CORBA.iiop.IIOPConnection.createInputStream | 0.86 3.02 0.11 < 0.19
com.sun.CORBA.iiop.IIOPConnection.processInput | 0.49 0.63 0.59 1.47
com.sun.CORBA.iiop.IIOPConnection.purge_calls | 0.36 0.31 < 0.11 < 0.19
com.sun.CORBA.iiop.ReaderThread.run | 0.12 0.47 0.11 0.58
java.lang.Thread.init | < 0.50 < 0.50 < 0.50 1.96

Table 5: Analysis of server overhead for the Reader Thread for IDL

Table 5 shows the Reader Thread overhead analysis. In the Reader Thread the majority of time is spent in reading the socket (method socketRead, ~56% to ~59%) and in creating new Worker Threads (~25% to ~49%).

### Method Name | % of time spent in the method
--- | ---
com.sun.CORBA.iiop.CDRInputStream.read_octet | N/A 26.08 22.09 16.35
com.sun.CORBA.iiop.CDRInputStream.alignAndCheck | N/A 26.08 20.93 10.72
com.sun.CORBA.iiop.CDROutputStream.write_octet | N/A 26.08 16.27 19.83
com.sun.CORBA.iiop.CDROutputStream.<init> | N/A 8.69 < 1.00 < 1.00
com.sun.CORBA.idl.TypeCodeImpl.copy | N/A 8.69 31.39 32.97
com.sun.CORBA.iiop.WorkerThread.run | N/A 4.34 1.16 0.26

Table 5: Analysis of server overhead for the Reader Thread for IDL
Table 6 shows the Worker Thread overhead analysis. Depending on the data size there are a few time consuming methods. The method read_octet reads the subsequent octet into the buffer. Before this happens it invokes the 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. The method write_octet writes the subsequent octet to the buffer. It uses the 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%).

On the IDL client side there are two important threads. The Java IDL Reader Thread spends the majority of time in the java.net.SocketInputStream.socketRead method which we have already described. The behavior is very similar to the RMI-IIOP client, which overhead is shown in Table 9. The client side overhead in the main thread is shown in Table 7. In the main thread the majority (over 90%) of the client side execution time is spent.

In Table 7 we can see that with the increasing data size the majority of time is spent in TypeCodeImpl.copy, write_octet, read_octet, alignAndReserve, alignAndCheck, and grow method. We have already described the functionality of these methods.
5.1.3 RMI-IIOP Beta

RMI-IIOP beta uses the CORBA infrastructure for implementing the distributed method invocations. As it builds on the same fundamentals as Java IDL there are several similarities\(^\text{12}\). RMI-IIOP uses the thread pool with the Listener, Reader, and Worker threads. It allocates 202 Worker Threads per client. Table 8 shows the server side overhead in the Reader Thread. The behavior of the Listener and Worker threads is similar as in IDL (described in previous subsection).

<table>
<thead>
<tr>
<th>Method Name</th>
<th>% of time spent in the method</th>
<th>Data size in elements:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>java.net.SocketInputStream.socketRead</td>
<td>49.56</td>
<td>43.93</td>
</tr>
<tr>
<td>java.lang.Thread.start</td>
<td>30.15</td>
<td>37.22</td>
</tr>
<tr>
<td>java.security.AccessControlContext.optimize</td>
<td>1.73</td>
<td>0.92</td>
</tr>
<tr>
<td>java.lang.Thread.init</td>
<td>1.30</td>
<td>2.31</td>
</tr>
<tr>
<td>java.lang.StringBuffer.toString</td>
<td>1.08</td>
<td>0.92</td>
</tr>
<tr>
<td>com.sun.rmi.iiop.IIOPInputStream.&lt;init&gt;</td>
<td>&lt; 1.00</td>
<td>4.39</td>
</tr>
</tbody>
</table>

Table 8: Analysis of server side overhead for the Reader Thread for RMI-IIOP beta

On the server side 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.

RMI-IIOP client creates two threads, the Reader Thread which handles the low-level communication and the main thread. The overhead analysis is shown in Table 9 and Table 10, respectively.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>% of time spent in the method</th>
<th>Data size in elements:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>java.net.SocketInputStream.socketRead</td>
<td>57.37</td>
<td>88.67</td>
</tr>
<tr>
<td>com.sun.rmi.iiop.IIOPConnection.createInputStream</td>
<td>31.96</td>
<td>1.88</td>
</tr>
<tr>
<td>com.sun.rmi.iiop.IIOPConnection.processInput</td>
<td>4.91</td>
<td>5.03</td>
</tr>
<tr>
<td>com.sun.rmi.util.Lock.unlock</td>
<td>4.09</td>
<td>3.77</td>
</tr>
<tr>
<td>java.util.Hashtable.&lt;init&gt;</td>
<td>1.63</td>
<td>&lt; 1.00</td>
</tr>
<tr>
<td>com.sun.rmi.iiop.ReaderThread.run</td>
<td>&lt; 1.00</td>
<td>&lt; 1.00</td>
</tr>
</tbody>
</table>

Table 9: Analysis of client overhead for the Reader Thread for RMI-IIOP beta

\(^{12}\) Although RMI-IIOP builds on CORBA it does not use the same classes as Java IDL does.
As already mentioned in the previous subsection the majority of the time in the Reader Thread is spent in the socketRead method. With the increasing data size the method processInput becomes time consuming too.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>% of time spent in the method</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.net.SocketOutputStream.socketWrite</td>
<td>13.77</td>
</tr>
<tr>
<td>java.lang.ClassLoader.defineClass0</td>
<td>6.14</td>
</tr>
<tr>
<td>java.lang.ClassLoader.findBootstrapClass</td>
<td>3.77</td>
</tr>
<tr>
<td>sun.misc.Resource.getBytes</td>
<td>3.33</td>
</tr>
<tr>
<td>com.sun.rmi.iiop.CDROutputStream.write_char</td>
<td>2.34</td>
</tr>
<tr>
<td>com.sun.rmi.iiop.CDROutputStream.write_repositoryId</td>
<td>2.81</td>
</tr>
<tr>
<td>java.lang.String.charAt</td>
<td>2.56</td>
</tr>
<tr>
<td>java.net.InetAddressImpl.lookupAllHostAddr</td>
<td>2.30</td>
</tr>
<tr>
<td>com.sun.rmi.iiop.CDROutputStream.alignAndReserve</td>
<td>2.04</td>
</tr>
<tr>
<td>java.lang.StringBuffer.append</td>
<td>1.92</td>
</tr>
<tr>
<td>com.sun.rmi.iiop.CDROutputStream.grow</td>
<td>&lt; 1.00</td>
</tr>
</tbody>
</table>

Table 10: Analysis of client overhead for the main thread for RMI-IIOP beta

Table 10 shows the client side overhead for the main thread. The results are comparable to the IDL client side main thread. With RMI-IIOP however more time is spend in writing to the sockets and in particular the output stream grow method has a larger share of up to ~26% of the total time.

5.2. Bottleneck Identification and Optimization

In the pervious section we have identified the sources of the overhead for RMI, RMI-IIOP, and IDL. We have seen that IDL and RMI-IIOP use a similar multithreading model which 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 and IDL use 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). IDL and 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. IDL and RMI-IIOP use 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).

An important source of the sender side overhead is inefficient multiplexing, easily identified by IDL and RMI-IIOP. 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 would 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 three 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 IDL and 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 IDL and 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 IDL (and 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.

All three 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 [23]. 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 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 and IDL 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, and FCS2b.

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. The data from build post-beta FCS2b shown here is the most recent available at the time of writing. It is expected at the time of writing that final released version will be based on post-beta FCS2b. (Note to reviewers - the final version will be built in the next few days - it would be very helpful to be able to update this with data from the released code, given that this article, if accepted, will not appear until many months after the RMI-IIOP release).

### 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 (Figure 38). This is most evident in moving from the beta release to build 7B, with a further small improvement in build 8C and FCS2B in most cases. RMI-IIOP post-beta 8C performs around 8% and FCS2B around 11% better than RMI for all basic data types except string. String is still 20% and 14% slower, respectively. Figure 39 shows the geometric means of basic data types and RTTs for compound types testStruct and object reference. The compound data types have shown variations. For testStruct, build 7B gave a large improvement, with a minimal further setback in 8C and improvements in FCS2B to give performance within 30% of RMI. For myObject, build 7B clearly regressed performance by around 4 fold from the beta, but this was recovered in build 8C and FCS2B to give RMI-IIOP times that are over 40% faster than RMI.
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 testStruct and myObject showed poor performance. Figure 40 shows performance improvements for handling one element arrays of basic data types. There is a large improvement from beta to version 7B. Again, version 8C shows a little drawback and FCS2B a another small improvement. The performances compared to RMI-IIOP beta are improved for more than 30%. In Figure 41 the average improvements for basic data types and for the compound types (testStruct and myObject) are shown. The greatest improvements have been achieved by the testStruct, where the version 8C is only 12% and FCS2B only 10% slower than RMI. The performance of myObject is improved too, but not as much as the testStruct.
To present the impacts of data size Figure 42, Figure 43, and Figure 44 present the throughput for different array sizes, from 1 to 16384 elements. Figure 42 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 finally achieved better performance than RMI. Figure 43 shows the throughput for testStruct, where versions 7B, 8C, and FCS2B continuously improved performance. This time however the RMI performance is not achieved. We have already discussed the problems with the testStruct, which has to be transferred as CORBA object-by-value, in previous
sections. Figure 44 presents the throughput for arrays of object references. We have already identified a performance drawback by the 7B version, but the 8C and FCS2B manage to offer slightly better performance than RMI.

Figure 42  Comparison of average throughput for basic data types in dependence to array size

Figure 43  Comparison of average throughput for the testStruct in dependence to array size
Some further comparisons of RMI to post-beta RMI-IIOP have shown that RMI-IIOP gives relatively slightly better performance than RMI on a single machine than when run across a network. This is believed to be due to better multithreading exploitation in RMI-IIOP. Consequently, the RMI-IIOP results presented here for single machine tests should be taken as 'best-case'.

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 the most important distributed object models for the Java has been done: RMI, IDL, and the newly developed RMI-IIOP beta, which enables the use of RMI over the IIOP protocol and makes it CORBA compliant. In the performance comparison RMI provided the best results in all tests\(^\text{13}\), followed by RMI-IIOP beta and IDL, which alternated the places depending on the testing scenarios. 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 ~38Mbps, which is more than a third of the theoretical network throughput. RMI-IIOP beta and IDL achieved

\[^{13}\text{But note the significant improvements in later versions of RMI-IIOP gave equivalent or better performance than RMI in most cases.}\]
~19 Mbps and ~12 Mbps, respectively. In multi-client scenarios IDL and RMI-IIOP beta were from ~75% to ~100% slower than RMI.

The overhead analysis showed that the bottlenecks fall in the following categories: (1) the overhead of the low-level methods that take care for 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. The presented results for the RMI-IIOP post-beta FCS2b show performance that is always comparable and in many times even superior to the RMI. Until the final version additional small optimizations will be applied. 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) for helpful suggestions, and to Ales Zivkovic (University of Maribor) and Tomaz Domajnko (University of Maribor) for helping by the measurements.

References

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 all 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: matjaz.juric@uni-mb.si) Mr. Rozman is a...
university professor at the University of Maribor. His interest include software design methods, software process improvement and quality assurance.

**Marjan Hericko**, 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. Hericko is an assistant professor at the University of Maribor. His interest include software object oriented design methods, formal specifications and object oriented metrics.