Java 2 RMI and IDL Comparison
Matjaz B. Juric, Ivan Rozman

1. Introduction

When developing applications using distributed objects, Java developers have to make the choice which distributed object model to use. They can choose among the Remote Method Invocation (RMI) bounded with Java from version 1.1, or, they can use one of the several CORBA implementations. In version 2 Java has been supplemented with a native CORBA implementation known as Java IDL. Lately, even the most enthusiastic users of RMI have started to question which model to use and why Sun has decided to support both, CORBA and RMI.

In this article we will present advantages and disadvantages for both models based on a set of selection criteria: features, performance, learning curve, ease of development and support for legacy systems. We will place special emphasis on performance and show how RMI and IDL behave under some relevant usage scenarios.

2. Java RMI

Java Remote Method Invocation (RMI) supports basic distributed object functionality. Therefore it can be seen as a fully functional object request broker (ORB). It has been developed for the Java platform and is implemented mostly in Java. There are three important layers:

- The stub/skeleton layer connects the application objects and the rest of the RMI system. A stub on the client side forwards the requests to the server side remote object. A skeleton dispatches the remote calls to the actual object implementation. In Java 2 (version 1.2) generic code is provided for skeletons. Therefore only stubs have to be generated for remote objects.
- The remote reference layer is responsible for the semantics of the communication. It connects the stubs and skeletons with the low-level transport layer.
- The transport layer is responsible for the communication management and for dispatching the messages.

For the wire communication RMI uses the Java Remote Method Protocol (JRMP). JRMP makes use of two other protocols: Java Object Serialization which is used for call marshaling and returning results and HTTP which is used to send remote method invocation data and obtain results. Usually the RMI transport layer opens direct sockets to hosts. To bypass firewalls two alternative HTTP based mechanisms are also available. Both send RMI data encapsulated into a HTTP POST request. According to the documentation the HTTP connections are at least an order of magnitude slower than those sent through direct sockets, therefore the latter have been used in the

\[1\] To avoid confusion please keep in mind that Java IDL is a complete CORBA implementation.
performance comparison. RMI transport protocol defines six messages: Call, Ping, DgcAck, ReturnData, HttpReturn and PingAck.

3. Java IDL

Java IDL is an implementation of the CORBA specification. It provides the functionality of the ORB and the Naming Service. Like RMI it is implemented in Java, except for some low level communication operations. Java IDL has the following components:

- Static stubs and skeletons are used for static invocation interface and are responsible for marshalling and demarshalling the requests, respectively. Static stubs and skeletons are generated at compile time by an IDL compiler.
- Dynamic invocation interface supports 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 (servers) that have no static knowledge about the interface they are implementing.
- 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.
- Interface Repository maintains the information about the interfaces, operations and their syntax. Usually it is used for dynamic method invocation.

For the communication, CORBA standard specifies the General Inter-ORB Protocol (GIOP) which 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 eight message formats: Request, Reply, CancelRequest, LocateRequest, LocateReply, CloseConnection, MessageError and Fragment (added in version 1.1). With these messages all the functionality of CORBA is supported. 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.

Both models use the low-level communication services provided by the operating system. To enable the transparent remote method invocation several steps are required like marshalling the client request, locating the appropriate target object, transferring the request, demultiplexing, demarshalling and dispatching the request, performing an operation upcall, and returning the result. In the process of request serialization several transformations take place. Supposing constant network bandwidth and the use of the same low-level operating systems services by both models, the mentioned steps are crucial for performance.

4. Functional Comparison
RMI and CORBA support similar distributed object functionality, because they are based on similar concepts. They have however taken different paths to provide support for multiple platforms and operating systems. Java has defined its own execution environment, provided a Virtual Machine and platform independent byte code. Therefore it can execute everywhere a JVM (Java Virtual Machine) is implemented. CORBA on the other hand has been designed as a “glue” for applications written in different programming languages on different operating systems and platforms. CORBA can be seen as an architecture for integrating diverse applications.

Figure 1 shows the steps necessary to build a distributed application in IDL and Figure 2 shows the steps for RMI. The main characteristics of distributed objects is their strict separation of interface and implementation. In CORBA the object interfaces are specified in a special language called Interface Definition Language (IDL). IDL is a declaration language only and provides constructs for defining interfaces, models and special data types. It defines the CORBA compliant data types as well. IDL represents a “common denominator” of all programming languages and architectures supported by CORBA. CORBA also defines mappings from IDL to all supported programming languages which includes Java. These mappings define how different IDL constructs like modules, interfaces, basic and compound data types etc. map to the destination programming language constructs. In RMI the interfaces are defined in Java using the interface construct. All RMI interfaces have to directly or indirectly extend the java.rmi.Remote interface. Also each method has to include the exception java.rmi.RemoteException in its throw clause. In Listing 1 a simple IDL and RMI distributed object interface declaration is shown.

```java
// IDL
interface charTestServer {
    void acceptType(in char Value);
    char returnType();
};

// Java RMI
public interface charTestServer extends java.rmi.Remote {
    void acceptType(char Value) throws java.rmi.RemoteException;
    char returnType() throws java.rmi.RemoteException;
}
```

Listing 1: Distributed object interface declaration in IDL and RMI
When the interfaces are defined their functionality has to be implemented. Prior to this in CORBA the IDL definitions have to be mapped to the appropriate Java equivalents - stubs and skeletons have to be generated using the IDL to Java compiler.
Interfaces defined in IDL can use some constructs that are unavailable in RMI and vice versa – RMI also provides some unique constructs. Following is a list of the most important differences.

In IDL a method can be declared as `oneway`, which makes a method invocation unidirectional without returning a result or success acknowledgment to the sender. One-way operations can be useful in cases where it is not important whether the receiver actually received the message. The advantage for the sender is that it is not blocked until the competition of the operation. In RMI there is no direct way to achieve the same functionality.

IDL provides parameter passing modes `out` and `inout` to return results through parameters. This actually represents passing parameters by reference. Java (and RMI) does not support this notation, therefore in IDL to Java mapping special Holder classes are generated by the IDL compiler. They have an attribute of the corresponding type and methods for setting and reading the attribute (and some utility methods which are not important in this context).

When in IDL an object (interface) is passed as a parameter or method return value this means that a reference is transmitted, but the object which implements the interface does not change its location. In RMI on the other hand an object can be passed by reference or by value. If an interface extends the `java.rmi.Remote` then it is passed by reference. Otherwise it is transmitted by value: the state of the object is sent to a remote location where a new instance is created and the state is recovered. For the latter to work the object has to provide serialization support.

The problem with sending objects by value is that it is not necessary that the receiver side has the corresponding implementation classes. RMI solves the problem by providing dynamic class and stub downloading in the case when the receiver does not have the code locally installed. Neither passing objects by value nor dynamic code downloading is supported by IDL (although specifications for similar functionality for CORBA are on the way – Objects By Value).

CORBA supports a special type `any` which can hold any other simple or compound data type. The functional equivalent in Java is `java.lang.Object` which has similar characteristics.

After the implementation of the interfaces the client and the server applications can be developed. The client application uses the services (methods) provided by distributed objects. The server application creates initial instances of distributed objects, makes them remotely available and starts the request processing loop. The source code can now be compiled with the Java compiler (`javac`). In RMI the stubs and possibly skeletons are generated with the `rmic` compiler.

In the implementation phase we are faced with further differences. The most obvious is the difference in obtaining object references. RMI supports the URL based naming and a reference to a remote object is obtained with a lookup that accepts an URL address and a logical object name. If the remote objects are moved to a different computer all client references have to be updated. Therefore it is wise to code the URL as constants that can be easy modified.
Java IDL (CORBA) on the other hand supports the Naming Service, one of the standardized CORBA services. The Naming Service provides a more powerful mapping between object references (IORs – Interoperable Object References) and names. It allows the definition of naming contexts and name bindings. Because the objects are referenced by their logical names only, they are location independent. In Java IDL it is only necessary to specify the initial host address of the ORB and even that can be done through command line arguments. It is however worthy of praise that Java IDL strictly uses the Naming Service and does not provide non-standardized methods like `bind` (Orbix, Visibroker) that allow simpler binding to remote objects without messing with the Naming Service. They do however prevent the interoperability between different CORBA implementations.

The notion of naming contexts is also more suitable for large, enterprise-scale applications. Because the object references can be persistent, the clients do not have to make name re-bindings if the server object is restarted. In IDL the Naming Service is started with the `tnameserv` command. It is interesting that although the RMIs naming scheme is simpler, it still needs the initial bootstrap naming object which is why the RMI Registry has to be started prior the naming lookup will function.

We have already mentioned that the CORBA specifies a dynamic method invocation interface, which permits clients to build requests in runtime and a dynamic skeleton interface, which permits an application to dynamically implement server side functionality. This is useful for several types of applications. The application can also gain information about the methods and attributes of an interface and about the method signatures. In CORBA this is achieved through the Interface Repository. RMI on the other hand provides support for introspection and dynamic method invocation through Java Core Reflection API. However, through dynamic invocation interface CORBA provides deferred synchronous method invocations. This means that a method invocation can be initiated without blocking the client. The client can later pool the results from the server.

CORBA standard specifies plenty of functionality. The most evident functionality that is missing in the Java IDL is the Implementation Repository. As a matter effect the object implementations (server applications) have to be started manually. There is however no obstacle for using another CORBA ORB on the server side if the functionality of the Implementation Repository is needed. RMI in its latest version provides a mechanism for automatic server object execution through the `Activatable` class and the RMI daemon (`rmid`). However, using this concept in RMI requires minor changes in the implementation code.

If we take a closer look at the JRMP, the transfer protocol used by RMI, we will see a `DgcAck` message. This message has an important role in the distributed garbage collection that is supported by RMI. Distributed garbage collection is the extension of the already known Java garbage collection. Java IDL does not support this functionality.

We have already identified the capability of CORBA to support multiple programming languages, operating systems and platforms. It is also widely known that different CORBA implementations are interoperable. Therefore they all have to
support the Internet Inter-ORB Protocol (IIOP), a specialization of the General Inter-ORB Protocol (GIOP). Because GIOP/IIOP is a language independent protocol it is suitable for company’s backbone. The protocol used by RMI, the JRMP is limited to Java.

The interoperability between different CORBA implementations is achieved only when all implementations support the protocol correctly and when no non-standard extensions are used (most noticeable the `bind` method instead of the Naming Services). The verification of different products with the specifications remains the Achilles’ heel of the CORBA standard. A major drawback of the Java IDL is its missing support for the IDL data types `wchar` and `wstring`. This is unacceptable particularly for international applications which rely on Unicode characters and also presents a problem for interoperability (including the interoperability with the newly developed RMI-IIOP).

If we compare the ease of learning and ease of development it is obvious that RMI is simpler to use and more elegantly integrated into Java. It is not only the fact that in RMI everything can be done in Java, but also some functionality that is currently missing in Java IDL. CORBA is not as easy to learn and use because it is a more complex architecture. Because of its language independence it introduces several abstractions and limitations that have to be taken into account. Although the IDL is relatively easy to learn the developers have to understand the mapping from IDL to Java, too.

On the other hand CORBA (Java IDL) enables the interoperability between Java applications and all current applications written in different languages. Achieving this interoperability with CORBA is straightforward. CORBA also enables easy access of Java applications to legacy systems. Legacy systems can be integrated in the distributed object environment using the object wrappers. In this area CORBA (Java IDL) is much stronger than RMI.

### Features, supported by CORBA only

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for different programming languages through the IDL language mappings</td>
<td></td>
</tr>
<tr>
<td>Language independent wire protocol (GIOP/IIOP)</td>
<td></td>
</tr>
<tr>
<td>Central Interface Repository and dynamic acquiring of object interfaces via Interface Repository</td>
<td></td>
</tr>
<tr>
<td>Dynamic server implementations (Dynamic Skeleton Interface)</td>
<td></td>
</tr>
<tr>
<td>Parameter passing modes (out, inout)</td>
<td></td>
</tr>
<tr>
<td>Persistent object references</td>
<td></td>
</tr>
<tr>
<td>Location independent object naming</td>
<td></td>
</tr>
<tr>
<td>Delayed synchronous method invocation</td>
<td></td>
</tr>
<tr>
<td>One-way method invocation</td>
<td></td>
</tr>
<tr>
<td>Different server activation modes</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Features, supported by CORBA only
<table>
<thead>
<tr>
<th>Features, supported by RMI only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for different operating systems and platforms through the Java Virtual Machine</td>
</tr>
<tr>
<td>Object passing by value</td>
</tr>
<tr>
<td>Dynamic class downloading</td>
</tr>
<tr>
<td>Dynamic stub downloading</td>
</tr>
<tr>
<td>URL based object naming</td>
</tr>
<tr>
<td>Automatic distributed garbage collection</td>
</tr>
</tbody>
</table>

Table 2: Features, supported by RMI only

CORBA is also an older and a more mature technology than RMI. The first version of CORBA has been released in 1992. RMI has been first introduced with JDK 1.1. However, Java IDL (as the implementation) is even younger than RMI. Therefore you have to be cautious when evaluating the maturity of a technology.

We have seen that Java IDL and RMI have advantages and disadvantages. They are gathered in Table 1 and Table 2. While RMI is more suitable for Java-only applications and is easier to learn and to develop, the Java IDL has benefits comparable to other CORBA implementations. It is based on an open architecture, offers easy connectivity and interoperability with different languages, platforms and operating systems. However it is also more complex and not as easy to use.

5. RMI + IDL = ?

When observing the advantages and disadvantages of RMI and CORBA it is reasonable to think about using both models and to turn to advantage the benefits of both. Technically it is possible to build objects that provide a bridge or a half-bridge between the RMI (JRMP) and Java IDL (IIOP). If these objects have the functionality that is known in advance such development is not very difficult.

However, developers have another choice. For a long time there have been rumors about an implementation of the RMI over the IIOP protocol. This will allow RMI objects to cooperate with CORBA objects and vice versa. Since June this year dream came true. The developers can download the final version of the RMI-IIOP product, that has been jointly developed by Sun and IBM. With minimal changes to code it allows RMI objects to work over the IIOP protocol. To support the whole functionality it is based on some very new standards that will be released in the new version of CORBA standard (Objects-by-Value and Java to IDL Mapping). We will present the RMI-IIOP in an article in the next issue. Because we have been working tightly with the IBM Java Technology Centre in Hursly, UK (the place where RMI-IIOP has been developed) on the performance evaluation and optimization we will also present performance results of RMI-IIOP that are very promising.
6. Performance Comparison

6.1. Testing Method

We have compared the Java RMI and IDL and have pointed to the most important differences. We have however promised to present a performance comparison as well. To measure the performance of the Java IDL and RMI we have used the Performance Evaluation Model for Distributed Objects, developed at the University of Maribor. The model enables performance evaluation of different object models while ensuring the comparability of the results. Several factors (criteria) are evaluated, but the most important are the latency of the method invocations or the round trip time (RTT), the throughput and the performance degradation in multi-client scenarios. To get a clear picture how a distributed object model performs it is important to understand:

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

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

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

The comparability of the results between models is achieved with the identical implementations that differ only in necessary details regarding obtaining the initial references. Further, there is a consistent mapping between Java and CORBA data types – with one exception. We have already identified the unsupported wchar and wstring in Java IDL. Therefore we have used the mapping to char and string, respectively, although the latter (IDL) data types are only 8-bits long when transferred over the wire compared to the 16-bits Unicode characters in Java RMI. The measurements have been accomplished on identical equipment in an identical environment.

To measure the performance factors, interfaces (like those presented in Listing 1) with corresponding classes have been defined on the server side for all basic data types, and for two compound data types:

- a testStruct – a structure in IDL / a class transferred by value in RMI (Listing 2) and
- an object reference - myObject.

To measure the impact of the data size on the latency and throughput, interfaces for all basic and compound data types that deal with arrays (RMI) and sequences (IDL) have been defined. To evaluate the multi-client scenarios, special synchronization interfaces have been defined that take care for clients. In multi-client scenarios the tested number of clients simultaneously invoked the same method without delays. Therefore the numbers presented here correspond to a much larger number of real-world clients (the actual number depends of the usage pattern). The synchronization objects have been designed so that they do not influence the measurements.
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.

All test have been repeated 20 times and each measurement has been iterated 200 times to get the necessary accuracy of the results. A standard deviation has been calculated for each measurement to prevent mistakes caused by external factors.

For all the performance measurements the Sun Java 2 SDK, Standard Edition, version 1.2 has been used. For RMI the 1.2 compliant stubs and skeletons and direct sockets to hosts have been used. 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. All the computers used Microsoft Windows NT 4.0 Workstation with Service Pack 3 as their operating system. The actual performance measurements have been done for up to eight simultaneous clients. Therefore ten identical Pentium II-333 MHz computers with 128 MB RAM have been used. Eight of them were used for client applications, one of them was the server and one was used for running the synchronization objects. For the single client scenarios the results are reported for client and server objects executing on two separate computers. The computers were connected into a 100 Mbps Ethernet network that was free of other traffic.

// RMI
class testStruct implements java.io.Serializable {
  boolean b;
  byte o;
  short s;
  int l;
  long ll;
  float f;
  double d;

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

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

// IDL
class testStruct {
  boolean b;
  octet o;
  short s;
  long l;
  long long ll;
  float f;
  double d;
};
Listing 2: The compound data type testStruct

6.2. Results

In single client scenario it is interesting to observe the method invocation performance for methods that accept and return primitive data types. We have observed that different primitive data types do not show any substantial differences in method invocation time, therefore we have decided to report only the averages. It is obvious that the remote method invocation overhead is larger than the over-the-wire transmission overhead. Therefore Figure 3 shows the round trip times for geometric average of basic data types (*boolean*, *char*, *byte/octet*, *short*, *int/long*, *long/long*, *float* and *double*), for *string*, *testStruct* and *myObject*.

![Figure 3: Round trip time comparison for primitive data types between RMI and IDL](image)

When using primitive data as parameters or method return values RMI shows great performance advantage over Java IDL. By basic data types RMI is 80% faster and by strings more than 65%, although the RMI used the Unicode characters and IDL the 8 bit char-set. RMI is faster by the compound data type *testStruct* and by the object reference *myObject*, too. It is 45% and 9% faster than IDL, respectively. We can see that only by handling of the object reference the IDL achieved times that are comparable to RMI.

To investigate the influence of the data size on the performance we have repeated the measurements with different sizes of arrays and sequences, from 1 element to 16384 elements. With larger data sizes the differences in data type sizes were easily identifiable. The *byte/octet* data type which is 8 bits long when transferred over the wire was the fastest, followed by *boolean* (16 bits) and so on up to the data type *double* which is 64 bits long. Both compound data types showed different behavior and the invocation times (RTTs) raised for them faster as for basic data types. Therefore in Figure 4 we present the average method invocation time for basic data
types and for string. In Figure 5 we present the method invocation times for testStruct and myObject.

Figure 4: Different data sizes for RMI and IDL – basic data types and string

For all data types we can observe almost linear dependence between data size and round trip time (method invocation time). In Figure 4 we can see that Java IDL performed poor for all basic data types. When the data size raised, the times increased and topped by 75 ms for transferring an 16K array. RMI transferred the same array in 12 ms – which means it is more than 6 times faster. When comparing strings, IDL and RMI achieved comparable results – on the first sight. As already mentioned the RMI had to deal with Unicode characters (16 bits long) while IDL used the standard 8 bit characters (wstring is not supported by Java IDL).

Figure 5: Different data sizes for RMI and IDL – testStruct and object reference myObject

The times for testStruct and myObject are reported up to 1024 elements and 256 elements arrays/sequences, respectively. This is because of the large increase in
RTTs. We can see that the testStruct is handled similarly well by RMI and IDL. IDL showed a deficit by smaller sequences (64, 256 elements) but was able to achieve times for 1024 elements that are practically equal to RMI times – around 60 ms for 1024 elements of testStruct. It is however important to understand, how testStruct is handled by IDL and how by RMI. For IDL it is a simple data structure and only data attributes have to be transferred. For RMI on the other hand the testStruct is a class that is transferred by value. This means that the object is serialized (though serialization methods provided), the stream is transferred to the remote location where a new instance of the testStruct class is created and the state is restored. By object references RMI again showed better performance and was more than 2 times faster than IDL.

In many occasions the distributed objects provide services that are used by several clients simultaneously. To be able to understand the performance levels offered by RMI and IDL in multi-client scenario we have run several tests where we have gathered the results for two to eight simultaneous clients that invoked the methods without delays. We have found out that different basic data types did not have any relevant impact on the performance. Only the testStruct and the object reference showed a little different behavior. However, to simplify the results, we have decided to present the geometrical mean of basic and compound data types. In Figure 6 we can see the method invocation times depending on the number of simultaneous clients and in Figure 7 the performance degradation factor.

![Figure 6: Average method invocation times for multi client scenarios](image)

In absolute times the RMI preserved its advantage over Java IDL. By RMI we can observe a linear increase in method invocation times when the number of client increases. By IDL the times rise faster for first four clients and than again slower. For example, by four simultaneous clients RMI is 2 times and by eight clients 30% faster than IDL.
Figure 7: Performance degradation comparison for multi client scenarios

When observing the performance degradation factor we can see that IDL is more comparable to RMI. By eight clients, IDL’s degradation is even lower than RMI’s, although RMI is better for every smaller number of clients. By four clients we can expect a performance degradation from three to four and by eight clients from five to six. However keep in mind that in these tests the clients repeated the method invocations without any delays, therefore the presented number of “synthetic” clients is equivalent to a much larger number of real world clients.

7. Conclusion

In this article we have compared Java RMI and Java IDL, two distributed object models included in Java 2. Although the basic functionality and the basic concepts of both models are similar, there are several differences in details. Each of them has its advantages and disadvantages. In general, we can conclude that RMI is more suitable for applications where no interoperability with objects written in other languages is needed. Java IDL (CORBA) on the other hand is better for heterogeneous environments. However, it is a more complex architecture and requires more learning and development effort.

The complexity of CORBA is expressed in performance, too. In the performance analysis we have seen that RMI outperformed Java IDL in almost every case. The reason for slower performance of Java IDL is the greater complexity of CORBA architecture and worse integration with Java. Java IDL, however, showed some considerable weaknesses, especially in handling large data sizes. We have done some additional research and found out that they are caused mostly by an ineffective buffer allocation for message marshalling, which does not implement message fragmentation (GIOP 1.1). It has to resize the buffer too many times. The other reason is a different (less efficient) threading policy. Together with some unimplemented and missing
functionality the Java IDL stays the second choice for distributed objects in Java if the developer can live without interoperability to other languages\textsuperscript{2}.

The position of RMI as the performance leader and the better distributed object model for Java is already endangered by the RMI-IIOP, the implementation of RMI over IIOP protocol. RMI-IIOP promises to gather the best of both worlds while preserving the RMI’s ease of use and CORBA’s interoperability. In a follow-up article we will present what the RMI-IIOP really has to offer and how good its performance is.

Matjaz B. Juric is a researcher at the University of Maribor, Ivan Rozman is a professor at the University of Maribor. Author can be reached by e-mail at matjaz.juric@uni-mb.si.

\textsuperscript{2} Please be aware that this conclusion is valid for Java IDL only and not for all CORBA implementations, like those provided by companies Iona, Inprise, BEA, ect.