Performance Comparison of CORBA and RMI

Matjaz B. Juric, Ivan Rozman, Marjan Hericko
Institute of Informatics, Faculty of Electrical Engineering,
Computer and Information Science, University of Maribor
Smetanova 17, SI-2000 Maribor, Slovenia, Europe
Telephone: +386 2 2355113, fax: +386 2 2355134
E-mail: matjaz.juric@uni-mb.si

Abstract

Distributed object architectures and Java are important for building modern, scalable, web-enabled applications. This paper is focused on qualitative and quantitative comparison of two distributed object models for use with Java: CORBA and RMI. We compare both models in terms of features, ease of development and performance. We present performance results based on real world scenarios that include single client and multi-client configurations, different data types and data sizes. We evaluate multithreading strategies and analyse code in order to identify the most time consuming methods. We compare the results and give hints and conclusions. We have found that because of its complexity CORBA is slightly slower than RMI in simple scenarios. On the other hand CORBA handles multiple simultaneous clients and larger data amounts better and suffers from far lower performance degradation under heavy client load. The article presents a solid basis for making a decision about the underlying distributed object model.

Keywords: Java, CORBA, RMI, performance analysis

1. Introduction

Exponential network growth, global connectivity and new application domains have reached the limits of traditional object oriented programming environments. The most critical aspects not supported by them are connectivity and interoperability. These two aspects are crucial for building a new generation of distributed information systems. Distributed systems require communication between computing entities.

The most promising language for building distributed portable applications is Java. Java offers basic low-level communication mechanisms with its support for sockets. For the new generation of information systems high level development approaches are needed. The alternative is a concept, known from procedural languages as remote procedure call (RPC). Pairing of RPC concepts and object paradigm results in a distributed object model. For Java there are two suitable distributed object models, *Common Object Request Broker Architecture* (CORBA) and *Remote Method Invocation* (RMI). Both CORBA and RMI hide the communications details of remote method invocations thus their basic functionality is similar.

For a software developer, several aspects are important [1] in order to make the right choice concerning which distributed object model to use. These include features, performances and scalability, maturity, support for legacy systems and ease of development. Traditionally one of the most important criteria are performances and scalability. As far as we know there are no in-depth analyses of distributed object models and Java language. In this paper a detailed analysis of CORBA/Java and Java RMI can be found. Focus is given to performance evaluation. Several testing scenarios are defined so that the results between RMI and CORBA/Java are comparable. Different data types and single-client and multi-client configurations are defined and the results are presented and analysed. With the use of a profiler tool the most time consuming parts of the code are identified. With the presented in-depth analysis of CORBA/Java and Java RMI the paper contributes to the understanding of advantages and disadvantages of both models.

The paper is divided into nine sections: in sections 2 and 3 a brief outline of CORBA and Java RMI can be found. Section 4 outlines the differences between them. Section 5 describes the comparison method and the configuration used for testing. Sections 6 and 7 present the results of CORBA and RMI performance tests, respectively. Section 8 presents a detailed performance comparison and interprets the results and section 9 gives a conclusion.

2. Overview of the CORBA architecture

Object Management Group's CORBA is the most important middleware project ever undertaken by industry [2]. It is based on object management architecture (OMA) and Core Object Model (COM) [3].

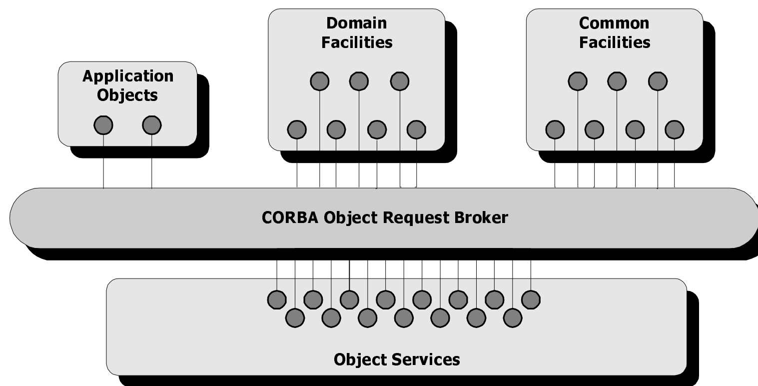

Figure 1. The overview of the CORBA platform

In Figure 1 the five main parts of the architecture [4] are presented:

- *Object request broker* (ORB) is the integral component of the architecture. It hides all the details of the communication between the two objects – the client and the server object. ORB is the object bus.
- *Object services* (CORBAservices) define the system level object frameworks that widen the range of basic functions of the ORB with services such as naming, event, life-cycle, transaction, relation, etc.
- *Common facilities* (CORBAfacilities) 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.

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

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

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

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

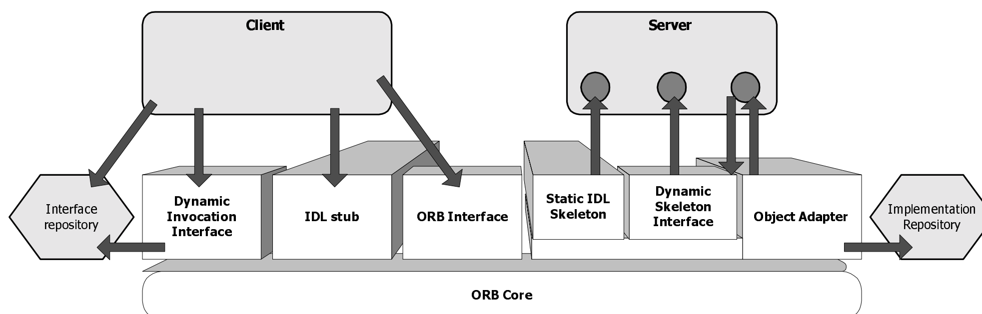

Figure 2. The structure of the object request broker

The client and the server are two CORBA objects which communicate by means of method invocation [5]. The client invokes a method on the remote server with a simple method invocation, i.e., `object.method(args)`. The server returns the result as a return value or through arguments. CORBA ORB supports two types of method invocation:

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

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

3. Overview of the Java RMI

Although Java can be used with the industry standard distributed object architecture CORBA, a native distributed model called Remote Method Invocation (RMI) was added to Java in the version 1.1. It offers a basic functionality of an object request broker and shares the basic concepts with CORBA. Because it was designed for Java it offers some services not found in CORBA.

Similar to CORBA, RMI utilises strict separation of the interfaces from the implementation. Therefore a construct interface has been introduced. Because RMI is bound to Java, interfaces are specified in the Java language. Figure 3 shows the three independent layers that constitute the RMI system. These three layers are [6]:

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

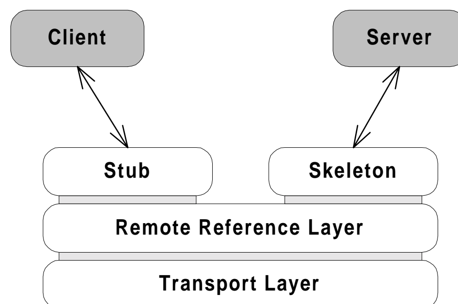

Figure 3. The Java RMI system architecture

The RMI implements a garbage collection algorithm with reference counting similar to Modula-3's Network Objects [7]. With dynamic class loading, the classes required to handle method invocations can be loaded at runtime. For the communication RMI uses a protocol which is not compatible with CORBA's IIOP. It uses:

- Java Object Serialisation for call marshalling and returning data and
- direct socket connections or HTTP version 1.0 or higher (when direct socket connections can not be established, like firewalls configurations etc.) for invoking remote method invocations and obtaining the return data.

4. CORBA versus RMI

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

- Features.
- Performances and scalability.
- Maturity.
- Support for legacy systems.

- Learning curve and ease of development.

Our hypotheses were as follows:

H1: Java RMI is suitable for small scale fully web-enabled applications where legacy support can be managed by custom build or pre-build bridges, where ease of learning and ease of use are more critical than performances are.

H2: CORBA/Java is suitable for large scale fully or partially web-enabled applications where legacy support is needed and good performances under heavy client load are required.

4.1. Feature comparison

It has already been stated that CORBA is much more than just an object request broker. It is a complete distributed object platform with support for different programming languages and several services and facilities not found in RMI [8, 9]. Therefore from now on we will compare RMI only to the CORBA object request broker (ORB).

Table 1 presents the crucial features supported only by CORBA ORB. The most important are language independent wire protocol, dynamic acquiring of object interfaces and the ability to compose and execute method invocations at the run-time. They are supported by GIOP/IIOP, interface repository and dynamic invocation interface, respectively. Other features include dynamic server implementations, different parameter passing modes, persistent naming, persistent object references, one-way operations, implementation repository and different server activation modes.

Feature
Language independent wire-protocol (GIOP/IIOP)
Dynamic acquiring of object interfaces via Interface Repository
Dynamic method invocations (Dynamic Invocation Interface)
Dynamic server implementations (Dynamic Skeleton Interface)
Parameter passing modes (in, out, inout)
Persistent naming
Persistent object references
One-way operations
Shared, unshared and per-method activation modes
Implementation repository

Table 1: Features found only in CORBA

In Table 2 the features supported only by RMI are listed. Because RMI was designed for Java from the beginning, it supports some features not found in CORBA/Java combination. These include dynamic class and stub downloads, object passing by value, URL based object naming and automatic garbage collection.

Feature
Dynamic class downloading
Dynamic stub downloading
Object passing by value
URL based object naming
Automatic garbage collection

Table 2: Features found only in RMI

4.2. Maturity

CORBA is an older and a far more mature technology than Java RMI. The first version of CORBA was published in 1992, the current version is 2.2 and version 3.0 is expected. CORBA is widely used by leading companies and is established as a mature, secure and scalable architecture. RMI on the other hand is not even two years old (when writing this article). Until now it has not been widely used in mission critical applications.

With support for IIOP and several programming languages, CORBA is suitable for building a company's backbone. RMI on the other hand uses its own wire protocol and supports only Java programming language.

CORBA supports shared, unshared and per-method activation modes. It supports implementation repository and smart proxies for effective load balancing and fault tolerance. RMI in its first version lacks support for such advanced features.

4.3. Support for legacy systems

A common way of reusing legacy applications in a distributed object architecture is through the use of object wrappers. A legacy system is any system that, regardless of age or architecture, has existing code and is still useful and in use today. In the context of distributed object architecture, this definition includes not only

traditional mainframe-based systems but also systems written in C, C++ or other languages, PC-based and client/server systems and personal productivity tools. Therefore the support for legacy applications can be crucial.

Object wrappers provide access to legacy systems through an encapsulation layer. Once wrapped, legacy systems can participate in distributed object environments using object request brokers. For effective wrapper building, support for different programming languages and different platforms is needed. The most welcome is the ORB's native support for the programming language of a legacy system. RMI supports multiple platforms (in fact it is platform independent) but connecting legacy code to the Java language can be painful. CORBA on the other hand supports multiple platforms and several programming languages and has an edge over Java RMI.

4.4. Learning curve and ease of development

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

5. Performance comparison

Both CORBA and RMI introduce a certain level of overhead into the method invocations [10]. Therefore one of the goals of the performance comparison was to measure this overhead. The other goal was to investigate the performance degradation under heavy client load. Several scenarios were developed for communication between client and server objects. The requirements for the tests were as follows:

(1) The results from Java RMI and CORBA/Java tests should be comparable.

(2) Single and multiple client scenarios should be simulated.

(3) Different data types should be used.

(4) Hardware equipment found in typical user environment should be used.

5.1. Testing method

We simulated typical interactions between client and server objects found in common three-tier applications [11]. Therefore we defined a set of interfaces with methods for Java RMI and CORBA IDL shown in Listing 1 and Listing 2, respectively. An Automatic Teller Machine application has been used as a basis for the tests.

```
public interface Atm extends java.rmi.Remote {
…  public boolean Working() throws java.rmi.RemoteException;
   public long getAtmNo() throws java.rmi.RemoteException;
…
}
public interface Account extends java.rmi.Remote {
…  public float getBalance() throws java.rmi.RemoteException;
   public java.lang.String getType()
                        throws java.rmi.RemoteException;
   public double getLimit()throws java.rmi.RemoteException;
…
}
public interface Card extends java.rmi.Remote {
…  public int getNumber() throws java.rmi.RemoteException;
…
}
```

Listing 1: Java RMI interfaces used for performance testing

The interfaces were implemented in Java. Attention has been paid to insure that implementations for CORBA and RMI were equal. The unavoidable differences were only in the initial invocations to the ORB and a few other details. Therefore requirement (1) for the tests was met.

```
interface Atm {
…   boolean Working();
    long long getAtmNo();
…
};
interface Account {
…   float getBalance();
    string getType();
    wstring getTypew();
    double getLimit();
…
};
interface Card {
…   long getNumber();
```

```
…
};
```

Listing 2: CORBA IDL interfaces used for performance testing

All the methods returned typed results. They did not accept any parameters nor did they carry out any processing. The goal was to measure the overhead of the distributed architectures and we wanted to omit any unnecessary influences on the results. The performances were measured for the following data types: integer, long, float, double, boolean and various string sizes, which satisfied the criterion (3).

The server side objects were located on one computer. With the above methods we have covered all the basic data types. You may notice that IDL data types are not the same as Java types. We have used these types to assure consistent mapping from IDL to Java, as shown in Table 3.

IDL Type	Java Type
boolean	boolean
long	int
long long	long
float	float
double	double
string	java.lang.String
wstring	java.lang.String

Table 3: Type mappings from IDL to Java

In Java, strings are represented using the `java.lang.String` class. Internally they are represented as sets of Unicode characters which are 2 bytes long. Native Java does not support 1 byte characters. Although IDL type `string` maps to the `java.lang.String` only 1 byte is transferred over the wire. If support for Unicode is needed then the mapping `wstring` should be used. On the other hand, CORBA objects written in other programming languages typically use `string`. To use the methods of a CORBA object written in Java which uses `wstring` conversion to other programs is needed. Therefore we decided to do the performance measurements with both mappings.

On the client side we designed an applet, which connected to the server objects and invoked the methods. The skeleton of the applet is shown in Listing 3. The two-way static invocation mechanism has been used. To simulate large string transfers we have measured response times for different string sizes. The method `Account.getType()` returned strings of 1, 1000, 2000, 3000, 4000, 5000 and 10000 characters, respectively.

```
String acc_type = "1";
my_account.setType(acc_type);
for(int j=0;j<15;j++) {
  long startTime=System.currentTimeMillis();
  for (n=0;n<NO_ITER;n++)
    acc_type=my_account.getType();
  long stopTime=System.currentTimeMillis();
  tMessage.append("Elapsed time "+
               (stopTime-startTime)+"\n");
} …
```

Listing 3: The client-side applet

Time was measured with the `System.currentTimeMillis()` method. The method returned elapsed time in milliseconds. To achieve the necessary level of accuracy all the invocations were repeated one thousand times. The results reported here are the average values of fifteen repetitions.

For CORBA/Java and Java RMI the tests were executed in three scenarios:

(a) The server object and the client applet ran on the same computer.

(b) The server object and the client applet ran on two separate computers.

(c) The server object ran on one computer, the client applets were simultaneously executed from 2, 3, 4, 5, 6, 7 and 8 clients.

The results of (a) and (b) lead to the conclusion regarding the network overhead and the comparison of (b) and (c) shows the performance degradation under heavy client load. With the described scenarios the criterion (2) has been satisfied. As you can see in Listing 3, the client applet invoked the methods continually without delays. This does not correspond to the typical user interaction, therefore the same results would be achieved with a much larger number of typical clients.

5.2. Server-side multithreading strategy

The testing method foresees that multiple clients invoke methods on a single server object. The multithreading strategy that is supported by the server object and the ORB is crucial and has an important impact on the performances. To determine the multithreading strategy used in the scenarios we have performed a simple test. We have defined a server object that had only one method. This method delays the program execution for 30

seconds (Listing 4). The method should be invoked by up to eight simultaneous clients. The execution time should be measured. If the time is around 30 seconds then the server side is capable of handling the requests in parallel. If the execution time is a around `no_clients * 30` seconds then the server side serialises the requests.

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

Listing 4: The server method for determining the multithreading strategy

This test is able to distinguish between the thread-per-servant architecture [12] that serialises the requests and architectures that treat the requests concurrently. It guarantees at least eight simultaneous requests which is sufficient for the described testing method.

5.3. Software and hardware testbed equipment

The Java source code was compiled and executed within JavaSoft's Java Development Kit 1.1.4 (JDK), which is the reference platform for Java development and was the latest version of JDK by the time of writing this article. For experiments with CORBA architecture the Inprise Visibroker for Java 3.0 has been used. The Visibroker is one of the most popular CORBA compliant object request brokers and is integrated into Netscape Navigator 4. Therefore it is the most commonly found ORB on the desktop. In certain performance tests with C++ programming language [13] it outperformed the Iona Orbix. As a profiler tool the JProbe Profiler 1.1 from KL Group has been used. All the computers used Microsoft Windows NT 4.0 Workstation as their operating system. Some subsequent experiments with Java 2 System Development Kit version 1.2 have shown, that for the tested scenarios there are no significant differences in performance, compared to version 1.1.4.

The server computer was a Pentium II 233 MHz computer with 64 MB RAM and the clients were Pentiums 200 MHz also with 64 MB RAM. In today's Internet applications the bandwidth is crucial. To simulate real world environment we have decided to connect computers into a 10 MBps Ethernet network. The network was free of other traffic. With the described software and hardware configuration requirement (4) was satisfied.

5.4. Related work

As far as we know there are no in-depth analyses of distributed object models and Java language that include performance comparisons. Existing research in distributed object performance evaluation is limited 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 [14, 15, 16, 17, 13, 18].

6. CORBA performance results

6.1. Single client

In the first scenario a single client applet invoked the methods on the server objects. The tests were done for two configurations: (a) basic data types and for varying string sizes where mapping to IDL (b1) string and (b2) wstring were used. Two scenarios were covered: (1) the server and the client program were located on the same computer and (2) on two different computers.

In Figure 4 the results for basic data types are gathered. The differences in method invocation time between the data types are marginal. In scenario (1) the average time was 2.17 ms and in scenario (2) it was 2.30 ms. The network connection had a very low impact on the times, only 6%.

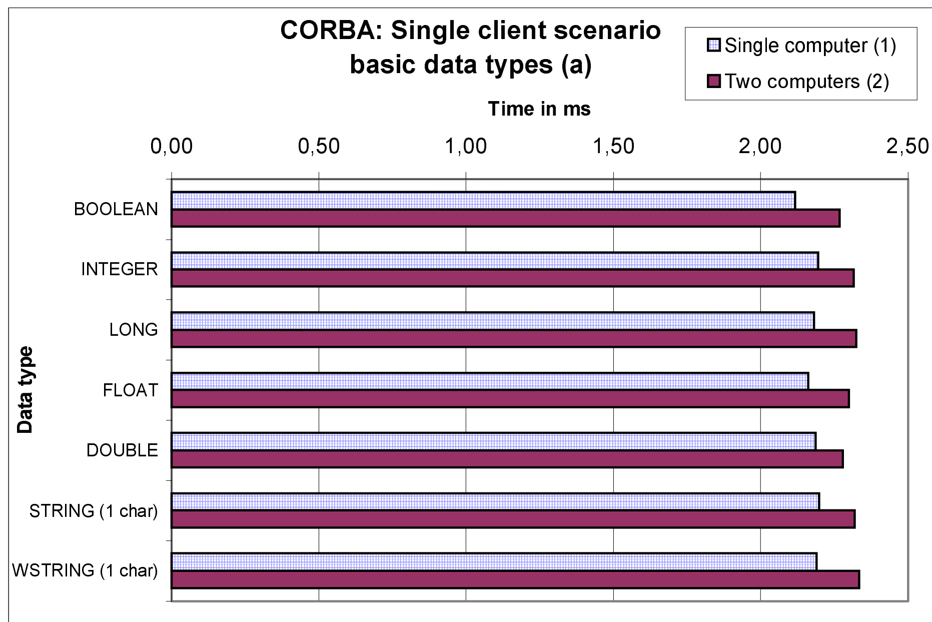

Figure 4. CORBA: Single client scenario, basic data types (a)

In the second test (b1) the influence of the string size, returned by the method was investigated. The results are shown in Figure 5. The overhead of the network grows with the string size from 18% for 1000 bytes string up to 51% for 10000 character string. Similar to RMI, the method response time is linearly dependant on the string size. It can be approximated with a linear function in the form $t(s) = ks + n$ where t is time in ms and s is string size in bytes. The comparison of the coefficient k between CORBA string, wstring and RMI string makes it possible to inference about the scalability. In the first scenario (1) the function is:

$$t(s) = 0{,}0018\ s + 2{,}3208 \qquad /1/$$

In the network scenario (2) the function is:

$$t(s) = 0{,}0029\ s + 1{,}9634 \qquad /2/$$

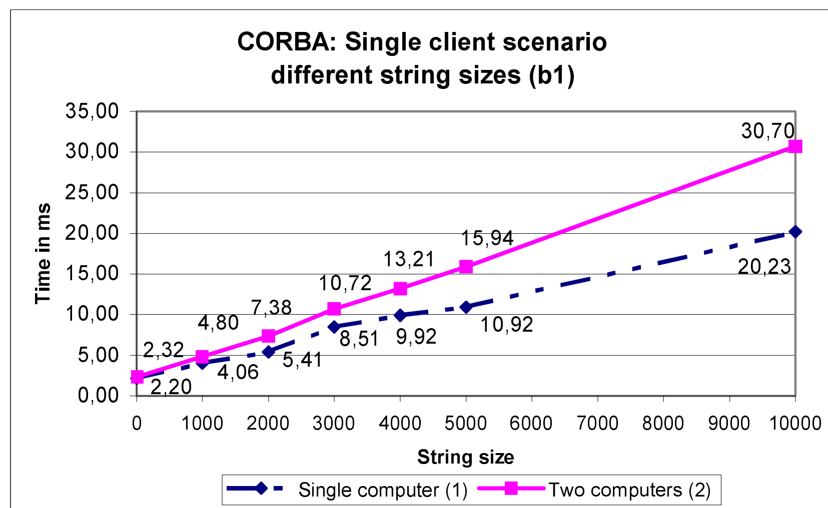

Figure 5. CORBA: Single client scenario, different string sizes (b1)

In the third test the wstrings (b2) were used. The results are shown in Figure 6. The network overhead is around 16%. The linear approximations for single computer and two computers scenarios are:

$$t(s) = 0{,}0116\ s + 2{,}6423 \qquad /3/$$

and

$$t(s) = 0{,}0135\ s + 2{,}1398 \qquad /4/$$

By CORBA string, the coefficient k in the two computers scenario is more than 60% larger then in the local scenario. By the wstring, the difference is only around 16%. In single computer scenario, wstring is handled more than 6 times slower than string. In the scenario with two computers the difference is around 4 times. A conclusion

can be drawn that wstring is not handled optimally by the tested CORBA implementation. This inefficiency is partially masked by the low network throughput, therefore in the two computers scenario the difference is lower than in single computer scenario.

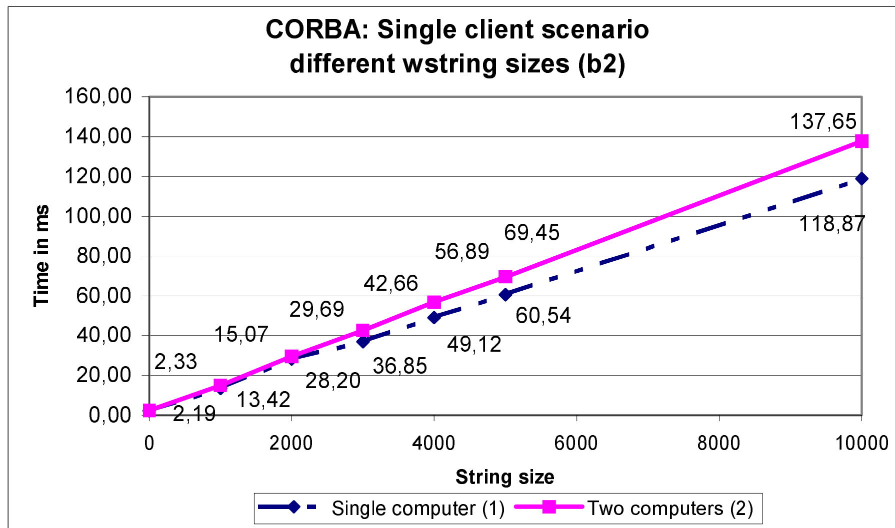

Figure 6. CORBA: Single client scenario, different wstring sizes (b2)

6.2. Multiple clients

We have investigated the performance degradation under heavy client load for (a) basic data types and for different string (b1) and wstring (b2) sizes. Figure 7 shows the results for the basic data types.

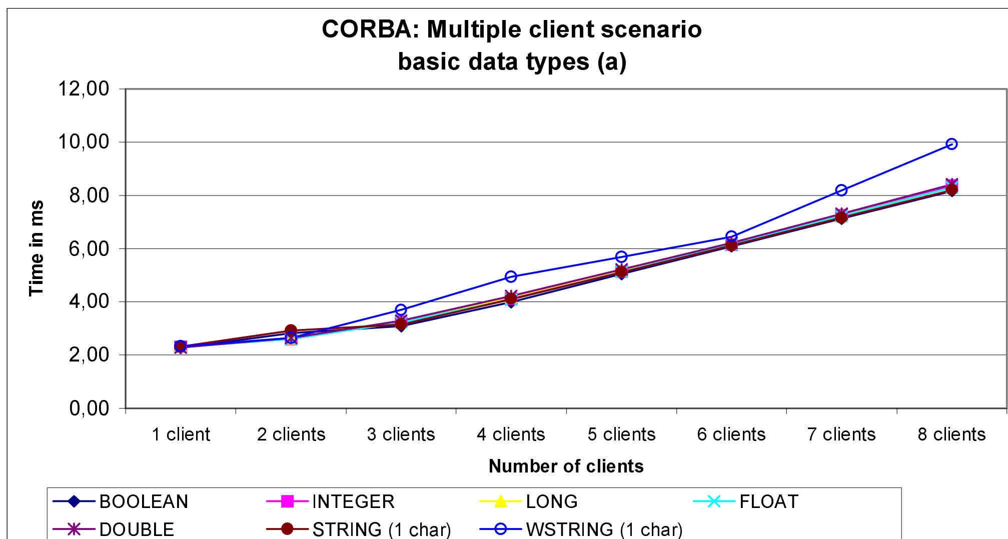

Figure 7. CORBA: Multiple client scenario, basic data types (a)

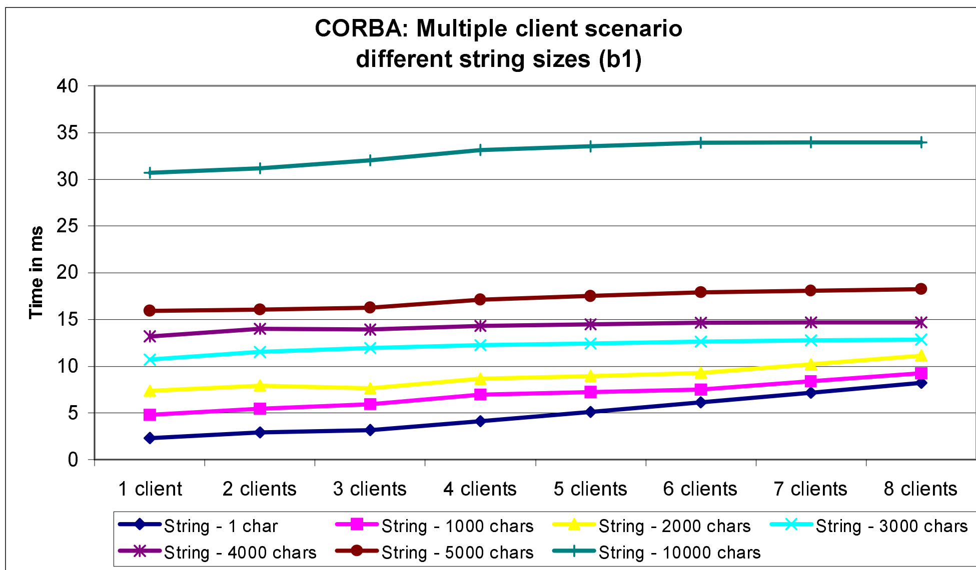

Figure 8. CORBA: Multiple client scenario, different string sizes (b1)

Single method invocation time grows with the number of clients. Behaviour is independent of the data type. By the six concurrent invocations the response time is 2.7 times longer and by eight concurrent invocations 3.6 times longer than by a single invocation. The exception is the 1 byte wstring data type, where the method invocation time is 2.8 and 4.3 times longer, respectively.

For the string method (b1), the average response time per client is shown in Figure 8. It can be seen that for the string ten times larger, the method invocation time is prolonged 6.4 times in one client scenario, 4.8 times in the four client scenario and 3.7 times in the eight client scenario. Irrespective of the number of simultaneous clients, the string method invocation time is linearly dependant from the string size.

The same analysis was done for the wstring method (b2), where for the string which was ten times larger you can see 9.1 times longer method invocation in one client scenario, 7.9 times in the four client scenario and 7.4 times in the eight client scenario. The average response time for various string size method per client is shown in Figure 9.

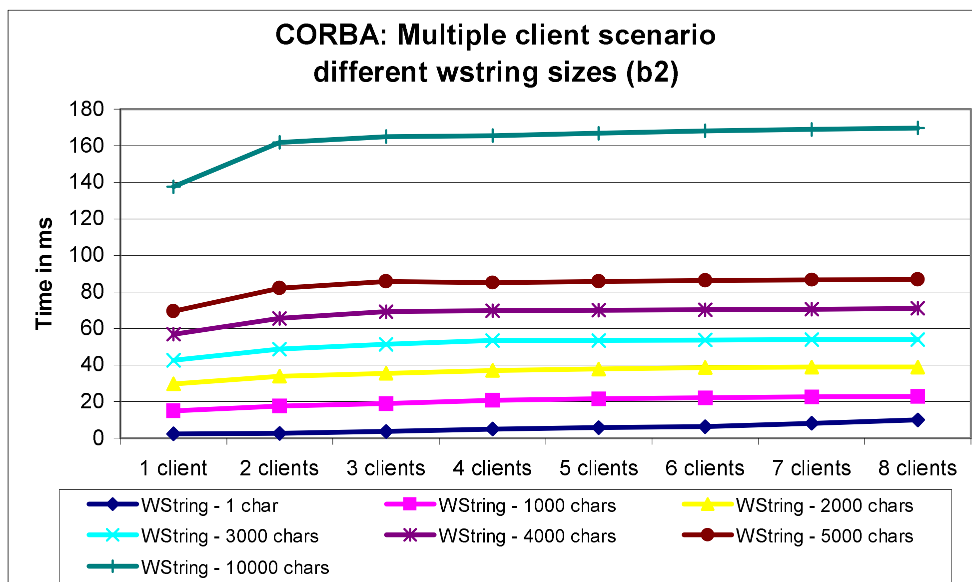

Figure 9. CORBA: Multiple client scenario, different string sizes (b2)

6.3. Multithreading strategy evaluation

When executing the test described in chapter 4.2 we found that the Visibroker supports parallel method invocations. The test lasted around 30 seconds regardless of the number of clients. This corresponds to [19] where it is stated that Visibroker supports thread pooling.

6.4. Code analysis

A profiler was used for code analysis. To investigate the most time consuming methods the client applet and the server application executing on separate computers were analysed. We have decided to analyse the `Account.getType()` method which returns a string. Test examples with IDL string were used. In our case the server sends the response to the client, therefore we should observe the server side for marshalling and the client side for de-marshalling and dispatching. In Table 4 the results for the client side and in Table 5 for the server side are presented. The results in percentages show the portion of the whole execution time consumed by a method.

On the client side there were 428 methods involved. When the string is only 1 character long the most time consuming method was the `GiopConnectionFactoryImpl.getMessage()` which was responsible for reception of GIOP/IIOP messages from the server object. Other important methods included `Socket.getInputStream()`, `Socket.getOutputStream()` which returned input and output streams for the socket and `GiopOutputStreamImpl.write_long()` and `GiopInput StreamImpl.read_long()` which handled the GIOP connection. When the string size increased the majority of time was spent in the string constructor method.

On the server side there were 640 methods executed. In the 1 character string scenario the most time consuming method was `Hashtable.put()` which is responsible for creating mappings from keys to values in a hashtable. The methods `GiopInputStreamImpl.read_long()`, `GiopOutputStreamImpl.write_long()`, `Socket.getInputStream()` and `SocketInputStream.read()` were involved in handling the communication. With the increasing string size the majority of the time was spent in the `String.getBytes()` method. This method converted characters into bytes. Each byte receives the 8 low-order bits of the corresponding character.

		String size in characters			
Method name	***Calls***	***1***	***1000***	***5000***	***10000***
.main.	1	27,28%	12,80%	3,99%	2,17%
String.<init>(byte[], int, int, int)	1072	<0,05%	52,71%	86,03%	91,93%
GiopConnectionFactoryImpl.getMessage()	1009	12,79%	6,88%	<0,05%	<0,05%
Socket.getInputStream()	2018	3,81%	1,76%	0,63%	0,32%
GiopOutputStreamImpl.write_long(int)	6064	3,54%	1,64%	0,53%	0,28%
StringBuffer.append(Object)	9	3,40%	1,47%	0,50%	0,61%
GiopInputStreamImpl.read_long()	5191	3,17%	1,47%	0,47%	0,27%
Socket.getOutputStream()	1010	2,19%	0,91%	0,27%	0,16%

Table 4: CORBA: Client side methods

		String size in characters			
Method name	***Calls***	***1***	***1000***	***5000***	***10000***
String.getBytes(int, int, byte[], int)	1195	2,16%	61,18%	87,51%	93,06%
Hashtable.put(Object, Object)	1082	10,01%	3,65%	1,05%	0,56%
GiopInputStreamImpl.read_long()	6085	5,47%	1,99%	0,58%	0,31%
GiopOutputStreamImpl.write_long(int)	5167	4,43%	1,61%	0,47%	0,25%
Socket.getInputStream()	2017	4,38%	1,60%	0,46%	0,12%
Class.forName(String)	47	3,49%	0,97%	0,28%	0,20%
SocketInputStream.read(byte[], int, int)	2017	3,35%	0,92%	<0,05%	<0,05%
GiopAdapterThread.doRequest(GiopMessage)	1002	3,12%	1,14%	0,33%	0,17%

Table 5: CORBA: Server side methods

7. Java RMI performance results

7.1. Single client

For evaluation of Java RMI the tests were executed following the same procedure as for CORBA/Java. Again, two types of results were obtained: (1) the server and the client programs executed on the same computer and (2) the server and the client programs run on separate network-connected computers. The tests were repeated for (a) basic data types (boolean, integer, long, float, double, 1 character string) and (b) for strings of different sizes.

Figure 10 shows the time of the six methods, needed to return the basic data types (a). The results for the boolean, integer, long, float and double were very close. In scenario (1) the average time was 1.54 ms. Only the method that returned string took longer, 1.68 ms. In scenario (2) the average time was 2.11 ms, which gives a 37% performance degradation. The string method took 2.39 ms. Performance degradation was slightly over 42%.

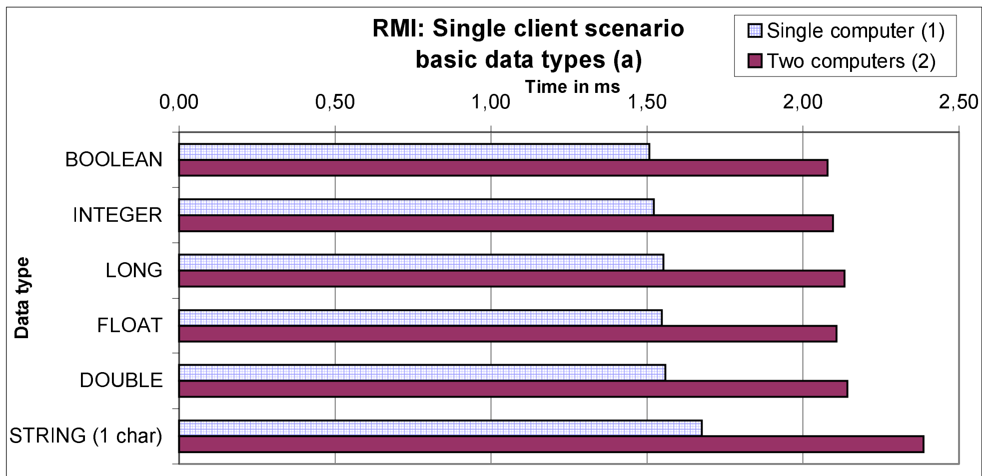

Figure 10. RMI: Single client scenario, basic data types (a)

RMI adds a significant overhead to the method invocation. On the same platform the native Java method invocation time for the same methods, which were implemented locally, was on average 400 ns (nanoseconds). However it is important to understand that RMI enables inter-process communication, that means communication between different Java Virtual Machines (JVMs).

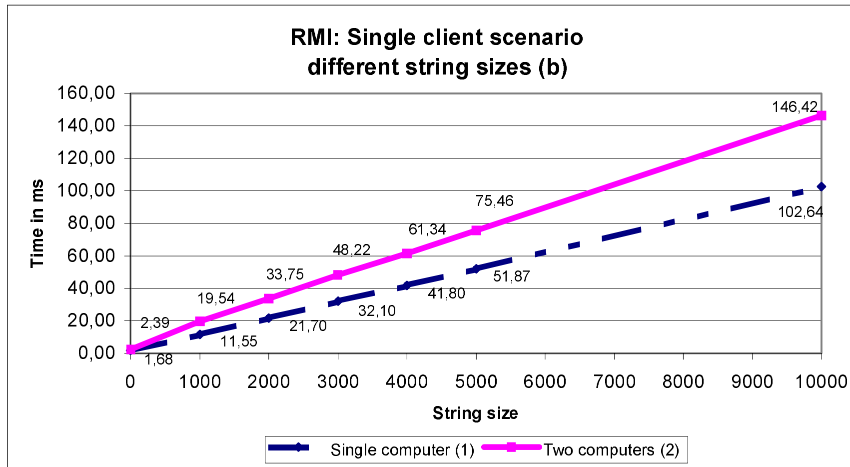

Figure 11. RMI: Single client scenario, different string sizes (b)

Figure 11 shows the results for varying string sizes (b). The overhead of the two computers scenario (2) over the single computer (1) varies and is around 50%. More interesting is the degradation caused by the string size. In both cases it can be approximated with a linear function in the form $t(s) = ks + n$ where t is time in ms and s is string size in bytes. In the scenario (1) the function is as follows:

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

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

$$t(s) = 0.0142\ s + 4.426 \qquad /6/$$

Comparing the coefficient k between single and two computers scenario shows that in the latter case the response times are around 40% higher. The result is between the times achieved by CORBA string and CORBA wstring.

7.2. Multiple clients

In the multiple client scenario the goal was to investigate the performance degradation under heavy client load. The tests were performed for (a) basic data types and for (b) different string sizes.

Figure 12 shows performance degradation for experiment (a). The method invocation time per client grows with the number of clients. With six clients the average invocation takes more than three times longer and with eight clients more than five times longer than with a single client. Data types boolean, integer, long, float and double behave similarly, only the one-character string shows marginally larger times.

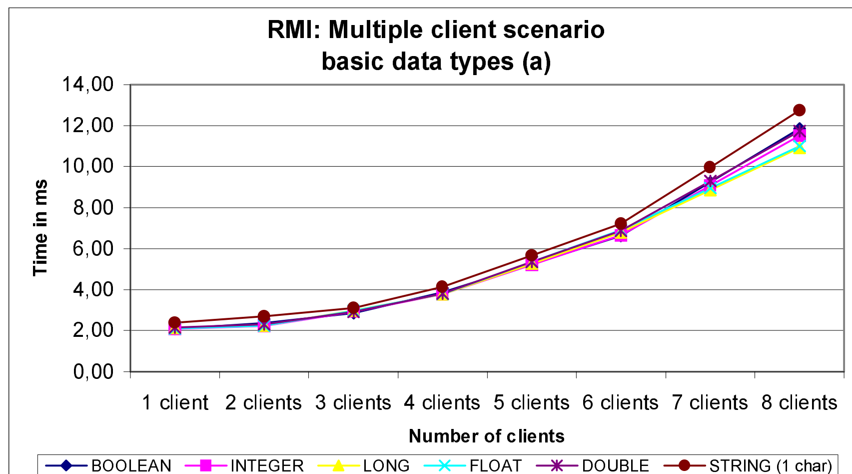

Figure 12. RMI: Multiple client scenario, basic data types (a)

The average response times per client for the string method with various string sizes is shown in Figure 13. For example, when the method returns a string ten time larger (10000 characters instead of 1000 characters) the invocation time is 7.5 times larger for the one client scenario, 6.3 times for four client and 5 times for the eight client scenario. Irrespective of the number of clients the response time for string-method invocation depends linearly from the string size.

7.3. Multithreading strategy evaluation

The test in chapter 4.2 leads us to the conclusion that RMI supports parallel method invocations on the server side because the test lasted around 30 seconds independent of the number of clients. The RMI Specification [6] guarantees that each method invocation originating from a different client virtual machine will execute in a different thread. This corresponds to our result.

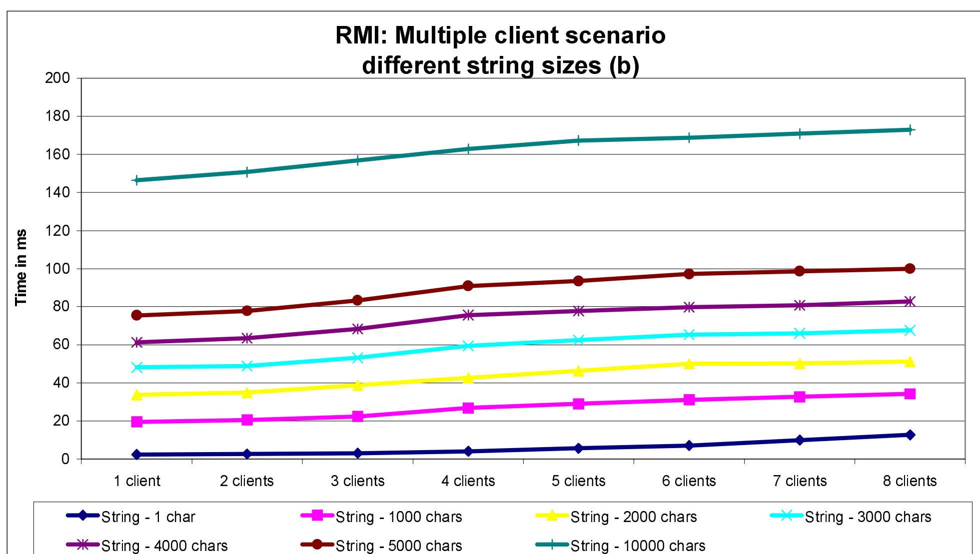

Figure 13. RMI: Multiple clients, different string sizes (b)

7.4. Code analysis

The code analysis was done the same way as with CORBA. In Table 6 the results for the client side are presented. Only the most important methods are listed. In the client applet using RMI there were a total of 40 methods executed. In the 1 character string scenario the most time consuming methods were `UnicastRef.newCall()`, `UnicastRef.invoke()`, `Naming.lookup()` and `UnicastRef.done()`. Method `newCall()` created a call object for a new remote method invocation on the object. Method `invoke()` executed the remote call and `done()` allowed the remote reference to clean up or reuse the connection. The `Naming.lookup()` method was used to associate the remote object with the specified name and to return a reference. The later method had to be used only once for a certain object. With the increasing string size the most time consuming method was `ObjectInputStream.readObject()` which was used to read an object from the stream.

In Table 7 the results for server side are presented. The server program executed 68 methods. On the server side when the string size increased the most time consuming method was `ObjectOutputStream.writeObject()`. This method was used to write an object to the stream. Important methods were also `StreamRemoteCall.getResultStream()`, `StreamRemoteCall.releaseInputStream()` and `Naming.rebind()`. The latter was responsible for rebinding the specified name to a new remote object and was executed only once for each object.

		String size in characters			
Method name	***Calls***	***1***	***1000***	***5000***	***10000***
ObjectInputStream.readObject()	1000	6,12%	84,95%	96,25%	97,90%
.main.	1	21,32%	3,91%	0,91%	0,47%
UnicastRef.newCall(RemoteObject, Operation[], int, long)	1002	31,41%	3,68%	0,86%	0,44%
UnicastRef.invoke(RemoteCall)	1002	17,82%	3,22%	0,73%	0,40%
Naming.lookup(String)	3	11,80%	1,99%	0,48%	0,25%
UnicastRef.done(RemoteCall)	1001	8,96%	1,62%	0,38%	0,19%

Table 6: RMI: Client side methods

		String size in characters			
Method name	***Calls***	***1***	***1000***	***5000***	***10000***
ObjectOutputStream.writeObject(Object)	1000	4,20%	88,40%	97,25%	98,42%
.TCP Accept-2.	1	34,24%	4,20%	0,96%	0,47%
StreamRemoteCall.getResultStream(boolean)	1001	27,97%	3,84%	0,90%	0,44%
StreamRemoteCall.releaseInputStream()	1002	8,72%	1,18%	0,26%	0,13%
Naming.rebind(String, Remote)	3	7,51%	0,98%	0,21%	0,12%
.TCP Accept-1.	1	4,90%	0,63%	0,13%	0,08%

Table 7: RMI: Server side methods

8. Interpretation of the results

8.1. Single client

In the single client scenario it can be seen that when using basic data types RMI is faster than CORBA (Figure 14). The difference is particularly noticeable when the client and the server are located on the same computer. For boolean, integer, long, float and double data types CORBA is on average 41% slower. With a one character string, the difference is smaller and is practically the same as if using the IDL mapping to string or wstring – CORBA is approximately 31% slower. The disadvantage of CORBA is lessened when the client and the server program are on separate computers. CORBA is then only around 9% slower with basic data types (boolean, integer, long, float and double). For the one character string CORBA is even faster than RMI in this scenario, although only marginally, 2.5% regardless of using either IDL string or wstring.

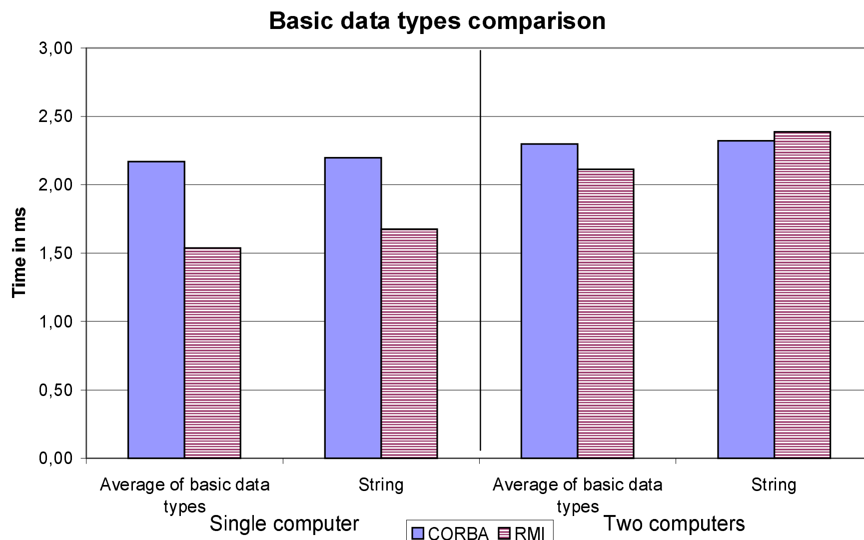

Figure 14. Basic data types comparison

The slower performance results of CORBA are affected by two factors. First, the complexity of the CORBA architecture (due to the support for several programming languages and platforms) is much greater, compared to RMI. In the client-server communication with CORBA, 428 methods were invoked on the client side and 640 on the server side. RMI on the other hand required only 40 and 68 methods, respectively, which is 10 times less.

Second, CORBA does not implement any communication optimisations when the client and the server are located on the same computer. Fortunately, this scenario is unusual in the real-world applications and not as important as the scenario where the client and the server are on the separate computers. This hypothesis is confirmed by the performance results of the latter scenario where the disadvantage of CORBA was much smaller. As a matter of fact, the performance degradation when objects are distributed across a network is much smaller with CORBA architecture and was only 6% compared to the 40% by RMI in our test.

The difference for larger strings is also very interesting. When CORBA uses IDL string it is on average 75% faster than RMI when executed on the same computer and 78% faster on separate computers. This result was expected, because IDL char is 1 byte long and Java character is 2 bytes long. When comparing CORBA's IDL wstring with RMI it can be seen that RMI is 18% faster on a single computer, but CORBA is 12% faster when the client and the server are on separate computers. The advantage of RMI in single computer scenario is a consequence of the local optimisations. The performance degradation when the client and server are moved to separate computers can be best observed from the coefficients *k* of the linear approximations. From equations /5/ and /6/ it can be seen that when the string size increases, the degradation with RMI is around 40%, with the CORBA string it is around 61% (equations /1/ and /2/) and with the CORBA wstring it is only 16% (/3/, /4/).

From the code analysis it can be seen that with RMI by the one character string the majority of time was spent in the methods which handled the communication. These method use native low-level functions written in C language through the Java Native Interface (JNI). When the string size increased the majority of time was spent in methods `ObjectOutputStream.writeObject()` and `ObjectInputStream.readObject()` for the server and the client programs, respectively. These methods were used for writing and reading the String objects to and from stream. With CORBA programs, by the one character string the majority of time was also spent in the communication methods. These communication methods use the low-level C functions through JNI, too. With larger strings, most of the time was used by the method `String.getBytes()` on the server side. This method was used for copying characters from the string into the destination byte array. This was necessary before the over-the-wire transmission could be performed. On the client side most of the time was used in the `String.<init>()`. This method was actually a string constructor which allocated a new String object from an array of 8-bit (byte) integer values.

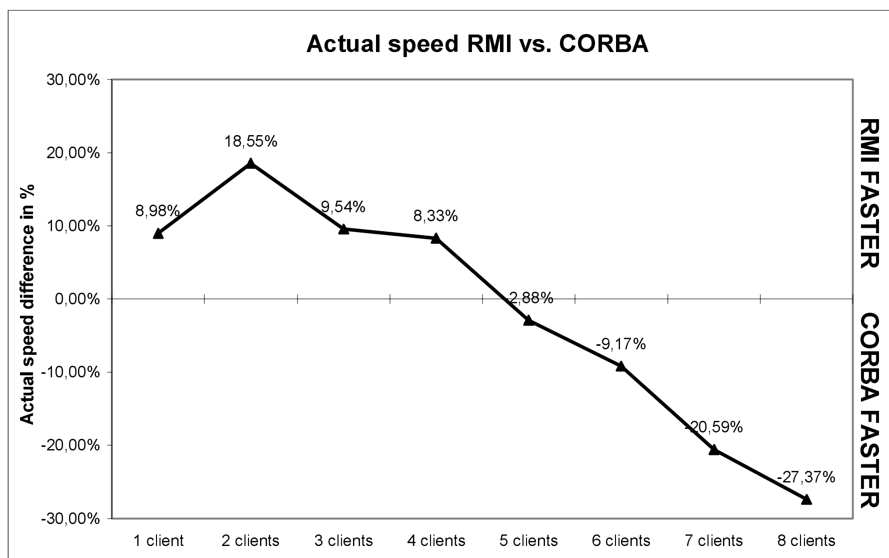

Figure 15. Actual speed RMI vs. CORBA: basic data types

8.2. Multiple clients

An even better picture of the performance issues can be achieved when observing the performance under a heavy client load. We would like to point out that the testing scenarios were designed so that the clients invoked the methods continuously. For a typical user environment this corespondents to a much larger number of clients. The actual number depends on the average delay between the invocations.

Figure 15 shows the actual performance comparison between CORBA and RMI for basic data types. RMI had the edge over CORBA for up to 4 clients. After that CORBA was faster. Please notice that in the single client

scenario RMI is faster for basic data types. As a consequence the performance degradations of both architectures under multiple client load differ a great deal. In Figure 16 the performance degradation for basic data types can be observed. It can be seen that under a heavy client load RMI has larger degradation than CORBA. In the eight clients scenario the degradation is 70%. We have already identified that both architectures support simultaneous server-side handling of client invocations. With the increasing number of simultaneous clients the lower complexity of the RMI architecture is nullified by the superior multiple client handling of the CORBA architecture, a consequence of the threading policy. RMI uses the thread-per-request threading policy, which adds significant overhead with the larger number of clients as each thread has to be created with the incoming request. The CORBA architecture, on the other hand, implements the worker thread pooling policy, where a pre-specified number of threads is already created. The incoming request therefore is assigned to the one of the available threads in the pool.

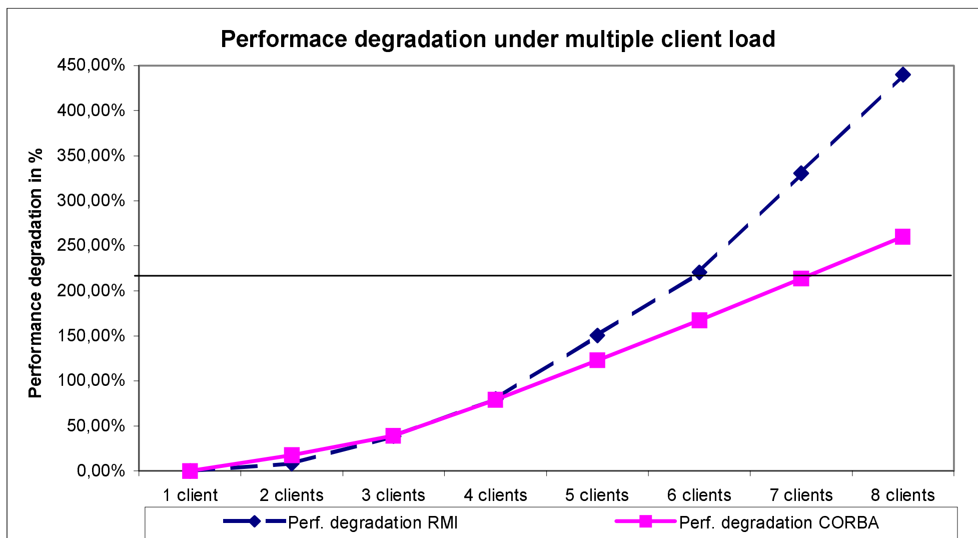

Figure 16. Performance degradation under load: basic data types

When observing different string sizes, two comparisons should be made. In Figure 17 the performance comparison between CORBA string and RMI/Java string is shown. With the increasing string size and larger number of clients CORBA is significantly faster than RMI. The comparison between CORBA wstring and RMI/Java string is shown in Figure 18. It can be seen that in most cases CORBA is still faster – up to 33%.

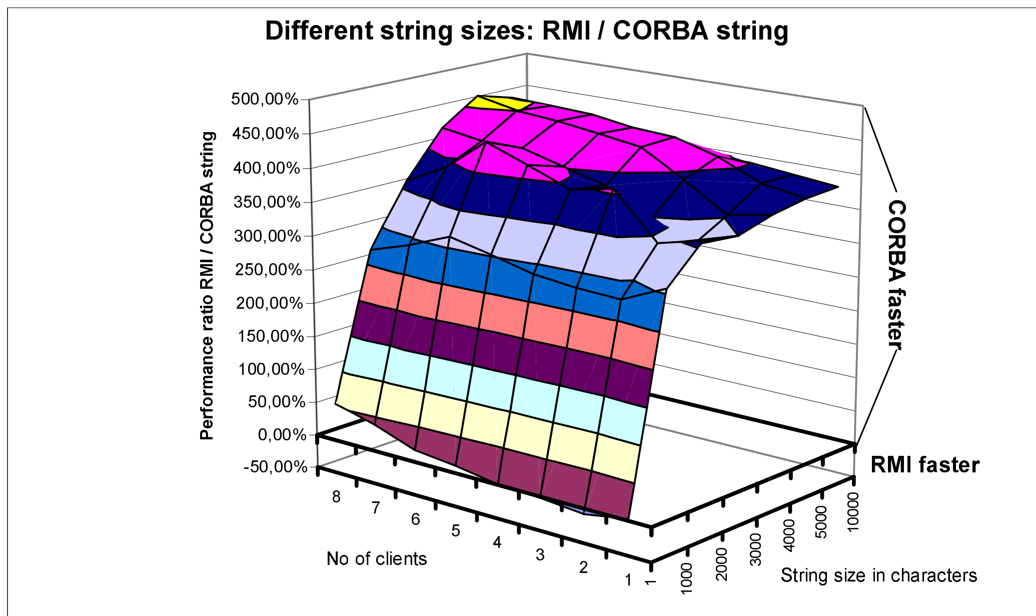

Figure 17. CORBA string vs. RMI string

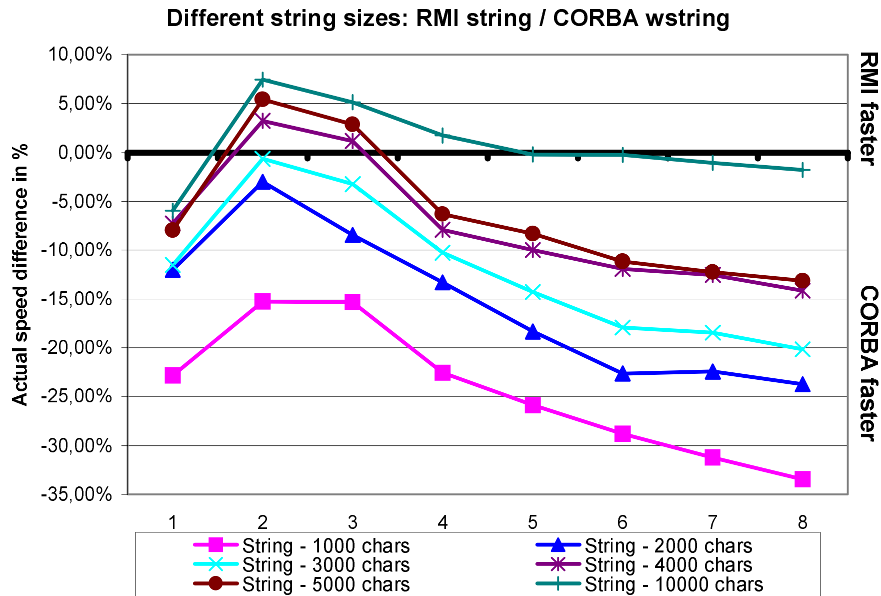

Figure 18. RMI string vs. CORBA wstring

In Figure 19 the performance degradation for the 10000 character string in the three scenarios is presented. The degradation is slight and is for the eight simultaneous clients less than 25%. The smallest degradation is achieved by CORBA string, followed by RMI/Java string and CORBA wstring. Again bear in mind that the differences are minor. The degradation of CORBA wstring is only 30% larger than RMI/Java string.

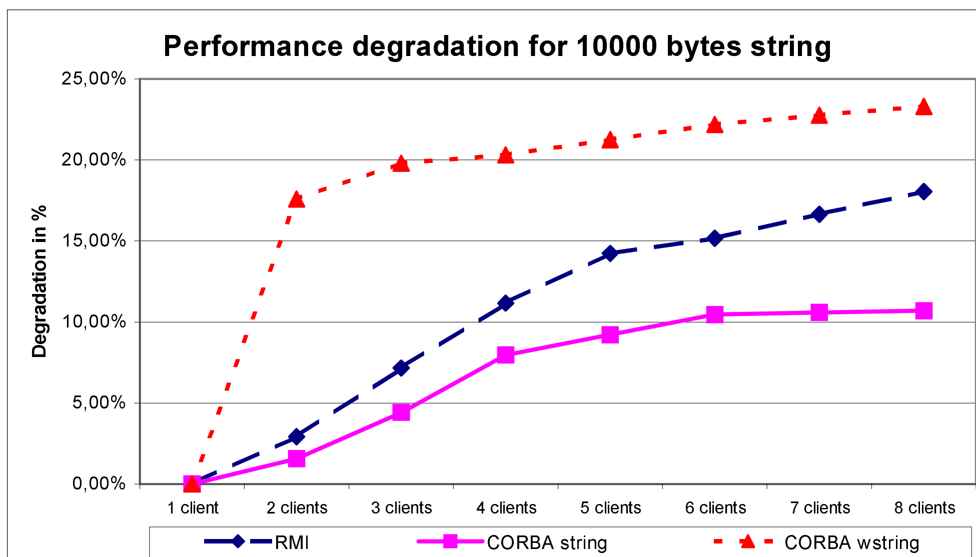

Figure 19. Performance degradation for 10000 bytes string

In these scenarios we are faced with a combination of two factors: the ability to efficiently handle multiple simultaneous method invocations and the ability to manage large data (in this case strings). As already stated, the IDL string is expected to be faster and indeed it is. But it is faster by more than 50% which confirms once again that CORBA handles multiple clients better. The implementation of CORBA we used – Visibroker – does not handle the IDL wstring optimally which is probably due to the fact that the wstring specification [4] has not been finished by the time Visibroker 3.0 was released.

8.3. Sources of overhead

The performance comparison has shown several differences. Listed bellow is a summary of the major sources of overhead that we have identified, with suggestions for optimisations:

- Much of the server side and some of the receiver side overhead is caused by the inefficient concurrency support – more exactly the multithreading exploitation shows comparably large overhead of the RMI, where the CORBA performs better. The multithreading policy thread-per-request proved to be the least efficient. To reduce the thread creation time the thread pooling can be used instead. The context switching should be minimised. In the leader/follower thread pooling architecture the context switching is successfully minimised because the request is not transferred from one thread to another, therefore providing better performance than the worker thread pooling architecture used by CORBA, although it is harder to implement.

- The other important source of the receiver side overhead can be found in the de-marshalling and in the presentation layer. De-marshalling overhead is minimised by using fast, de-layered, and flexible algorithms. Instead of the layered de-marshalling a perfect hashing and active de-marshalling can be implemented for best performance. Presentation layer overhead optimisation is achieved by generating optimised stubs and skeletons.

- CORBA uses an inefficient algorithm for de-marshalling strings. It converts IDL strings to java.lang.Strings. Java.lang.String is a static object. When a single character is added, a new String instance has to be created and the contents copied. The old instance is left to be collected by the Garbage Collector. By using larger strings a considerable overhead is caused. The solution is to use the StringBuffer instead and not to make the conversion to java.lang.String before the whole string is de-marshalled.

- CORBA does not implement local optimisations when the client and the server are located on the same computer. By using shared memory instead of the network protocol stack a considerable time saving could be achieved.

- An important overhead factor lies in the implementation of the Java Native Interface (JNI) which manages the execution of native C functions. JNI shows poor performance.

- Vast amount of the communication overhead resides in the low-level methods that take care of low-level communications. These methods are not implemented in Java, but rather they are written natively for each target platform in the C language and use the underlying operating system calls. Improving the low-level communication overhead requires optimisations in the native code that handles the communication and optimisations in the JNI. On one side this is the integration with the operating system network features, particularly advanced features, such as high-speed network interfaces, and real-time threads. On the other side something can be done with the buffer optimisations, their optimal management, and with the transport protocol tuning (for example, the socket lengths can be adjusted). None of the ORBs evaluated use 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.

- Other sources of overhead lie in the excessive data copying and local ORB method invocations, in stub and skeleton generation, and no caching architecture. Both ORBs can be optimised by omitting the unnecessary data copying, which would provide improvements most noticeable when using larger data sizes. Some optimisations could be achieved with the optimised range checking techniques. By a more careful internal design the count of the unnecessary local ORB method invocation can also be minimised. Implementation based optimisations, such as minimising the invocation overhead of frequently called methods with the optimisation for the common case, the replacement of large methods with efficient small special purpose methods, avoiding the repeated computation of invariant values, inlining and storing redundant data would all provide performance improvements.

8.4. Lessons learned

First of all it can be seen that neither the tested implementation of CORBA nor RMI in the tested version is considerably faster or slower. But from the test results some conclusions can be derived. In simple scenarios where the number of clients and the amount of data transferred is small both architectures demonstrate comparable results although RMI has an edge over CORBA. The advantage of CORBA can be seen when multiple clients are involved. Then the response times for CORBA were almost always better than for RMI. This is important for the scalability of the applications being developed.

Even more interesting is the comparison of large strings which are a common occurrence in today's programs. Because of the other programming languages that are supported by CORBA architecture two mappings of Java string to CORBA exist. We have shown that CORBA handles 1 byte character strings very well and is substantially faster than RMI – up to 80%. The 2 bytes wstring is not handled as well as regular string. The results of CORBA and RMI are closer, but CORBA still has an edge over RMI. In the future versions of CORBA the wstring support will certainly be improved. Therefore if there is no need for Unicode support then string should be used in the applications. Strings should also be used if the connection with legacy systems or objects in other programming languages is planned.

9. Conclusion

In this paper we have presented a qualitative and quantitative comparison of the two distributed object architectures most commonly used with Java. These are RMI and CORBA. We have compared important aspects such as features, ease of learning and ease of use. We have given focus to the in-depth quantitative analysis. With several testing scenarios we have measured performances for methods that returned different data types. We have carried out the test on a single computer, on two connected computers and under a heavy client load with up to eight simultaneous clients which invoked methods without any delays. The results reflect real world performances where it is common occurrence to have multiple clients accessing a single server. We have examined the multithreading strategies. With a profiler tool we have analysed the code and identified the most time consuming methods.

Through the presented facts we have confirmed the hypotheses *H1* and *H2*. We can conclude that CORBA/Java is suitable for large scale fully or partially web-enabled applications where legacy support is needed and good performances under heavy client load are expected. Java RMI on the other hand is suitable for small scale fully web-enabled applications where legacy support can be managed with custom build or pre-build bridges, where ease of learning and ease of use is more critical than performances are.

Through the evaluation of the related work we have established that this is, to the best of our knowledge, the first in depth analysis of CORBA and RMI in conjunction with Java programming language. Therefore we have provided a solid foundation to make a responsible decision about which architecture should be used on a particular software project. We have also contributed to the understanding of performance levels provided by the two distributed object models.

References

[1] Györkös József, Measurements in Software Requirements Specification Process, Microprocessing and Microprogramming, 1994, No. 40, Pg. 893-896

[2] Robert Orfali, Dan Harkey, Client/Server Programming with Java and CORBA, Wiley Computer Publishing, John Wiley & Sons, Inc., 1997

[3] Object Management Group, Richard Mark Soley, Christopher M. Stone, Object Management Architecture Guide, John Wiley & Sons, Inc., 1995

[4] Object Management Group, The Common Object Request Broker: Architecture and Specification, Revision 2.2, February 1998

[5] Vinoski Steve, CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments, IEEE Communications Magazine, Vol. 14, No. 2, February, 1997

[6] Sun Microsystems, Inc., Java Remote Method Invocation Specification, Sun, February 1997

[7] Birell, Nelson, Owicki, Network Objects, Digital Equipment Corporation Systems Research Center Technical Report 115, 1994

[8] Object Management Group, CORBAservices: Common Object Services Specification, Revised Edition, July 1997

[9] Object Management Group, Common Facilities Architecture, Revision 4.0, November 1995

[10] Juric B. Matjaz, The efficiency of distributed object models, ACM OOPSLA'99, New York, The Association for Computing Machinery, 1999

[11] Juric B. Matjaz, Zivkovic Ales, Rozman Ivan, Are Distributed Objects Fast Enough, JavaREPORT, SIGS Publications, May 1998

[12] Douglas C. Schmidt, Evaluating Architectures for Multi-threaded CORBA Object Request Brokers, submitted to the CACM special issue on CORBA edited by Krishnan Seetharaman

[13] Aniruddha S. Gokhale, Douglas C. Schmidt, Measuring and Optimizing CORBA Latency and Scalability Over High-Speed Networks, IEEE Transactions on Computers, Vol. 47, No. 4, April 1998

[14] Douglas C. Schmidt, Tim Harrison, Ehab Al-Shaer, Object-Oriented Components for High-speed Network Programming, 1st Conference on Object-Oriented Technologies, USENIX, Monterey, CA, June, 1995

[15] Aniruddha S. Gokhale, Douglas C. Schmidt, The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks, IEEE GLOBECOM '96 conference, November 18-22nd, 1996, London, England

[16] Aniruddha S. Gokhale, Douglas C. Schmidt, Measuring the Performance of Communication Middleware on High-Speed Networks, SIGCOMM Conference, ACM 1996, Stanford University, August 28-30, 1996

[17] Aniruddha S. Gokhale, Douglas C. Schmidt, Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks, IEEE 17th International Conference on Distributed Systems (ICDCS 97), May 27-30, 1997, Baltimore, USA

[18] Sai-Ali Lo, The Implementation of a Low Call Overhead IIOP-based Object Request Broker, Olivetti & Oracle Research Laboratory, April 1997 (www.orl.co.uk)

[19] Visigenic Software Inc, Visibroker for Java, Programmer's Guide, Version 3.0, Visigenic, September 1997

Vitae

Dr. 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 senior researcher at the University of Maribor. His interest include many areas of object technology with emphasis on distributed object models, performance analysis and optimization, the Java platform, and network computing. He is the author of more than 100 bibliographic units that include journal and conference articles, books, and educational material. He is also the head of the Distributed Object Models Performance and Efficiency Research Group (http://lisa.uni-mb.si/~juric/).

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

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