STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLIX**, Number 1, 2004

RMI VERSUS CORBA: A MESSAGE TRANSFER SPEED COMPARISON

FLORIAN MIRCEA BOIAN AND RAREŞ FLORIN BOIAN

Abstract. RMI (Remote Method Invocation) [5][1] and CORBA (Common Object Request Broker Architecture) [9][1] are two technologies for communicating between objects distributed across a network. The use of distributed objects is an attractive paradigm for designing distributed applications because of the simple abstraction layer that hides the network communication details. This article presents the results of tests that compared the speed of the message transfer between applications using these technologies. Five of the various implementations available for RMI and CORBA have been selected for the purpose of these tests. The interoperability between these implementations is also discussed.

1. Introduction

Distributed applications are very common in today's world. Every serious business has a web site that besides containing static information often provides automatically generated pages. Intranet applications within the domain of the same company usually provide direct communication between desktop applications and the server. The very popular file sharing P2P systems have to bring together computers from all over the world. In such environments, a distributed application has to be ready to work on various platforms (Intel, SPARC, Alpha, etc.) and operating systems (Windows, Solaris, Linux, etc.). More than that, such applications have to interface with legacy systems written in various programming languages. Although socket communication simplifies the communication in heterogeneous environments, the complexity of the communication protocols is ever increasing.

In 1989, the Object management Group (OMG) released the first specification of CORBA [9][1], a standard that allowed applications distributed in heterogeneous environments to exchange objects. The design made transparent to the developer the platforms on which the communicating applications were running as well as the programming language in which they were implemented.

Approximately at the same time, Microsoft created a similar product called Distributed Object Model (DOM) implemented in C++. The main restriction brought by DOM was the lack of implementations on non-Microsoft platforms, which made it unsuitable for heterogeneous network environments running other systems than Windows.

After the Java programming language was widely adopted as a programming language of choice, Sun Microsystems introduced in 1995 the Remote Method Invocation (RMI) [5][1] technology. RMI allowed Java objects owned by different

Received by the editors: 1/10/2004.

applications to call each other's methods and receive back the results of the process. The RMI benefited from Java's platform independence and portability, but did not adhere to the CORBA standard, thus it did not provide means to connect to legacy systems implemented in languages other than Java. The initial RMI implementation relied on URL naming to access remote objects. The Java Naming and Directory Service (JNDI) [2][8] brought to RMI a flexible and logical modality of naming services, beside the existing direct addressing approach [2][9].

Besides RMI, Sun Co. packaged with the release of Java 2, a native implementation of the CORBA standard, named Java IDL. With a few exceptions concerning internationalization [10], the Java IDL implementaion instrumented the Java distribution with full capabilities to communicate with system implemented in other programming languages.

To close the gap between the two supported standards (RMI and CORBA) and to bring the RMI ease of use into cross-language programming, Sun Co. released the RMI-IIOP solution. Using RMI-IIOP, RMI servers can communicate with CORBA clients implemented in any language, by respecting to a few restrictions [11]. The RMI-IIOP is basically built on top of Java IDL the rmic compiler (given the –iiop command line option) being able to generate IIOP stubs, ties and IDL.

Beside the four Java-based technologies mentioned above, the free omniORB [4] C++ implementation of CORBA was used to test the interoperability of CORBA and the performance difference among different implementation languages.

Choosing between using CORBA and using RMI has been a problem ever since Sun Co. decided to support both these competing technologies. Both options have advantages and disadvantges related to the ease of use, flexibility and level of integration with the Java language [12]. A study of their inter-communication performance may help taking a decisionin this matter.

2. Experimental setup

The five technologies used in this experiment are: classical RMI, classical RMI using JNDI as name service, RMI with IIOP, Java IDL and omniORB. For performance measurement purposes a simple Echo client-server application was developed. The clients send strings to the server, and the server objects transform the message to upper case letters and prefix it with the server request processing time measured in milliseconds . The resulting string is returned to the client. The same program was implemented for all five technologies. Figure 1 shows the communication diagram of the experimental setup.

Each of the five servers creates its context and then creates an EchoImpl object specific to its implementation, as seen in Figure 1. Each object is then registered (bind) to a name service. The RMI and RMI over JNDI use the standard rmiregistry name service. The remaining three implementations use tnameserv as name service, which is a default name service provided with the JavaIDL package. The omniORB implementation offers its own name service implementation (omniNames) but it can only be used by omniORB applications. After the EchoImpl object is registered, the servers stay idle waiting for requests.

The four Java clients are all implemented as standalone applications. The omniORB client is implemented as a WIN32 console application. Each program is given on the command line the name of the remote object to invoke and the string to send. Each client sends the same message a predefined number of times to the server, records timing information and then stops. The time is measured

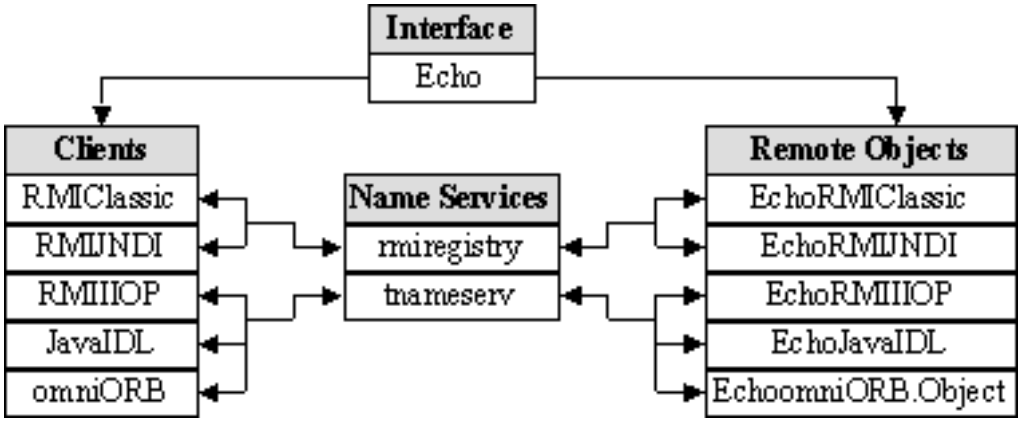

FIGURE 1. Architecture of the experiment

around the remote object invocation. For accuracy, the server-side processing time returned with the response is subtracted from the total time elapsed during the call. This approach eliminates the program startup time from the measured time and gives a more accurate performance measurement smoothing out inherent delays caused by the operating system by averaging across all the readings. Neither the clients, nor the servers write anything to the standard output (console) to avoid affecting the communication time measurement. In addition to measuring the performance of the remote method calls, the time necessary for naming lookups were also measured. The same multiple iteration approach was used as in the case of the remote invocations.

The tests were executed on an AMD Athlon 1.4 MHz computer running Windows 2000 Professional. To eliminate any delays caused by network traffic, the servers and the clients resided on the same machine. The servers were started and continued to function while the clients were executing one at a time.

3. Implementation

The interface necessary to implement the remote objects invoked by the client applications is shown in the table below. Two implementations are necessary, one using IDL for the CORBA technologies and another one using Java for the pure RMII applications.

IDL	Java

`interface Echo {`
`  string echoString(in string s);`
`}; //Echo`
	`import java.rmi.*;`
	`public interface Echo extends Remote {`
	`  public String echoString(String s)`
	`  throws RemoteException;`
	`}//Echo`

The IDL and Java interface implementations are very similar. Both define an echoString method that receives and input parameter of type string and returns a result of type string.

A generic Java implementation of the server object is presented below. The implementation differences between the four Java technologies consist only in the class declaration, namely in class extended and the interfaces that are implemented. The values of <EXTENDS> and <IMPLEMENTS> for each of the four Java based technologies are given in the table following the source code.

```
public class EchoImpl <EXTENDS> <IMPLEMENTS> {
   int invoke;
   public EchoImpl() throws RemoteException {
      super();
      invoke = 0;
   }//EchoImpl.EchoImpl

   public String echoString(String s) {
      invoke++;
      return invoke+"\t"+s.toUpperCase();
   }//EchoImpl.echo
}//EchoImpl
```

Technology	<EXTENDS >	<IMPLEMENTS >
Classical RMII	extends UnicastRemoteObject	implements Echo
RMI with JNDI		
RMI-IIOP	extends PortableRemoteObject	
Java IDL	extends EchoPOA	

Similar to Java, the C++ omniORB implementation is presented below.

4. Interoperability between the tested technologies

There are twenty-five possible ways to select a client and a server implementation from the five chosen technologies, but not all of them are interoperable. The ten combinations that are compatible with each other are presented in the table below.

As implementations of the CORBA standard, Java IDL, omniORB and RMI-IIOP are expected to be fully compatible to each other. Indeed, the first two technologies can be used on either the server or the client end of the application. However, their interaction with RMI-IIOP requires additional steps.

The communication between RMI-IIOP and Java IDL is conditioned by having access to the other end's technology stub. In other words, the variable CLASSPATH in the client's environment has to contain the class stub of the server object, and reverse.

```
class EchoImpl : public POA_Echo, public PortableServer::RefCountServantBase {

   private:
      int invoke;
   public:
       EchoImpl() {
            invoke = 0;
      }//EchoImpl.EchoImpl

      char* echoString(const char* s) {
         int i;
         char t[1000];
         invoke++;
         sprintf(t, "%d \t %s", invoke, s);
         _strupr(t);
         return CORBA::string_dup(t);
      }//EchoImpl.echoString
};//EchoImpl
```

Client \ Server	Java IDL	omniORB	RMI-IIOP	Classic RMI	RMI-JNDI
Java IDL	X	X	X		
omniORB	X	X			
RMI-IIOP	X	X	X		
Classic RMI				X	
RMI-JNDI					X

An intermediate level of Java IDL implementation is necessary in order to communicate between RMI-IIOP and omniORB, and in general with other non-Java CORBA applications.

The pure RMI implementations, with or without using JNDI cannot communicate outside their technology bounds. The communication between pure classic RMI and pure RMI-JNDI is made impossible by the naming protocol used. In order to access an object offered by an RMI-JNDI server, a classic RMI client needs to access the JNDI service to access the object. The reverse applies to an RMI-JNDI application trying to access an object offered by a classic RMI application.

The similar with RMI classic, RMI over JNDI can connect only with RMI over JNDI partner.

5. Results of the first experiment

The test client and server programs were executed several times with different iteration counts for name lookup operations and remote method calls. The iteration numbers ranged from 1 to $6x10^5$ in multiples of powers of ten. The results presented below are chosen from the run with most repetitions. Three aspects of the results are discussed below: variations in the measurements, the name lookup duration, and the duration of a remote call.

5.1. **Measurement Variations.** The measurements focused on the duration of the operations. Running the programs with iteration numbers different by orders of ten, made visible that initial operations are slower than the subsequent ones. The two tables below present the average duration of lookup and remote call operations measured over various numbers of repetitions. The configuration chosen is classic

RMI client and server but the behavior is consistent for all the other nine situations. The durations are expressed in milliseconds.

Lookup Iteration Count	**Lookup Total Time**	**Lookup Average Duration**
1	210	210.00
11	240	21.81
111	581	5.23
1111	2997	2.69
11111	13922	1.25
111111	123044	1.10

Method Call Iteration Count	**Method Call Total Time**	**Method Call Average Duration**
1	10	10.00
20	30	1.50
300	320	1.06
4000	2201	0.55
50000	15310	0.30
600000	169286	0.28

For both, name lookup operations and remote method call operations, it is visible that the first operations are significantly slower than subsequent ones. For name lookups, the operation duration is around one millisecond in for large number of iterations. Considering the second case with 11 iterations, if the duration of the first operation (210 ms measured on the line above) is subtracted from the total duration and the average of the remaining calls is averaged we get about 3 ms per call. This is still larger than the 1 ms obtained in the last case. Repeating this operation for the remaining rows, the lookup durations decrease slowly toward 1 ms, which proves that although the initial operations are by far the slowest, there are further interferences (possibly external to the test application) that slow down the communication. The same conclusion can be reached examining the remote method call measurements.

5.2. **Name Lookup Duration.** The table below presents the duration of name lookup measurements for the configuration with 111111 interations. It is apparent that the lookup operation durations are dependent on the client technology. The configuration involving omniORB are consistently faster than the rest. The RMI implementations are faster the the Java CORBA-compatible ones. This is most likely due to the lighter RMI implementation and the direct integration with the Java language, with the need for conversion in a universal format.

5.3. **Remote Method Call Duration.** The table below presents the duration of remote method call measurements for the configuration with 60000 interations.

The duration of the remote method calls are reversed in behavior when compared with those of the name lookup operations. This time the executionm times are dependant on the server technololgy instead of the client technology. The configurations with omniORB as server are significantly fatser then the rest, while those with omniORB as client are faster than thier counterparts. The higher speed

Client Technology	Server Technology	Avg. Duration per Lookup
Java IDL	Java IDL	1.418329
Java IDL	omniORB	1.354654
Java IDL	RMI-IIOP	1.434691
omniORB	omniORB	0.929962
omniORB	Java IDL	1.003096
RMI-IIOP	RMI-IIOP	2.515178
RMI-IIOP	omniORB	1.880749
RMI-IIOP	Java IDL	2.658926
Classic RMI	Classic RMI	1.107397
RMI-JNDI	RMI-JNDI	1.205047

Client Technology	Server Technology	Avg. Duration per Call (client)	Avg. Duration per Call (server)
Java IDL	Java IDL	0.997540	0.011038
Java IDL	omniORB	0.308578	0.003203
Java IDL	RMI-IIOP	1.004380	0.009730
omniORB	omniORB	0.161997	0.001988
omniORB	Java IDL	0.690818	0.009608
RMI-IIOP	RMI-IIOP	1.033056	0.010533
RMI-IIOP	omniORB	0.273586	0.002858
RMI-IIOP	Java IDL	0.988003	0.011101
Classic RMI	Classic RMI	0.282143	0.006160
RMI-JNDI	RMIJNDI	0.305676	0.007006

of the omniORB implementation is obviously explained by the fact that it is implemented in C++ and hence the program is a native and optimized executable instead of interpreted bytecode, which is the case of Java.

Another interesting aspect of the results is in the duration of teh server side method execution. Although the EchoImpl object executes the same code in all cases, the duration times differ amongst the RMI and and Java/CORBA implementations.

6. Conclusions

The experiments presented above were intended to assist in deciding the technology to use when starting a project. The results favor the C++ implementations which is expected given the optimized native code versus the Java bytecode. However, these tests cover only a small aspect of the many issues to be taken into account when choosing a technology. In addition to application speed, the following important criteria have to be considered before making a decision.

Licensing costs. While Java distributions are free, most of the C++ implementations are not.

Development team experience. Developing in a familiar environment will always yield faster and better results than using a new platform that first needs to be learned.

Flexibility. If the distributed application includes Java components, using RMI can make a developer's job easier with features such as distributed garbage collection, object passing by value, or dynamic class downloading [10].

Platform independence. If the application has to run in a heterogeneous environment, Java is a better option that saves the effort of adapting the implementation to each platform separately.

Productivity. Although the familiarity of the development team with one technology or another will be the decisive factor in this matter, it is generally considered that it is more productive to program in Java than in C++. This is mostly due to the lack of memory handling issues, hidden by the automatic garbage collector, and the consistent and simple design of the Java library.

References

[1] Harkey O. Client – server programming with Java and Corba, John Wiley, 1999
[2] Todd N., Szolkowski M. Locating Resources Usind JNDI (Java Naming and Directory Interfaces, SAMS, www.developer.com/java/ent/article.php/2215571
[3] Vinoski S. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. IEEE Communication Machine, 35, 2, 1997. http://www.iona.com
[4] * * * AT&T Cambridge Laboratory http://www.uk.research.att.com/omniORB or omniORB.sourceforge.net.
[5] * * * Java^{TM} Remote Method Invocation Documentation, http://java.sun.com/docs/guide/rmi/index.html
[6] * * * The JNDI Tutorial: Building Directory-Enabled Java Applications http://www.java.sun.com/products/jndi/tutorial/
[7] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, T. Nakatani, Overview of the IBM Java Just In Time Compiler, IBM SYSTEMS JOURNAL, Vol 39, No 1, 2000, http://www.research.ibm.com/journal/sj/391/suganuma.pdf
[8] * * *, Java^{TM} 2 SDK, Standard Edition Documentation, http://java.sun.com/j2se/1.4.2/docs/index.html
[9] * * * Object Management Group *CORBA Services Specification*, http://www.omg.org/library/csindx.html
[10] Java 2 RMI and IDL Comparison, http://lisa.uni-mb.si/∼juric/J2rmiidlc.pdf
[11] RMI-IIOP Programmer's Guide, http://java.sun.com/j2se/1.4.2/docs/guide/rmi-iiop/rmi_iiop_pg.html
[12] Java RMI & CORBA A comparison of two competing technologies, http://www.javacoffeebreak.com/articles/rmi_corba/

Babes Bolyai University, Cluj Napoca,Romania
E-mail address: florin@cs.ubbcluj.ro

Rutgers University, New York
E-mail address: boian@caip.rutgers.edu