Journal of Natural Sciences and Mathematics of UT, Vol. 4, No. 7-8, 2019

UDC: 004.434.045Java-048.22
Review

Implementation Issue Analysis of Java RMI and CORBA

Besar Huseini[1]*, Agon Memeti[1]

[1]*Faculty of Natural Sciences and Mathematics, University of Tetova, Tetova, RNM*

**Corresponding author e-mail: besar.huseini@unite.edu.mk*

Abstract

Extendable object-based program's (processes) architecture from logically partitioned object model to physically distributed object model is a key feature onto expanding the programming abilities, resource use/share possibilities and space-limited opportunities. The distributed environment, beside the design, requires new approaches and implementation methods. The idea behind this paper is to provide implementation analysis of Java remote method invocation (RMI) and Common object request broker architecture (CORBA) as most known distributed object oriented architectural platforms. Explicitly focused on the issues of such distributed object-oriented implementation methods.

***Keywords**: Java RMI, CORBA, Distributed Computing, implementation issues, analysis.*

Introduction

Forced by fast global network growth and connectivity, the actual limits of object oriented environments have been reached and that doesn't traditionally include two very important aspects like connectivity and interoperability. The wide-spread use of object based technology on one hand and the need of distributed environment on the other hand, lead to the increasing correlation of these technologies under several different implementation architectures, methodologies and communication paradigms [1]. Remote invocation is the most common communication paradigm on distributed systems. Generally, it represents duplex relationship between sender and receiver (*space decoupling*), with explicitly defined flow of communication based on real time non-parallel existence (*time decoupling*) of both communication tube endpoints. From architectural and design point of view, such distributed communication paradigm has risen to a higher level the ease of implementation [2].

Remote Method Invocation (RMI) and Common Object Request Broker Architecture (CORBA) can be defined, among the rest, as mostly used object based middleware solutions representing object model of distributed programming paradigm, object oriented analogs of RPC (remote procedure call) in distributed OO environment and its successors [3].

RMI is an effective solution, simple and easy to write framework for developing distributed applications providing location awareness and transparency. It is specifically created for Java; therefore, it is language dependable and always refers to Java RMI. It has a single purpose, to implement distributed objects.

CORBA represents middleware solution that provides integration, standardization and interoperability, separating object implementation from interfaces using Interface Definition Language (IDL) [4].

Using Java RMI is much easier to build complicated systems. Since it is language-specific (Java), complex applications development requires much smaller amount of code than most language undependable (IDL) distributed solutions like CORBA. It is much easier to evolve

and maintain RMI application thanks to its serialization support and Java specific garbage collector. On other hand, CORBA is language independent, using IDL its scope over distributed system build upon more different technologies is bigger. It inherits all the advantages of its target languages and underlying technologies it is built upon, yet, their limitations too [5].

The paper is organized in three sections. The first section describes the process of method invocation on CORBA and RMI.

On the second section, more detailed implementation analysis of both technologies is provided. It includes the basic RMI and CORBA method implementation and code examples from different approaches to the same problem. The issue of object persistence and portability is included during this section too.

On the last section, deeper analyze of major implementation issues of CORBA is done, on technical point of view, but also the market impact over it.

Briefly On Methods

Java RMI relies on a protocol called the Java Remote Method Protocol (JRMP). Java relies heavily on Java Object Serialization, which allows objects to be marshaled (or transmitted) as a stream. Since Java Object Serialization is specific to Java, both the Java RMI server object and the client object have to be written in Java [5].

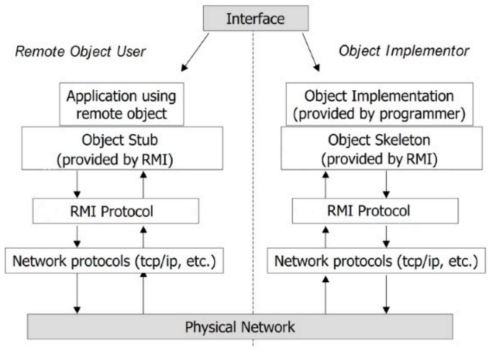

Figure 1. RMI interface [6]

Each Java RMI Server object defines an interface, which can be used to access the server object outside of the current Java Virtual Machine (JVM) and on another machine's JVM. The interface exposes a set of methods, which are indicative of the services offered by the server object [7].

A client RMI call invokes the client-side *stub* (the proxy of the remote method that resides on the client's machine). The stub uses Object Serialization to marshal the arguments, i.e., render argument object values into a stream of bytes that can be transmitted over a network. The *stub* then passes control to the Java Virtual Machine's RMI layer. The *skeleton* on the server side dispatches the call to the actual remote object after un-marshaling the arguments into variables. The stub and skeleton programs are generated by the *rmic* compiler [6].

On other hand, CORBA uses stubs and skeletons in much the same way as Java RMI. The Proxy or a local representative for the client side is called the IDL stub; the server-side proxy is the IDL skeleton.

A *stub* is a local proxy for a remote object. It presents the same interface as the server object, but runs on the same computer as the client which is used for more functionality like support for Dynamic invocation. For Marshalling the request and the response, the information is delivered in a canonical format defined by the IIOP protocol used for CORBA

interoperability on the Internet. IDL stub makes use of dynamic invocation interface for marshalling on the client side.

A *skeleton* is a remote interface to the server object implementation. It runs on the same computer as the server object and provides an interface between the server object's implementation and other objects. IDL Skeletons use the Dynamic Skeleton Interface for un-marshalling the information. The request (response) can also contain Object Reference as parameters; remote object can be passed by reference [5].

The stub and skeleton are connected via an ORB. The ORB forwards method invocations from the stub to the skeleton and uses a special object called an Object Adapter (OA), which runs on the same computers as the server object.

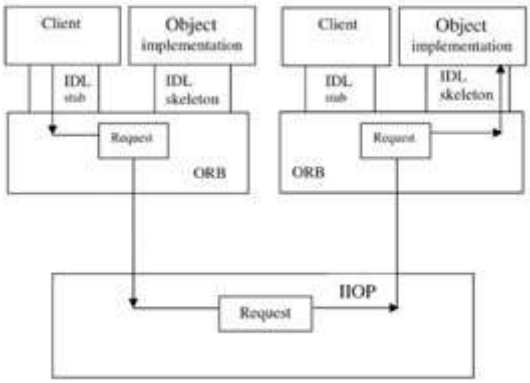

Figure 2. CORBA remote invocation when client and server are using different ORBs [8]

The OA activates the server object, if required, and helps to manage its operation. You can think of the ORB as analogous to the remote reference and transport layers of Java RMI, and the OA as being like the remote registry. The OA is sometimes referred to as a *Basic Object Adapter (BOA)* [9].

Methods Implementation

RMI is simpler to work with since the Java developer does not need to be familiar with the Interface Definition Language (IDL). In general, however, CORBA differs from RMI in the following areas:

CORBA interfaces are defined in IDL and RMI interfaces are defined in Java. RMI-IIOP allows you to write all interfaces in Java (see RMI-IIOP).

CORBA supports in and out parameters, while RMI does not since local objects are passed by copy and remote objects are passed by reference.

CORBA was designed with language independence in mind. This means that some of the objects can be written in Java, for example, and other objects can be written in C++ and yet they all can interoperate. Therefore, CORBA is an ideal mechanism for bridging islands between different programming languages. On the other hand, RMI was designed for a single language where all objects are written in Java. Note however, with RMI-IIOP it is possible to achieve interoperability.

CORBA objects are not garbage collected. As we mentioned, CORBA is language independent and some languages (C++ for example) does not support garbage collection. This can be considered a disadvantage since once a CORBA object is created, it continues to exist until you get rid of it, and deciding when to get rid of an object is not a trivial task. On the other hand, RMI objects are garbage collected automatically [10].

To implement RMI, three processes are needed:

- *client* – to invoke a remote object,
- *server* – the remote object is owned by the server process and
- *object registry* – this is a name server for objects. The names registered in the server can be referenced for particular objects.

Two different kinds of classes that can be used in RMI: *remote* and *serializable classes*.

A *remote object* is an instance of a remote class. When a remote object is used in the same address space, it can be treated just like an ordinary object. But, if it will be used from the outside of the address space, the object has to be referenced by an object handle. There are differences between them but most of the time can be used in the same way.

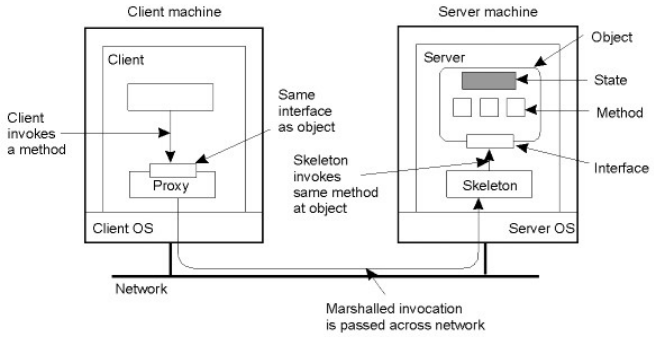

Figure 3. RMI implementation [11]

Correspondingly, a *serializable object* is an instance of a *serializable class*. A serializable object can be copied from one address space to another. This means a serializable object can be a parameter or a return value. If a remote object will be returned, actually it is the object handle being returned. Technically, the object could be both remote and serializable, it can be compiled in JDK, but there will be a run-time error.

A class is serializable if it implements the *java.io. Serializable* interface. Any subclasses of a serializable class are serializable classes, and any data inside a serializable class are also serializable data.

It is a little difficult to define a remote class. A remote class includes two parts: the interface and the class itself.

The remote interface must have the following aspects:

- being declared public,
- extend the interface *java.rmi.Remote* and
- the exception *java.rmi.RemoteException* must be thrown in all the methods inside the interface.

And the remote class itself has the following characteristics:

- it must implement a remote interface and
- extends the *java.rmi.server.UnicastObject* class, the object of which exists in the address space of the server.

Transparency (location and language) is the key at CORBA. The location transparency means that when a client calls a remote object method, it seems it calls an object just located in the same address space as the client itself. It is implemented via a stub code, which is produced by an IDL compiler. The language transparency indicates the objects being called can be implemented in any language without knowing by the calling client [8]. The IDL module defines a naming scope for all IDL type and interface definitions of the application to avoid naming conflict with other application IDL specifications. With the IDL interface described, objects can be set-up and client programs written for these objects.

First, we *compile the IDL interface* and do *the programming language mappings*. Mappings are language- specific and CORBA provides IDL compiler for each. The compiler generates outputs comprising the stubs used by clients of the described object types and skeletons used by servers that implement them.

The second step is to *implement IDL interfaces*. It is done through completing or reusing skeleton code, in any language following a set of specific mapping rules.

Third, *object servers and object clients are implemented*. The written server programs include the implementation of objects as well as the generated skeletons. This program contains the code required for connecting to ORB, creating the object and its reference public. Another hand, a set of client programs act on the object by accessing their attributes and invoking operations. These programs include the stub code, GUI and the application code.

The next step is to *install and configure servers,* to automate activation when requests are received and *distribute and configure client programs* so clients debugged programs must be distributed to the sites where they will be used.
The last step is the *distributed execution* itself, where the ORB will perform the communication between the distributed objects [12].

A. Comparison of the amount of work for implementation code

Table 1. Interface [13]

Java RMI	***CORBA***
Interface definition: Server *package JRMIIRC;* *import java.rmi.*;* *public interface JRMIIRCServerI extends java.rmi.Remote {* *void register_callback(JRMIIRCCallback callbackClient) throws RemoteException;* *void send_message(String message) throws RemoteException;* *void unregister_callback(JRMIIRCCallback callbackClient) throws RemoteException;}* **Interface definition: Client** *package JRMIIRC;import java.rmi.*;* *public interface JRMIIRCCallback extends java.rmi.Remote {* *void message_callback(String message) throws RemoteException;}*	**Interface** *CORBAIRCCallback {* *void message_callback(in string message);* *};* *interface CORBAIRCServerI {* *void register_callback(in CORBAIRCCallback callbackClient);* *void send_message(in string message);* *void unregister_callback(in CORBAIRCCallback callbackClient);* *};* *};*

Java RMI (Sun ONE Studio 5 Standard Edition), CORBA (ORB implementation in Java 2 Standard Ed. with Sun ONE Studio 5 Standard Edition)

Table 2. Instantiating Remote Object [13]

Java RMI	***CORBA***
Instantiating remote object *server=(JRMIIRC.JRMIIRCServerI)* *Naming.lookup("rmi://"+serverTF.getText()+* *"/JRMIIRCServer");*	**Instantiating remote object** *java.util.Properties props=new java.util.Properties();* *props.put("org.omg.CORBA.ORBInitialPort","900");* *props.put("org.omg.CORBA.ORBInitialHost",* *serverTF.getText());* *orb=ORB.init(new String[] {},props);* *POA rootpoa=POAHelper.narrow* *(orb.resolve_initial_references("RootPOA"));* *NamingContextExt* *root=NamingContextExtHelper.narrow* *(orb.resolve_initial_references("NameService"));* *NameComponent[] name=new NameComponent[1];* *name[0]=new* *NameComponent("CORBAIRCServer","");* *server=CORBAIRC.CORBAIRCServerHelper.narrow* *(root.resolve(name));*

Java RMI (Sun ONE Studio 5 Standard Edition), CORBA (ORB implementation in Java 2 Standard Edition with Sun ONE Studio 5 Standard Edition)

Table 3. Registering Callback And Sending Message To Server [13]

Java RMI	***CORBA***
Registering callback procedure *callbackServer=(JRMIIRC.JRMIIRCCallback) new* *JRMIIRCCallbackImpl("JRMIIRCCallback",this);* *server.registriraj_callback(callbackServer);* **Sending message to server** *server.send_message(nickTF.getText()+":* *"+messageTA.getText());* [reference: an overview of distributed programming techniques]	**Registering callback procedure** *callbackServer=new* *CORBAIRCCallbackImpl(this,orb);* *rootpoa.activate_object(callbackServer);* *callbackServerRef=* *CORBAIRC.CORBAIRCCallbackHelper.* *narrow(rootpoa.servant_to_reference(* *callbackServer));* *server.register_callback(callbackServerRef);* *rootpoa.the_POAManager().activate();* *Thread callbackServerThread=new* *Thread(callbackServer);* *callbackServerThread.start();* **Sending message to server** *server.send_message(nickTF.getText()+":* *"+messageTA.getText());*

Java RMI (Sun ONE Studio 5 Standard Edition), CORBA (ORB implementation in Java 2 Standard Edition with Sun ONE Studio 5 Standard Edition)

Object Persistence

In commercial ORBs, object references *persist*. They can be saved by clients as strings and subsequently be recreated from those strings. The methods required to perform these operations are object_*to_string* and string_*to_object* respectively, both of which are methods of class *Orb*. With the latter method, an object of (CORBA) class *Object* is returned, which must then be 'downcast' into the original class via method *narrow* of the appropriate 'helper' class.

Suppose that we have a reference to a *StockItem* object and that this reference is called *itemRef.* Suppose also that the ORB on which the object is registered is identified by the variable orb. The following Java statement would store this reference in a String object called *stockItemString*:

String stockItemString = orb.object_to_string(itemRef);

The following statements could subsequently be used to convert this string back into a *StockItem* object reference:

org.omg.CORBA.Object obj = orb.string_to_object(stockItemString);

StockItem itemRef = StockItemHelper.narrow(obj);

Of course, the client would have needed to save the original string in some persistent form (probably within a disc file).

Since Java IDL supports *transient objects* only (i.e., objects that disappear when the server process closes down), the above technique is not possible. However, it is possible to implement an object so that it stores its state in a disc file, which may subsequently be used by the object's creation method to re-initialize the object [8].

Portability (RMI-IOOP)

To overcome the language-specific disadvantages of RMI when compared with CORBA, Sun and IBM came together to produce RMI-IIOP (Remote Method Invocation over Internet Inter-Orb Protocol), which combines the best features of RMI with the best features of CORBA and gives an answer to the big portability problems. Using IIOP as the transport mechanism, RMI-IIOP implements OMG standards to enable application components running on a Java platform to communicate with components written in a variety of languages (and vice-versa) – but only if all the remote interfaces are originally defined as Java RMI interfaces. It is intended to be used by software developers who program objects in Java and wish to use RMI interfaces (written in Java) to communicate with CORBA objects written in other languages. Using RMI-IIOP, objects can be passed both by reference and by value over IIOP [14].

So, with the newly adopted CORBA standards Objects by Value and Java to IDL mapping have made the birth of RMI-IIOP possible. This also means any ORB, which has adopted these standards will work with RMI-IIOP [8].

Against CORBA?!

The most obvious technical problem is CORBA's complexity—specifically, the complexity of its APIs. Many of CORBA's APIs are far larger than necessary. For example, CORBA's object adapter requires more than 200 lines of interface definitions, even though the same functionality can be provided in about 30 lines. See section 0 above for comparison example.

Another problem area is the C++ language mapping. The mapping is difficult to use and contains many pitfalls that lead to bugs, concerningthread safety, exception safety, and memory management. On contrary, Java RMI is language dependable, fully java implemented with total absence of this problem [15].

CORBA provides quite rich functionality, but fails to provide two core features like *security* and *versioning*.

CORBA's unencrypted traffic is subject to eavesdropping and man-in-the-middle attacks, and it requires a port to be opened in the corporate firewall for each service. This conflicts with the reality of corporate security policies. Java RMI implements custom socket factories to control address binding, control connection establishment (such as to require authentication), and to control data encoding (such as to add encryption or compression). Java use serialization too [16].

Deployed commercial software requires middleware that allows for gradual upgrades of the software in a backward-compatible way. CORBA does not provide any such versioning mechanism (other than versioning by derivation, which is utterly inadequate). Instead, versioning a CORBA application generally breaks the on-the-wire contract between client and server. This forces all parts of a deployed application to be replaced at once, which is typically infeasible. Java is very specific on the versioning system issues and there are explicitly built libraries for that [17]. Example:

String version = Runtime.class.getPackage().getImplementationVersion();

Microsoft never embraced CORBA and instead chose to push its own DCOM (Distributed Component Object Model). No language mappings exist for C# and Visual Basic, and CORBA has completely ignored .NET.

Java works perfectly well over the Microsoft machines. The Microsoft JVM won the PC Magazine Editor's choice awards in 1997 and 1998 for the best Java support.

Three basic approaches are deeply considered onto Java-.NET class-level interoperability. First is *porting the platform*, which means to port the entire .NET platform to Java or vice versa. In addition, compile the developed code to the alternate platform. Second is *cross-compilation*, convert Java or .NET source or binaries to .NET or Java source or binaries. Third one is *bridging* which means to run the .NET code on a .NET Common Language Runtime (CLR), and the Java code on a Java Virtual Machine (JVM) or a Java EE application server [18].

Another important factor in CORBA's decline was XML. During the late '90s, XML had become the new silver bullet of the computing industry: Almost by definition, if it was XML, it was good.

Using Java and XML through Java Architecture for XML Binding applies a lot of defaults thus making reading and writing of XML via Java very easy. JAXB is a Java standard that allows us to convert Java objects to XML and vice versa. JAXB defines a programmer API for reading and writing Java objects from XML documents. It also defines a service provider that allows the selection of the JAXB implementation [19].

The on-the-wire encoding of CORBA contains a large amount of redundancy, but the protocol does not support compression. This leads to poor performance over wide-area networks.

The specification ignores threading almost completely, so threaded applications are inherently nonportable (yet threading is essential for commercial applications). CORBA does not support asynchronous server-side dispatch.

Conclusions

We discussed and analyzed CORBA and Java RMI platforms and their implementation issues. They have their strong and weak sides and surely, they contain them both. The similarities and differences among them, for particular issues, vary from slightly different to completely distinct. That surely is not based only on the implementation differences or programming

philosophy but the differences are often inherited from the approach on attempting to solve a problem and the platform design itself.

They both have their advantages and disadvantages and the decision on which we can rely on is a matter of further discussion based on real needs. For building bigger applications in the sense of technologies involved and the number of users, CORBA will surely be a wise choice. On the other hand, for smaller applications and language dependable, Java RMI would surely do better. Yet, the CORBA's bad cooperation with XML and .NET is a big handicap having in mind the large size of the market they cover nowadays.

References

[1]. F. Plasil and M. Stal, ""An Architectural View of Distributed Objects and Components in CORBA, Java RMI, and COM/DCOM"," A Submission to Software - Concept and Tools, vol. LXVI, 1998.

[2]. G. Coulouris, J. Dollimore, T. Kindberg and G. Blair, "Distributed Systems, Concepts and Design", 5th ed., Boston: Addison-Wesley, 2012, pp. 185-224, 335-378.

[3]. W. Grosso, "Java RMI, 1st ed., O'Reilly, October 2001, p. 572.

[4]. R. Metkowski and P. Bala, ""Parallel computing in java: looking for the most effective RMI implementation for clusters"," in Proceedings of the 6th international conference on Parallel Processing and Applied Mathematics, Poznan, 2006.

[5]. A. Patil, R. Korde and K. Sabharwal, ""Comparison of Middleware Technologies - CORBA, RMI & COM/DCOM"," [Online]. Available: https://pdfs.semanticscholar.org/f4a2/2d516dbb7482ab01722760d8da586b87c2a0.pdf. [Accessed 17 May 2017].

[6]. R. Ramos, "Firenze Physics Department and INFN," [Online]. Available: http://hep.fi.infn.it/JAVA9.pdf. [Accessed 18 May 2017].

[7]. J. Weijia and Z. Wanlei, "Distributet Network Systems, From Concepts to Implementation", vol. XV, Boston: Springer, 2005, pp. 163-172, 419-425.

[8]. R. Bao. [Online]. Available: https://pdfs.semanticscholar.org/bfe2/264c88facfddfb87d3563663128a76dd16bc.pdf. [Accessed 10 May 2017].

[9]. W. Zhou. [Online]. Available: http://www.kiv.zcu.cz/~ledvina/vyuka/PSI/literatura/scc751sg.pdf. [Accessed 18 May 2017].

[10]. Q. H. Mahmoud, ""Java Feature Stories"," Oracle, January 2002. [Online]. Available: http://www.oracle.com/technetwork/articles/javase/rmi-corba-136641.html. [Accessed 14 May 2017].

[11]. A. S. Tanenbaum and M. V. Steen, "Distributed Systems: Principles and Paradigms", 2nd ed., Pearson Prentice-Hall, 2007.

[12]. P. Merle, C. Gransart and J. Geib. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.4010&rep=rep1&type=pdf. [Accessed 17 May 2017].

[13]. M. Golub and D. Jakubovic. [Online]. Available: http://www.zemris.fer.hr/~golub/clanci/mipro2005_Parallel.pdf. [Accessed 19 May 2017].

[14]. J. Graba, "An Introduction to Network Programming with Java", Addison-Wesley, 2007, pp. 136-186.

[15]. M. Henning, ""The Rise and Fall of CORBA"," ACMQueue, vol. 4, no. 5, pp. 28-34, 30 June 2006.

[16]. "JDK 5.0 Documentation," [Online]. Available: http://docs.oracle.com/javase/1.5.0/docs/guide/rmi/socketfactory/index.html. [Accessed 09 06 2017].

[17]. "J2SE SDK/JRE Version String Naming Convention," [Online]. Available: http://www.oracle.com/technetwork/java/javase/versioning-naming-139433.html. [Accessed 09 06 2017].

[18]. A. Bridgwater, "Bridging the Java to .NET interoperability divide," [Online]. Available: http://www.computerweekly.com/blog/CW-Developer-Network/Bridging-the-Java-to-NET-interoperability-divide. [Accessed 12 06 2017].

[19]. L. Vogel, Vogella GmbH, 06 10 2016. [Online]. Available: http://www.vogella.com/tutorials/JavaXML/article.html. [Accessed 11 06 2017].

[20]. N. Gray, ""Performance of Java middleware - Java RMI, JAXRPC, and CORBA"," in The Sixth Australasian Workshop on Software and System Architectures, Brisbane, 2005.

[21]. S. Hunt, B. Jeram, M. Plesko and C. Watson. [Online]. Available: https://www.researchgate.net/publication/2331204_The_Implementation_of_an_OO_Control_System_API_with_CORBA. [Accessed 20 May 2017].

[22]. C. Munos and J. Zalewski, Kluwer Academic Publishers, 2001. [Online]. Available: https://drive.google.com/file/d/0BwFWPDFEijEMZWloYm9DTm9mMHM/view. [Accessed 14 May 2017].