STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLV**, Number 1, 2000

NEW INTERACTION MECHANISMS BETWEEN JAVA DISTRIBUTED OBJECTS

FLORIAN MIRCEA BOIAN AND CORINA FERDEAN

Abstract. This article proposes some solutions to very common problems and requirements concerning the interaction between Java objects spread across several machines. Thus, an object should be able to access another remote object without knowing where that object resides. This location transparency induces also migration transparency, allowing objects to be found and accessed by their clients, even if they are changing their location.

Another extension, of the standard interacting protocols for the collaboration between distributed objects, could be the definition of patterns used to match the remote objects, which have certain attributes or which implement the services specified in the pattern.

1. Introduction

Basically, every distributed system implies two or more active entities (processes, threads, running objects) performing computations, in different address spaces, potentially on different hosts. Also, these active execution entities should be able to communicate.

For a basic communication mechanism, the Java programming language supports the *sockets*, which are flexible and sufficient for general communication. However, sockets require the client and server to define applications-level protocols to encode and decode messages for exchange, and the design of such protocols is cumbersome and sometimes error-prone. Besides, even if these protocols allows the communication between programs written in different languages and on heterogeneous platforms, they are not flexible and neither extensible.

Another distance communication mechanism, as an alternative to sockets, is *Remote Procedure Call* (RPC), which abstracts the communication interface to the level of a procedure call. Instead of working directly with sockets, the programmer has the illusion of calling a local procedure, when in fact the arguments of the call are packaged up and send to the remote target of the call. RPC systems encode

2000 *Mathematics Subject Classification.* 68M14.

1998 *CR Categories and Descriptors.* C.2.4 [**Computer Systems Organizations**]: Computer-Communication Networks – *Distributed Systems.*

arguments and return values using an external standard data representation, such as XDR.

However, the RPC mechanism is not suitable for the distributed object systems, where communication between program-level objects residing in different address spaces is needed. In order to match the semantics of object invocation, distributed object systems require *remote method invocation* or RMI. In such systems, a local surrogate (stub) object manages the invocation on a remote object.

2. Java RMI mechanism and JRMP protocol

Java RMI (Remote Method Invocation) offers a distributed object model for the Java Platform. Thus, the Java RMI system assumes the homogeneous environment of the Java virtual machine (JVM), and it uses the standard Java object model, extending it into a distributed context.

RMI is unique in that it is a language-centric model that takes advantage of a common network type system. In other words, RMI extends the Java object model beyond a single virtual machine address space.

The underlying communication protocol used in Java RMI mechanism is JRMP. This protocol allows the object methods to be invoked between different Virtual Machines across a network, and actual objects can be passed as arguments and return values during method invocation. The JRMP protocol uses object serialization to convert object graphs to byte-streams for transport. Any Java object type can be passed during invocation, including primitive types, core classes, user-defined classes, and JavaBeans. Java RMI could be described as a natural progression of procedural RPC (Remote Procedure Call), adapted to an object-oriented paradigm for the Java platform environment.

In the following we'll describe shortly how a typical object interaction works in Java RMI.

Any object whose methods are available to be invoked by another Java object must publish these methods by implementing an interface, which extends the `java.rmi.Remote` interface.

To make a remote object accessible to other virtual machines, a program typically registers it with the RMI registry. The program supplies to the registry the string name of the remote object as well as the remote object itself.

A client program, in fact a Java object, which wants to access a remote object, must supply the remote object's string name to the registry that is on the same machine as the remote object.

The string name accepted by the RMI registry has the syntax "`rmi://hostname:port/remoteObjectName`", where `hostname` and `port` identify the machine and port, respectively, on which the RMI registry is running and `remoteObjectName` is the string name of the remote object. `hostname`, `port`, and the prefix, "`rmi:`"

are optional. If `hostname` is not specified, then it defaults to the local host. If `port` is not specified, then it defaults to 1099. If `remoteObjectName` is not specified, then the object being named is the RMI registry itself.

The registry returns to the caller a reference, called *stub*, to the remote object.

As it turns out, the communication between Java objects is structured in a layer hierarchy, depicted in Figure 1.

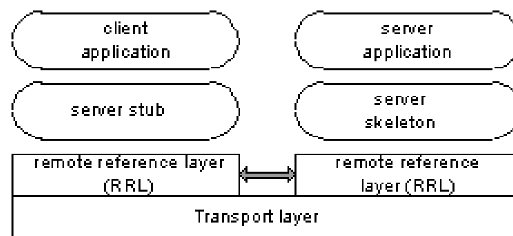

FIGURE 1. RMI Architecture

When the object's methods are invoked remotely, its arguments are *marshalled* and sent from the local virtual machine to the remote one, where the arguments are *unmarshalled* and used. When the method terminates, the results are marshalled from the remote machine and sent to the caller's virtual machine. An important observation which worths mentioning here is that the remote objects (that implement the Remote interface) are passed by reference, and the others objects by value (also, they must implement the `Serializable` interface). Another observation is that passing by value a Java object in a different Java environment is equivalent with a primitive form of object migration, where the "mobile agent" (the object passed by value) is static, and can be called when and if its destination environment decides).

3. Providing Java RMI with support for location transparency and fault-tolerance

A natural question, which arises in a Java RMI communication context, is how it would be possible for a client object to access a remote object, without having to know a priori the server object location. This feature of location independency becomes a fundamental requirement if it is assumed that Java server objects could change their location, migrating between different hosts.

3.1. **Using JNDI and LDAP.** The first solution proposed for providing location transparency is based on complementary technologies like JNDI (Java Naming Directory Interface) and LDAP (Light Directory Access Protocol).

Our discussion begins with a brief description of these technologies, followed by the presentation of the support they provide for accessing Java objects transparently.

JNDI (Java Naming Directory Interface). The Java Naming and Directory Interface (JNDI) is an application programming interface (API) that provides *naming* and *directory* functionality to applications written using the Java programming language [4, 5]. This API is defined to be independent of any specific directory service implementation, allowing a variety of directories to be accessed in a common way.

The JNDI architecture consists of an API and a service provider interface (SPI).

The primary goal for Java applications, that use the JNDI API, is to access a variety of naming and directory services. The services can be plugged in transparently, by using SPI [11]. This interface allows the developers of different naming/directory service providers to hook up their implementations so that the corresponding services are accessible from applications that use JNDI [4, 5].

These implementations include those for the *Initial Context* and for its descendent contexts that can be plugged in dynamically to the JNDI architecture to be used by the JNDI application clients.

JNDI is included in the Java 2 SDK, v1.3 and later releases. It is also available as a Java Standard Extension for use with the JDK1.1 and the Java 2 SDK, v1.2.

As it turns out, in order to use the JNDI, besides the JNDI classes, also, one or more service providers should be available. The Java 2 SDK, v1.3 includes three service providers for the following naming/directory services:

- Lightweight Directory Access Protocol (LDAP);
- Common Object Request Broker Architecture (CORBA) Common Object Services (COS) name service;
- Java Remote Method Invocation (RMI) Registry.

In this survey, we use LDAP as a directory service that provides a repository for the Java distributed shared objects.

LDAP. LDAP was originally developed as a front end to X.500, the OSI directory service. X.500 defines the Directory Access Protocol (DAP) for clients to use when contacting directory servers. DAP is a heavyweight protocol that runs over a full OSI stack and requires a significant amount of computing resources to run. LDAP runs directly over TCP and provides most of the functionality of DAP at a much lower cost [6].

LDAP directory service is based on a client-server model. One or more LDAP servers contain the data making up the LDAP directory tree. An LDAP client connects to an LDAP server and asks it a question. The server responds with the answer, or with a pointer to where the client can get more information (typically, another LDAP server). No matter which LDAP server a client connects to, it sees the same view of the directory; a name, presented to one LDAP server, references the same entry it would at another LDAP server. This is an important feature of a global directory service, like LDAP [7, 8].

In LDAP, directory entries are arranged in a hierarchical tree-like structure that reflects political, geographic and/or organizational boundaries. Entries representing countries appear at the top of the tree. Below them, there are entries representing states or national organizations. Below them, might be entries representing people, organizational units, printers, documents, or any other entities someone needs to define.

In addition, LDAP allows the control and the configuration of which attributes are required and allowed in an entry, through the use of a special attribute called `objectclass`. The values of the `objectclass` attribute determine the schema rules the entry must obey.

Using LDAP and JNDI to extend Java distributed computing. In the Java distributed computing context, LDAP provides a centrally administered and possibly replicated service for use by Java applications spread across the network. For example, an application server might use the directory for registering objects that represent the services that it manages so that a client can later search the directory to locate those services as needed.

The JNDI provides an object-oriented view of the directory, thereby allowing Java objects to be added to and retrieved from the directory without requiring the client to manage data representation or location execution issues.

There are different ways in which Java applications can use the directory to store and locate objects. Thus, an application might store (a copy of) the object itself, a reference to an object, or attributes that describe the object.

In general terms, a Java object's serialized form contains the object's state and an object's reference is a compact representation of addressing information that can be used to contact the object. An object's attributes are properties that are used to describe the object; attributes might include addressing and/or state information.

Which of these three ways to use depends on the application/system that is being built and how it needs to interoperate with other applications and systems that will share the objects stored in the directory. Another factor is the support provided by the service provider and the underlying directory service.

Transparent Java remote objects communication. In this survey, we will show how Sun's LDAP service provider supports the binding of `java.rmi.Remote` objects into directories. When `java.rmi.Remote` objects and/or RMI registries are bound into an LDAP enterprise-wide shared namespace, RMI clients can look up `java.rmi.Remote` objects without knowing on which machine the objects are running [1, 9, 10].

Instead of storing the entire serialized state of an object, which could be too large, it is preferable to store, into directories, a reference to that object. For that purpose, JNDI offers the `javax.naming.Reference` class. This class makes it possible to record address information about objects not directly bound to the directory service. The reference to an object contains the following information [7]:

- The class name of the referenced object;
- A vector of `javax.naming.RefAddr` objects that represents the addresses, identifying the connections to the object;
- The name and location of the object factory to use during object reconstruction.

`javax.naming.RefAddr` is an abstract class containing information needed to contact the object (e.g., via a location in memory, a lookup on another machine, etc.) or to recreate it with the same state. This class defines an association between content and type. The content (an object) stores information required to rebuild the object and the type (a string) identifies the purpose of the content.

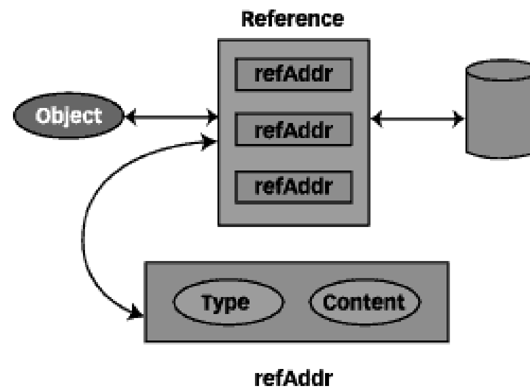

FIGURE 2. The relation between a Reference, RefAddr, Type, and Content

RefAddr also overrides the methods `java.lang.Object.equals(Object obj)` and `java.lang.Object.hashcode()` to ensure that two references are equal if the content and type are equal. `RefAddr` has two specific subclasses, namely, `javax.naming.StringRefAddr` and `javax.naming.BinaryRefAddr`, which store strings, and respectively arrays of bytes. For example, a string reference address could be an IP, URL, hostname, etc.

4. Example

In the following, we'll give a simple example of storing references to Java remote objects in a LDAP directory. We mention that it is also possible to store copies of objects as streams of bytes, but this alternative requires much more space, and it isn't flexible, as it is not possible to change an object implementation once it was bound in the directory service. Using references to objects provide this flexibility, and besides it saves a lot of space in the directory tree.

Our example is constructed conforming to the following steps, performed on different machines:

(1) We define on the machine `cronos.cc.ubbcluj.ro` a Java shared object `HelloImpl`, which implements a `Remote` interface called `Hello`. We register this object with the `rmiregistry` name service, on the same machine.

```
import java.rmi.*;
public interface Hello extends Remote {
    public String sayHello() throws RemoteException;
    }
```

Program 1. Hello.java

```
import java.rmi.*;
import java.rmi.server.*;
public class HelloImpl extends UnicastRemoteObject
implements Hello {
    public HelloImpl() throws RemoteException {
    }

    public String sayHello() throws RemoteException {
        return ("Hello, the time is " + new java.util.Date());
    }
}
```

Program 2. HelloImpl.java

```
import java.rmi.*;
public class ServHello {
```

```
public static void main(String args[]) {
    try {
        System.setSecurityManager(new RMISecurityManager());

        // create a registry if one is not running already.
        try {
            java.rmi.registry.LocateRegistry.createRegistry(1099);
        } catch (java.rmi.server.ExportException ee) {
            // registry already exists, we'll just use it.
        } catch (RemoteException re) {
            System.err.println(re.getMessage());
            re.printStackTrace();
        }
        Naming.rebind("rmi://cronos.cc.ubbcluj.ro/hello",
                new. HelloImpl());
    } catch(Exception e) {
        System.out.println("Error: "+e.getMessage());
        e.printStackTrace();
    }
}
}
```

Program 3. ServHello.java

(2) On another machine, `hermes.cc.ubbcluj.ro`, we create a reference of type `StringRefAddr` to the `HelloImpl` object, which contains an RMI URL of the form `rmi://cronos.cc.ubbcluj.ro/RemoteObjectName` and it is bound to a name into a LDAP directory. We also define a value for the `javacodebase` attribute, which will be used by the service provider to find the stub class for the remote object.

```
import java.util.Hashtable;
import javax.naming.*;
import javax.naming.directory.*;
import java.rmi.*;
public class HelloServ {
public static void main(String argv[]) {
    String rmiurl = "rmi://cronos.cc.ubbcluj.ro/hello";

    // Set up environment for creating the initial context
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
```

```
        env.put(Context.PROVIDER_URL,
                "ldap://rave.scs.ubbcluj.ro:389/cn=CORI,
                o=CCUBB,c=RO");

        try {
            // Create the initial context
            DirContext ctx = new InitialDirContext(env);

    // Create the reference containing the (future) location of the object
            Reference ref = new Reference("Hello",
                    new StringRefAddr("URL", rmiurl));

            BasicAttributes battr = new BasicAttributes(
                    "javaCodebase",
                    "http://www.cs.ubbcluj.ro/~cori/t/");

            // Bind the object to the directory
            ctx.rebind("cn=RefHello", ref);

            ctx.close();
        } catch (NamingException e) {
            System.out.println("Operation failed: " + e);
        }
```

Program 4. HelloServ.java

(3) We finally invoke the remote object from a client resident on machine `nessie.cs.ubbcluj.ro`. As we proposed from the beginning, the client makes the remote invocation without knowing the server object address, which allows the latter to change its location, without affecting the potential clients.

```
import java.util.Hashtable;
import javax.naming.*;
import javax.naming.directory.*;
import java.rmi.*;
public class HelloCl {
public static void main(String argv[]) {
    String rmiurl = "rmi://cronos.cc.ubbcluj.ro/hello";

    // Set up environment for creating the initial context
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
```

```
                "com.sun.jndi.ldap.LdapCtxFactory");
        env.put(Context.PROVIDER_URL,
                "ldap://rave.scs.ubbcluj.ro:389/cn=CORI,
                o=CCUBB,c=RO");

        try {
            DirContext ctx = new InitialDirContext(env);

            // lookup the object
            Hello h = (Hello)ctx.lookup("cn=RemoteHello");
            System.out.println(h.sayHello());

            ctx.close();
        } catch (NamingException e) {
            System.out.println("Operation failed: " + e);
        } catch (RemoteException e1) {
            System.out.println("Operation failed: " + e1);
        }
    }
}
```

Program 5. HelloCl.java

The method that we presented uses the information stored in the directory. This information, represented by the `Reference` object, is actually a pointer to the information stored in another naming service (the RMI registry), which in turn, contains the reference to the `java.rmi.Remote` object.

Even if, in the simple example presented above, this level of indirection seems to be overheading, besides location transparency, it has important applications like providing fault-tolerance to distributed Java objects.

We use fault-tolerance to refer to the situation when a server object isn't available anymore (it was stopped or its host crashed), its services are being provided by other identical backup server objects. This is the traditional method of providing fault-tolerance by replication of the services that require high availability. In our case, a fault-tolerant Java server object is registered with different `rmiregistries`, and the corresponding rmi object's identifying URLs are stored as a `Reference` in a LDAP directory.

A client invocation uses one of the available server objects (in fact, the first available server in the stored addresses references order), without being aware of the duplication. The management of the duplicated objects is done totally transparent for the potential clients, and is completed by replication service provided by LDAP (for example `slapd` – Stand-alone LDAP Daemon – can be configured to

provide replicated service for a database with the help of `slurpd`, the standalone LDAP update replication daemon) [12].

5. Conclusions

In the article we presented the standard RMI mechanism available on the Java platforms, and some possible extensions to its basic features.

Within the Java language domain, Java RMI offers powerful new features for remote object distribution. Besides the powerful objects interaction facilities this mechanism provides, it can be extended with features that respects new constraints and requirements like location and migration transparency of the server objects. Also, the basic distributed systems requirement of fault-tolerance can be successfully integrated into the Java RMI mechanism.

References

[1] **Directory Examples**
http://java.sun.com/products/jndi/tutorial/getStarted/examples/directory.html
http://java.sun.com/products/jndi/tutorial/objects/storing/src/RemoteObj.java

[2] **Filterfresh: Hot Replication of Java RMI Server Objects**
http://www.usenix.org/publications/library/proceedings/coots98/full_papers/baratloo/baratloo_html/baratloo.html

[3] **Java IDL**
http://sophia.dtp.fmph.uniba.sk/javastuff/tutorial/idl/summary/

[4] **JNDI API**
http://sunsite.ccu.edu.tw/java/jdk1.3/api/javax/naming/InitialContext.html# ENVIRONMENT
http://java.sun.com/products/jndi/

[5] **JNDI Tutorial**
http://java.sun.com/products/jndi/tutorial/

[6] **LDAP: A Next Generation Directory Protocol**
http://www.intranetjournal.com/foundation/ldap.shtml

[7] **LDAP and JNDI: Together forever**
http://www.javaworld.com/javaworld/jw-03-2000/jw-0324-ldap_p.html

[8] **RFC LDAP**
http://www.ietf.org/rfc/rfc2713.txt

[9] **RMI and Java Distributed Computing**
http://java.sun.com/features/1997/nov/rmi.html

[10] **RMI Registry Service Provider JNDI**
http://sunsite.ccu.edu.tw/java/jdk1.3/guide/jndi/jndi-rmi.html#USAGE

[11] **SLAPD Daemon**
http://www.umich.edu/ dirsvcs/ldap/doc/guides/slapd/1.html#RTFToC1

[12] **SPI**
http://java.sun.com/j2se/1.3/docs/guide/jndi/spec/spi/jndispi.fm.html#1005286
Service Provider Package
http://java.sun.com/products/jndi/tutorial/getStarted/overview/provider.html

Department of Computer Science, Faculty of Mathematics and Computer Science, "Babeş-Bolyai" University, Cluj-Napoca, Romania
E-mail address: florin|cori@cs.ubbcluj.ro