

RMI

PATRIK FUHRER

CONTENTS

1. Introduction	1
2. Why distribute objects?	2
3. Features of Distributed Object Systems	3
3.1. Object Interface Specification	4
3.2. Object Manager	5
3.3. Registration / Naming Service	5
3.4. Object Communication Protocol	5
3.5. Miscellaneous	5
4. Java RMI	6
4.1. Remote Object Interfaces	6
4.2. Server Implementations	6
4.3. The RMI Registry	6
4.4. Client Stubs and Server Skeletons	7
4.5. Registering and Using a Remote Object	7
4.6. Serializing Objects	8
4.7. Summary	9
5. A simple RMI-based application	9
5.1. Defining a remote interface	9
5.2. Implementing the remote interface and the server	9
5.3. Developing a client that uses the remote interface	10
5.4. Generating stubs and skeletons	10
5.5. Starting the RMI registry	11
5.6. Running the server and the client	11
6. Building a Generic Compute Engine	11
6.1. The big picture	11
6.2. Defining a remote interface	12
6.3. Implementing the remote interface and the server	12
7. RMI versus CORBA	12
8. Approximation of π	13
9. A brief history of Java language	13
References	15

1. INTRODUCTION

For the past decade, "distributed computing" has been one of the biggest buzz phrases in the computer industry. At this point in the information age, we know

how to build networks; we use thousands of engineering workstations and personal computers to do our work, instead of huge behemoths in glass-walled rooms. Surely we ought to be able to use our networks of smaller computers to work together on larger tasks. And we do - an act as simple as reading a web page requires the cooperation of two computers (a client and a server) plus other computers that make sure the data gets from one location to the other. However, simple browsing (i.e., a largely one-way data exchange) isn't what we usually mean when we talk about distributed computing. We usually mean something where there's more interaction between the systems involved.

You can think about distributed computing in terms of breaking down an application into individual computing *agents* that can be distributed on a network of computers, yet still work together to do cooperative tasks. The motivations for distributing an application this way are many. Here are a few of the more common ones:

- Computing things in parallel by breaking a problem into smaller pieces enables you to solve larger problems without resorting to larger computers. Instead, you can use smaller, cheaper, easier-to-find computers.
- Large data sets are typically difficult to relocate, or easier to control and administer located where they are, so users have to rely on remote data servers to provide needed information.
- Redundant processing agents on multiple networked computers can be used by systems that need fault tolerance. If a machine or agent process goes down, the job can still carry on.

Assorted tools and standards for assembling distributed computing applications have been developed over the years. These started as low-level data transmission APIs and protocols, such as RPC and DCE, and have recently begun to evolve into object-based distribution schemes, such as CORBA, RMI, and OpenDoc. These programming tools essentially provide a protocol for transmitting structured data (and, in some cases, actual runnable code) over a network connection. Java offers a language and an environment that encompass various levels of distributed computing development, from low-level network communication to distributed objects and agents, while also having built-in support for secure applications, multiple threads of control, and integration with other Internet-based protocols and services.

2. WHY DISTRIBUTE OBJECTS?

Distributed objects are a potentially powerful tool that has only become broadly available for developers at large in the past few years. The power of distributing objects is not in the fact that a bunch of objects are scattered across the network. The power lies in that any agent in your system can directly interact with an object that "lives" on a remote host. Distributed objects, if they're done right, really give you a tool for opening up your distributed system's resources across the board. And with a good distributed object scheme you can do this as precisely or as broadly as you'd like.

Partitioning and distributing data and functions is one of the main issues that you face when developing distributed systems. One of the optimal data/function partitioning capabilities that you'd like to have available when developing distributed applications is being able to distribute data/function "modules" freely

and transparently, and have these modules be defined based on application structure rather than network distribution influences. Distributed object systems try to address these issues by letting developers take their programming objects and have them "run" on a remote host rather than a local host. The goal of most distributed object systems is to let any object reside anywhere on the network, and allow an application to interact with these objects exactly the same way as they do with a local object. Additional features found in some distributed object schemes are the ability to construct an object on one host and transmit it to another host, and the ability for an agent on one host to create a new object on another host.

The value of distributed objects is more obvious in larger, more complicated applications than in smaller, simpler ones. That's because much of the trade-off between distributed objects and other techniques, like message passing, is between simplicity and robustness. In a smaller application with just a few object types and critical operations, it's not difficult to put together a catalog of simple messages that would let remote agents perform all of their critical operation through on-line transactions. With a larger application, this catalog of messages gets complicated and difficult to maintain. It's also more difficult to extend a large message-passing system if new objects and operations are added. So being able to distribute the objects in our system directly saves us a lot of design overhead, and makes a large distributed system easier to maintain in the long run.

3. FEATURES OF DISTRIBUTED OBJECT SYSTEMS

In this section we'll discuss some of the features that distributed object systems need. These features are illustrated in Figure 1. An object interface specification is used to generate a server *implementation* of a class of objects, an interface between the object implementation and the object manager, sometimes called an object *skeleton*, and a client interface for the class of objects, sometimes called an object *stub*. The skeleton will be used by the server to create new instances of the class of objects and to route remote method calls to the object implementation. The stub will be used by the client to route transactions (method invocations only) to the object on the server. On the server side, the class implementation is passed through a *registration service*, which registers the new class with a *naming service* and an *object manager*, and then stores the class in the server's storage for object skeletons.

With an object fully registered with a server, the client can now request an instance of the class through the naming service. The runtime transactions involved in requesting and using a remote object are shown in Figure 2. The naming service routes the client's request to the server's object manager, which creates and initializes the new object using the stored object skeleton. The new object is stored in the server's object storage area, and an object handle is issued back to the client in the form of an object stub interface. This stub is used by the client to interact with remote object.

While Figure 2 illustrates a client-server remote object environment, a remote object scheme can typically be used in a peer-to-peer manner as well. Any agent in the system can act as both a server and a client of remote objects, with each maintaining its own object manager, object skeleton storage, and object instance storage.

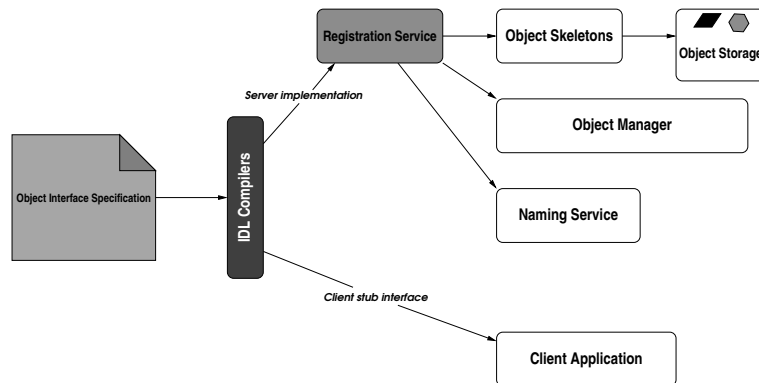


FIGURE 1. General architecture for distributed object systems (cf. [1])

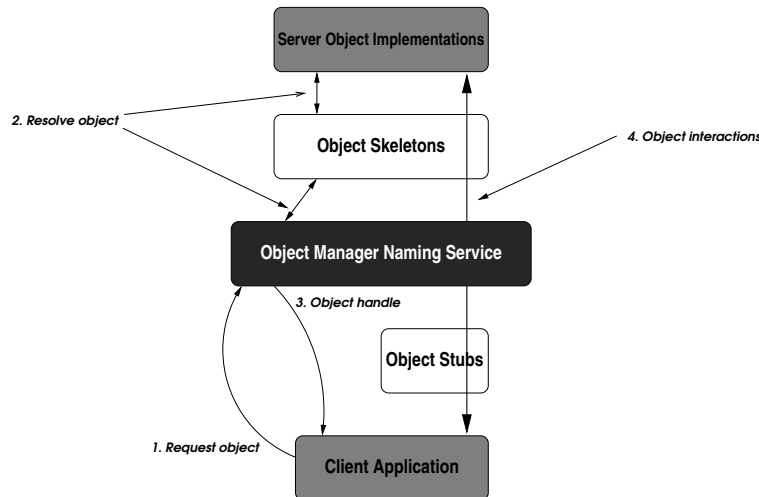


FIGURE 2. Remote object transactions at runtime (cf. [1])

3.1. Object Interface Specification. To provide a truly open system for distributing objects, the distributed object system should allow the client to access objects regardless of their implementation details, like hardware platform and software language. It should also allow the object server to implement an object in whatever way it needs to. Although in this paper we are talking about implementing systems in Java, you may have valuable services already implemented in C, C++, Smalltalk, and these services might be expensive to reimplement in Java. In this situation you'd like the option of wrapping your existing services with object interfaces and using them directly via the remote object system.

Some distributed object systems provide platform-independent means for specifying object interfaces. These object interface descriptions can be converted into server skeletons, which can be compiled and implemented in whatever form the server requires. The same object interfaces can be used to generate client-side stub interfaces. If we're dealing with Java-based distributed system, the server skeletons

and client stubs will be generated as Java class definitions, which will be compiled into bytecodes.

In CORBA, object interfaces are described using a platform-independent language called the Interface Definition Language (IDL). A similar language is the Component Object Model language (COM) used in Microsoft's DCOM system.

3.2. Object Manager. The object manager is really at the heart of the distributed object system, since it manages the object skeletons and object references on an object server. The object manager plays a role similar to that of an Object Request Broker (ORB) in a CORBA system, or the registry in RMI, which will be discussed in more detail. When a client asks for a new object, the object manager locates the skeleton for the class of object requested, creates a new objects based on the skeleton, stores the new object in object storage, and sends a reference to the object back to client. Remote method calls made by the client are routed through the manager to the proper object on the server, and the manager also routes the results back to the client. Finally, when the client is through with the remote object, it can issue a request to the object manager to destroy the object. The manager removes the object from the server's storage and frees up any resources the object is using.

3.3. Registration / Naming Service. The registration/naming service acts as an intermediary between the object client and the object manager. Once we have defined an interface to an object, an implementation of the interface needs to be registered with the service so that it can be addressed by clients. In order to create and use an object from a remote host, a client needs a naming service so that it can specify things like the type of object it needs, or the name of a particular object if one already exists on the server. The naming service routes these requests to the proper object server. Once the client has an object reference, the naming service might also be used to route method invocations to their targets.

3.4. Object Communication Protocol. In order for the client to interact with the remote object, a general protocol for handling remote object requests is needed. This protocol needs to support, at a minimum, a means for transmitting and receiving object references, and data in the form of objects or basic data types. Ideally we don't want the client application to need to know any details about this protocol. It should simply interact with local object interfaces, letting the object distribution scheme take care of communicating with the remote object behind the scenes. This minimizes the impact on the client application source code, and helps you to be flexible about how clients access remote services.

3.5. Miscellaneous. Security is a need for any network interaction, and encryption methods, authentication and authorization features should be supported directly by the object distribution scheme.

We'll also need a lot of **development tools** like object interface editors and project managers, language cross-compilers, and symbolic debuggers. And these tools should provide a reasonable method to monitor and diagnose object systems spread across the network.

4. JAVA RMI

While there are several distributed object schemes that can be used within the Java environment, we'll only cover two that qualify as serious options for developing your distributed applications: CORBA and RMI. Both of them have their advantages and their limitations. We'll look in detail at Java RMI in this section. CORBA won't be studied in detail but there are many similarities between the two technologies. However, in the following section we'll shortly compare RMI and CORBA and list some critical differences between them.

The Java Remote Method Invocation (RMI) package is a Java-centric scheme for distributed objects that is now part of the core Java API. RMI offers some of the critical elements of a distributed object system for Java, plus some other features that are made possible by the fact that RMI is a Java-only system.

4.1. Remote Object Interfaces. Since RMI is a Java-only distributed object scheme, all object interfaces are written in Java. Client stubs and server skeletons are generated from this interface. First, the interface for the remote object has to be written as extending the `java.rmi.Remote` interface. The `Remote` interface doesn't introduce any methods to the object's interface; it just serves to mark remote objects for the RMI system. Also, all methods in the interface must be declared as throwing the `java.rmi.RemoteException`. The `RemoteException` is the base class for many of the exceptions that RMI defines for remote operations, and the RMI engineers decided to expose the exception model in the interfaces of all RMI remote objects. This is one of the drawbacks of RMI: it requires you to alter an existing interface in order to apply it to a distributed environment.

4.2. Server Implementations. Once the remote object's Java interface is defined, a server implementation of the interface can be written. In addition to implementing the object's interface, the server also typically extends the `java.rmi.server.UnicastRemoteObject` class. `UnicastRemoteObject` is an extension of the `RemoteServer` class, which acts as a base class for server implementations of objects in RMI.

4.3. The RMI Registry. In RMI, the registry serves the role of the Object Manager and Naming Service for the distributed object system. The registry runs in its own Java runtime environment on the host that's serving objects. The RMI registry is only required to be running on the server of a remote object. Clients of the object use classes in the RMI package to communicate with the remote registry to look up objects on the server. You start an RMI registry on a host by running the `rmiregistry` command, which is included in the standard JDK distribution. By default the registry listens to port 1099 on the local host for connections, but you can specify any port for the registry process by using a command-line option:

```
objhost% rmiregistry 4001
```

Once the registry is running on the server, you can register object implementations by name, using the `java.rmi.Naming` interface. We'll see the details of registering server implementations in paragraph 4.5. A registered class on a host can then be located by a client by using the `lookup()` method on the `Naming` interface. You address remote objects using URL-like scheme. For example

```
MyObject obj1 =  
(MyObject)Naming.lookup("rmi://iiufsun00.unifr.ch/Object1");
```

will look up an object registered on the host `iiuvsun00.unifr.ch` under the name `Object1`.

RMI also provides mechanisms to remove an object from the registry and to rebind an object to the registry.

4.4. Client Stubs and Server Skeletons. Once you've defined your object's interface and derived a server implementation for the object, you can create a client stub and server skeleton for your object. First the interface and the server implementation are compiled into bytecodes using the `javac` compiler, just like normal classes. Once we have bytecodes for the interface and the server implementation, we have to generate the linkage from the client through the RMI registry to the object implementation we just generated. This is done using the RMI stub compiler, `rmic`. Suppose, we've defined a remote interface called `MyObject`, and we've written a server implementation called `MyObjectImpl`, and compiled both of these into bytecodes. We would generate the RMI stub and skeleton for the class with the `rmic` compiler:

```
myhost% rmic MyObject
```

The `rmic` compiler bootstraps off of the Java bytecodes for the object interface and implementation to generate a client stub and a server skeleton for the class. A client stub is returned to a client when a remote instance of the class is requested through the `Naming` interface. The stub has hooks into the object serialization subsystem in RMI for marshaling method parameters.

The server skeleton acts as an interface between the RMI registry and instances of the object implementation residing on a host. When a client request for a method invocation on an object is received, the skeleton is called on to extract the serialized parameters and pass them to the object implementation (cf. Figure 3).

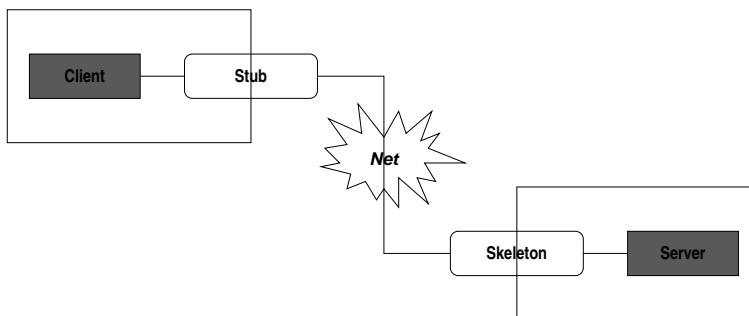


FIGURE 3. Stubs and skeletons (cf. [2])

4.5. Registering and Using a Remote Object. Now we have compiled interface and implementation for our remote object, and we've created the client stub and server skeleton using the `rmic` compiler. The final hurdle is to register an instance of our implementation on a remote server, and then look up the object on a client. Since RMI is a Java-centric API, we can rely on the bytecodes for the interface, the implementation, the `rmic`-generated stub, and skeleton being loaded automatically over the network into the Java runtimes at the clients. A server process has to

register an instance of the implementation with a RMI registry running on the server:

```
System.setSecurityManager(new java.rmi.RMISecurityManager());
MyObjectImpl obj = new MyObjectImpl();
Naming.rebind("Object1",obj);
```

Once this is done, a client can get a reference to the remote object by connecting to the remote registry and asking for the object by name (cf. Figure 4):

```
MyObject obj1 =
(MyObject)Naming.lookup("rmi://iiufsun00.unifr.ch/Object1");
```

We created an instance of the `RMISecurityManager` and installed it with the `System` object. The purpose of the method is to protect the host from malicious code from the client. If you do not want to use the default security manager `RMISecurityManager` you can define and customize your own.

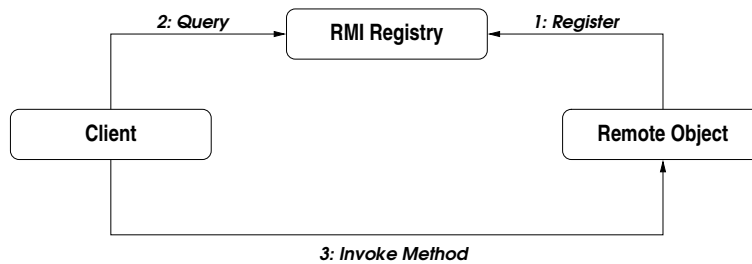


FIGURE 4. The RMI registry manages RMI references (cf. [2])

4.6. Serializing Objects. Another Java facility that support RMI is object serialization. Java's serialization system provides a way for you to transfer or request an object instance by value from one remote process to another. The `java.io` package includes classes that can convert an object into a stream of bytes and reassemble the bytes back into an identical copy of the original object. Using these classes, an object in one process can be serialized and transmitted over a network connection to another process on a remote host. The object (or at least a copy of it) can then be reassembled on the remote host. An object that you want to serialize has to implement the `java.io.Serializable` interface. With this done, an object can be written just as easily to a file, a string buffer, or a network socket. for example, assuming that `Foo` is a class that implements `Serializable`, the following code writes `Foo` on an object output stream, which sends it to the underlying I/O stream:

```
Foo myFoo=new Foo();
OutputStream out=... // Create output stream to object destination
ObjectOutputStream oOut=new ObjectOutputStream(out);
oOut.writeObject(myFoo);
```

The object can be reconstructed just as easily:

```
InputStream in=... // Create input stream from source of object
ObjectInputStream oIn=new ObjectInputStream(in);
Foo myFoo=(Foo)oIn.readObject();
```


We've simplified things a bit by ignoring any exceptions generated by these code snippets. The object serialization facility allows an actual object to be serialized in its entirety, transmitted to any destination, and then reconstructed as a precise replica of the original. When you serialize an object, all of the objects that it references as data members will also be serialized, and all of their object references will be serialized, and so on. If you attempt to serialize an object that doesn't implement `Serializable` interface, or an object that refers to non-serializable objects, then a `NotSerializableException` will be thrown. Method arguments that aren't objects are serialized automatically using their standard byte stream formats. In RMI, the serialization facility is used to marshal and unmarshal method arguments that are objects, **but that are not remote objects**. Any object argument to a method on a remote object in RMI must implement the `Serializable` interface, since the argument will be serialized and transmitted to the remote host during the remote method invocation.

4.7. Summary.

- Developing distributed applications with RMI is far easier than programming using sockets. When you're programming with RMI, there is no need to design an application-level protocol. RMI takes care of all the nitty gritty communication details.
- The task of creating an RMI-based application invokes six simple steps:
 - Defining a remote interface
 - Implementing the remote interface and the server
 - Developing a client that uses the remote interface
 - Generating stubs and skeletons
 - Starting the naming service, the RMI registry
 - Running the server and the client

5. A SIMPLE RMI-BASED APPLICATION

In this section we develop a simple RMI-based application and we go concretely through the six simple steps seen in paragraph 4.7.

All the files used in this example can be downloaded from:

`http://www-iiuf.unifr.ch/courses01-02/genie_log/exercices.htm`

5.1. Defining a remote interface. We want to create a simple server that add two numbers passed as arguments. The remote interface is thus very simple. You can check that it respects all the rules that are explained in paragraph 4.1.

```
public interface ArithInterface extends java.rmi.Remote {
    int add(int a, int b) throws java.rmi.RemoteException;
}
```

5.2. Implementing the remote interface and the server. Here we implement the interface defined in the previous paragraph and we extend the `UnicastRemoteObject` as shown in paragraph 4.2.

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class ArithImpl extends UnicastRemoteObject
implements ArithInterface {
```

```

    public ArithImpl() throws RemoteException {}

    public int add(int a, int b) {
        return a+b;
    }
}

```

And now we can implement the server using the mechanism explained in paragraph 4.5.

```

import java.rmi.*;

public class ArithServerMain {

    public static void main(String argv[]) {
        System.setSecurityManager(new RMISecurityManager());
        try {
            ArithImpl obj = new ArithImpl();
            Naming.rebind("rmi://myhostname/A", obj);
            System.out.println("AdderServer bound in registry");
        } catch (Exception e) {
            System.out.println("AdderImplementation error: "+ e.getMessage());
            e.printStackTrace();
        }
    }
}

```

5.3. Developing a client that uses the remote interface.

```

import java.rmi.*;

public class ArithClient {
    public static void main(String argv[]) {
        int result=0;
        try {
            ArithInterface obj = (ArithInterface)Naming.lookup("rmi://myhostname/A");
            result = obj.add(7,12);
        } catch (Exception e) {
            System.out.println("AdderClient: "+e.getMessage());
            e.printStackTrace(); }
        System.out.println("The sum = "+result);
    }
}

```

5.4. Generating stubs and skeletons. According to paragraph 4.4 we now have to compile the interface and the server implementation into bytecodes using the javac compiler. In our example we have to execute:

```
javac Arith.java ArithImpl.java
```

Now, we generate the RMI stub for our class executing the command:

```
rmic -v1.2 AdderImplementation
```

This generates two class files names `AdderImplementation_Stub.class`.

5.5. Starting the RMI registry. This is simply done by executing the command: `rmiregistry` on the machine you want to launch the server with.

5.6. Running the server and the client.

- Launch an HTTP server from which one can download the stub file,
`ArithImpl_Stub.class`
.
- Running our server on a Windows machine you probably will have to execute the following command:

```
java -Djava.security.policy=java.policy  
-Djava.rmi.server.codebase=http://yourhost:yourport/  
ArithServerMain
```

Note that the `java.security.policy` property is used to specify the policy file that contains the permissions you intend to grant specific code bases.

- Running the client, is simply done by executing the command:

```
java ArithClient iauf20
```

where `iauf20` is the name of the machine running the server and 7 and 4 are the numbers we want to add.

6. BUILDING A GENERIC COMPUTE ENGINE

6.1. The big picture. This example focuses on a simple yet powerful distributed application called a compute engine. The compute engine, a remote object in the server, takes tasks from clients, runs them, and returns any results. The tasks are run on the machine where the server is running. This sort of distributed application could allow a number of client machines to make use of a particularly powerful machine or one that has specialized hardware.

The novel aspect of the compute engine is that the tasks it runs do not need to be defined when the compute engine is written. New kinds of tasks can be created at any time and then given to the compute engine to be run. All that is required of a task is that its class implements a particular interface. Such a task can be submitted to the compute engine and run, even if the class that defines that task was written long after the compute engine was written and started. The code needed to accomplish the task can be downloaded by the RMI system to the compute engine, and then the engine runs the task, using the resources on the machine on which the compute engine is running.

The ability to perform arbitrary tasks is enabled by the dynamic nature of the Java platform, which is extended to the network by RMI. RMI dynamically loads the task code into the compute engine's Java virtual machine and runs the task without prior knowledge of the class that implements the task. An application like this, which has the ability to download code dynamically, is often called a behavior-based application. Such applications usually require full agent-enabled infrastructures. With RMI such applications are part of the basic mechanisms for distributed computing on the Java platform.

6.2. Defining a remote interface. The `Task` interface defines a single method, `execute`, which returns an `Object`, has no parameters, and throws no exceptions. Since the interface does not extend `Remote`, the method in this interface doesn't need to list `java.rmi.RemoteException` in its `throws` clause.

The return value for the `Compute`'s `executeTask` and `Task`'s `execute` methods is declared to be of type `Object`. This means that any task that wants to return a value of one of the primitive types, such as an `int` or a `float`, needs to create an instance of the equivalent wrapper class for that type, such as an `Integer` or a `Float`, and return that object instead.

Note that the `Task` interface extends the `java.io.Serializable` interface. RMI uses the object serialization mechanism to transport objects by value between Java virtual machines. Implementing `Serializable` marks the class as being capable of conversion into a self-describing byte stream that can be used to reconstruct an exact copy of the serialized object when the object is read back from the stream.

Different kinds of tasks can be run by a `Compute` object as long as they are implementations of the `Task` type. The classes that implement this interface can contain any data needed for the computation of the task and any other methods needed for the computation.

6.3. Implementing the remote interface and the server. Here is how RMI makes this simple compute engine possible. Since RMI can assume that the `Task` objects are written in the Java programming language, implementations of the `Task` object that were previously unknown to the compute engine are downloaded by RMI into the compute engine's virtual machine as needed. This allows clients of the compute engine to define new kinds of tasks to be run on the server machine without needing the code to be explicitly installed on that machine. In addition, because the `executeTask` method returns a `java.lang.Object`, any type of object can be passed as a return value in the remote call.

The compute engine, implemented by the `ComputeEngine` class, implements the `Compute` interface, allowing different tasks to be submitted to it by calls to its `executeTask` method. These tasks are run using the task's implementation of the `execute` method. The compute engine reports results to the caller through its return value: an `Object`.

If you read the source code of the `ComputeEngine` class you'll see all the steps we treated in section 4.

7. RMI VERSUS CORBA

RMI is a Java-centric distributed object system. CORBA, on the other hand, is designed to be language-independent. Object interfaces are specified in a language that is independent of the actual implementation language. This interface description can then be compiled into whatever implementation language suits the job and the environment.

This distinction is really the heart of the split between the two technologies. In addition to this core distinction between CORBA and RMI, there are other differences to keep in mind:

- Relatively speaking, RMI can be easier to master, especially for experienced Java programmers, than CORBA. CORBA is a rich, extensive family of

standards and interfaces, and delving into the details of these interfaces is sometimes overkill for the task at hand.

- CORBA is more mature standard than RMI, and has had time to gain richer implementations. The CORBA standard is a fairly comprehensive one in term of distributed objects, and there are CORBA implementations out there that provide many more services and distribution options than RMI or Java.
- Various low-level technical details about the two schemes may be seen as advantages or disadvantages to you as a developer: the fact that RMI imposes additional `throws` clauses in your remote interface is one that we've already mentioned, and CORBA's peer-to-peer ORB communication model as opposed to RMI's server-centric model may be another.
- Other differences are :
 - RMI objects are garbage-collected, but CORBA objects are not.
 - Unlike CORBA, RMI does not support out and inout parameters since local objects are passed by copy and remote objects are passed by reference to a stub.

Finally, which is better, CORBA or RMI? Basically, it depends. If you're looking at a system that you're building from scratch, with no hooks to legacy systems and fairly mainstream requirements in terms of performance and other language features, then RMI may be the most effective and efficient tool for you to use. On the other hand, if you're linking your distributed system to legacy services implemented in other languages, or if there is the possibility that subsystems of your application will need to migrate to other languages in the future, or if your system depends strongly on services that are available in CORBA and not in RMI, or if critical subsystems have highly-specialized requirements that Java can't meet, then CORBA may be your best bet.

8. APPROXIMATION OF π

In the class `Pi` we have to compute the value of π to the specified number of digits after the decimal point.

The value is computed using Machin's formula:

$$\frac{\pi}{4} = 4 * \arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right)$$

and a power series expansion of $\arctan(x)$ to sufficient precision. This is done in the method `computePi`.

The value of $\arctan(x)$ is computed in the method `arctan` which computes the value, in radians, of the arctangent of the inverse of the supplied integer to the specified number of digits after the decimal point. The value is computed using the power series expansion for the arc tangent:

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} \dots$$

9. A BRIEF HISTORY OF JAVA LANGUAGE

The development of the Java language began as a response to a letter written in 1990 to the CEO of Sun Microsystems. Patrick Naughton, a disgruntled Sun software engineer, detailed in his letter the many reasons he was leaving Sun to work

for NeXT; chief among these complaints was the confusion and frustration he and his co-workers felt when programming which stemmed from the huge assortment of application programming interfaces (API's) which Sun supported. Shaken by Naughton's keen assessment of the problems their software division faced, Sun's management commissioned Naughton, Bill Joy, James Gosling, and three others to form a research group to create something new and exciting which alleviated this problem.

The team, codenamed Green, set for themselves the goal of creating a single operating environment, both processor and the software to run upon it, which could be used by all consumer electronic devices, from computers and video game machines all the way down to remote controls and VCR's. Their vision was to enable interactivity between all such devices, as well as to speed development and reduce the cost of implementing new features through the use of a single, small core operating environment. After much research and testing, the result was a simple object-oriented programming language named Oak, after the tree which grew out the window of Gosling's office (although other stories abound).

After many months of testing and tweaking Oak, the designers developed a small hand-held device which featured an easy-to-use, graphic-intensive and appealing interface called "7". The user's guide and helper for this device was an animated figure named Duke, who later became Java's official mascot. At this point, Sun made Green its own company and changed its name to First Person. The company began searching for a market for its unique product; after deals with companies such as 3DO and Time-Warner fell through in the interactive television area, the outlook for the fledgling company seemed bleak.

Then in 1993 the National Center for Supercomputing Applications (NCSA) released Mosaic, an application which allowed users to easily and graphically access portions of the previously arcane, text-based Internet. Within a year, this visually-based, readily accessible network of Internet sites known as the World Wide Web had grown from a mere research project into a revolutionary medium to transmit ideas and data. The number of web-sites, or areas accessible by a web-browser such as Mosaic and its descendants (most notable among these being Netscape), was growing at a phenomenal rate.

In 1994, the First Person team decided to adjust its design and began developing a "language based operating system" to enable online multimedia software. In a most unusual, but increasingly common and Internet-savvy, move the company also decided to give the source form of the Oak language away over the Internet. Renamed Java by Sun, this new language enabled users to produce programs named applets which could be transmitted and run over the Internet, a somewhat different end product from what was originally envisioned back at the beginning of the decade. Although Java can produce stand-alone applications, it is its Internet-spanning ability that has brought it to the forefront of the modern computing stage. The final piece to the puzzle, somehow enabling browsers to execute these new Java applets, was provided in 1995 when Sun produced its HotJava web browser and allowed Netscape to support the Java language as well.

JavaSoft, the current name of the company which oversees the development of the Java language, was founded in January of 1996, and a few months later released the Java Development Kit (JDK) version 1.0. The newest version, version 1.1, is scheduled to be released sometimes soon in the first or second quarter of 1997. The

result of the entire project is the Java programming language: a small-instruction set, object-oriented, interpreted language. Actually, users can compile Java code directly into stand-alone executable applications, but its real power lies in its ability to produce machine-independent bytecodes, low-level instructions which can be placed on a server and transferred over networks to client machines. Once transferred, these bytecodes are allotted a secure space on the client machine through the web-browser in which to run; the Java Virtual Machine (the browser's Java run-time environment) translates these architecture-neutral bytecodes into machine-specific code, which is then tested and executed. This form of network distributed program is called a Java applet, and this is the weapon with which Sun is hoping to revolutionize the computing and information-processing world.

REFERENCES

1. James Farley, *Java distributed computing*, O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1998.
2. Qusay H. Mahmoud, *Distributed programming with Java*, Manning Publications, Greenwich, CT, USA, October 1998.

E-mail address: `patrik.fuhrer@unifr.ch`