Distributed Objects

Table of Contents

[Distributed Object Systems 3](#_Toc100596779)

[Implementations 4](#_Toc100596780)

[Java RMI 4](#_Toc100596781)

[CORBA 4](#_Toc100596782)

[DCOM 4](#_Toc100596783)

[SOAP 5](#_Toc100596784)

[REST 5](#_Toc100596785)

[Terminologies 6](#_Toc100596786)

[Stubs and Skeletons 6](#_Toc100596787)

[Marshalling and Unmarshalling 6](#_Toc100596788)

[Java RMI 7](#_Toc100596789)

[CORBA 13](#_Toc100596790)

In object-oriented programming, we saw how we can create multiple objects from different classes and allow objects from one class to invoke methods belonging to objects of another class. However, in those cases, all the code was in the same machine. What if we have a situation where a **client-side machine** has a class which wishes to invoke a method belonging to a class that is on a **server-side machine**? This is where **distributed objects** come in.

The basic structure follows the **client-server** architecture. The client-side machine has a **stub**, which is a representation of the server-side code. This stub is used to send requests to a **skeleton** on the server-side, which processes the request and invokes the required methods. On the flip side, the response is sent from the skeleton back to the stub. This process is called **Remote Method Invocation** (RMI).



This entire process is transparent to the client. To them, it is as though they are calling a local method.

## Distributed Object Systems

**Distributed Object Systems** like this one extend the concept of object-oriented programming to distributed systems, allowing objects to be distributed across multiple **nodes**. Local processes can operate on objects stored in other nodes as though they were stored locally. This allows interoperability between **heterogenous systems**, i.e. different programming languages or different operating systems can be used on the client and server side machines.

The benefits of distributed object systems are:

* **Modularity** – The client and server-side code are completely separated and modularized, which means we can easily replace a module on one end without affecting the operation of the other end.
* **Reusability** – Due to the modularity present, a specific module can easily be reused in another project depending on its needs.
* **Extensibility** – Adding new features is as easy as adding a new module to the existing code. If we add something to the server side, the client can begin using it without us having to modify things on the client-side as well.
* **Interoperability** – As mentioned before, we can work with heterogenous environments.

## Implementations

### Java RMI

**Java RMI** is the Java variant of the remote method invocation (RMI) process. It relies on a protocol called the **Java Remote Method Protocol**. The protocol requires a Java Virtual Machine (JVM) to implement, which means that both the client and server-side code must be written in Java. However, due to using a JVM, it is portable across different operating systems.

### CORBA

The **Common Object Request Broker Architecture** (CORBA) is not really a protocol, but rather a **specification**. It uses the **Internet Inter-ORB Protocol** (IIOP), which is both platform and language independent. This makes CORBA well suited for complex, heterogenous systems.

### DCOM

The **Distributed Component Object Model** (DCOM) uses a protocol called the **Object Remote Procedure Call**. This protocol is language independent, but depends on the Component Object Model (COM) platform, which only works on Windows machines.

### SOAP

The **Simple Object Access Protocol** (SOAP) is an XML-based message protocol that is both platform and language independent. It allows distributed object systems to be implemented via Web Services, with the details of the object type and methods encoded in an XML file.

### REST

**Representational State Transfer** (REST) is an architectural style. Web services that conform to this style are said to be **RESTful Web Services**. They provide interoperability between computer systems on the internet. In RESTful implementations, all kinds of information (e.g. data records, objects, etc.) are considered to be resources. Requesting systems can manipulate textual representations of these resources via uniform and predefined stateless operations.

## Terminologies

### Stubs and Skeletons

A **stub** is a class that implements a remote interface on the client-side. A **skeleton** is the stub’s counterpart on the server-side. The stub and the skeleton communicate via a network. The stub does not actually know how the methods of the server-side work, it just knows which ones there are. When the client makes a request, the request is made via the stub, which forwards it to the skeleton. The skeleton actually has access to the different classes and methods, since it works on the server-side. Thus, it is able to process the request as required and send back a response.

### Marshalling and Unmarshalling

Before the stub can forward the request to the skeleton, it has to process the arguments and convert them to a stream of bytes. This process is called **marshalling**. The skeleton in turn must convert the stream of bytes back into the arguments, a process called **unmarshalling**. When sending the response, the roles are reversed, with the skeleton marshalling the response and its values and the stub unmarshalling them.

## Java RMI

We will be studying the Java implementation of RMI thoroughly. RMI is a core Java API and class library which can be used to call methods in objects running on one virtual machine from objects running on another virtual machine, even if the two are on physically separate devices.

The remote objects live on the server. Each of these implements one or more **interfaces** that specify which of its methods can be invoked by clients. This involves several steps:

1. Develop an **interface** that extends java.rmi.*Remote*. This is where we declare the **methods** that will be made available to clients. This interface is the **stub**.

*// CalculatorInterface.java*import java.rmi.*Remote*;  
import java.rmi.RemoteException;  
  
public interface *CalculatorInterface* extends *Remote* {  
 public int add(int a, int b) throws RemoteException;  
 public int sub(int a, int b) throws RemoteException;  
 public int mul(int a, int b) throws RemoteException;  
 public int div(int a, int b) throws RemoteException;  
}

JAVA

1. Develop a **class** that implements the interface. This class is known as the **servant class** and it extends java.rmi.server.UnicastRemoteObject. Since this is an implementation of the interface, this is where we declare the **method bodies**. This class is also responsible for **marshalling** and **unmarshalling**. It is the **skeleton**.

*// CalculatorInterfaceImpl.java*import java.rmi.RemoteException;  
import java.rmi.server.UnicastRemoteObject;  
  
public class CalculatorInterfaceImpl extends UnicastRemoteObject implements *CalculatorInterface* {  
 public CalculatorInterfaceImpl() throws RemoteException {  
 super(); *// handles marshalling and unmarshalling* }  
 @Override  
 public int add(int a, int b) throws RemoteException {  
 return a + b;  
 }  
 @Override  
 public int sub(int a, int b) throws RemoteException {  
 return a – b;  
 }  
 @Override  
 public int mul(int a, int b) throws RemoteException {  
 return a \* b;  
 }

@Override  
 public int div(int a, int b) throws RemoteException {  
 return a / b;  
 }  
}

JAVA

1. We need to provide a mechanism to allow the stub to communicate with the skeleton. To do this, we create a new class in which we bind each of the skeletons we have created to a **unique name** using a naming service.

*// Server.java*import java.rmi.Naming;  
  
public class Server {  
 public static void main(String[] args) {  
 try {  
 CalculatorInterfaceImpl server = new CalculatorInterfaceImpl();  
 Naming.*rebind*("rmi://Calculator", server);  
 System.*out*.println("Server is waiting…");  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

JAVA

1. On the client side, we need to get a reference to the **remote objects** and invoke their **methods**.

*// Client.java*import java.rmi.Naming;  
import java.util.Scanner;  
  
public class Client {  
 public static void main(String[] arg) {  
 Scanner sc = new Scanner(System.*in*);  
 System.*out*.println("Enter the first number: ");  
 int a = sc.nextInt();  
 System.*out*.println("Enter the second number: ");  
 int b = sc.nextInt();

String host = "localhost";  
 try {  
 *CalculatorInterface* stub =(*CalculatorInterface*) Naming.*lookup*("rmi://" + host + "/Calculator");  
 System.*out*.println("Addition: " + stub.add(a, b));  
 System.*out*.println("Subtraction: " + stub.sub(a, b));  
 System.*out*.println("Multiplication: " + stub.mul(a, b));  
 System.*out*.println("Division: " + stub.div(a, b));  
 } catch (Exception e) {  
 System.*err*.println("Client Exception: " + e.toString());  
 e.printStackTrace();  
 }  
 }  
}

JAVA

Note that the **hostname** here refers to the domain on which we are hosting the server. By default, rmiregistry uses port 1099. If we want to use a different port, we can specify it as Naming.*rebind*("rmi://localhost:4000/Calculator", server). We also have to do the same during the lookup stage, i.e. String host = "localhost:4000".

Once all of this is done, we can see our code in action. This also involves several steps:

1. On the server side, we compile the **skeleton** and the class in which we use the **naming service**.

C:\> javac CalculatorInterfaceImpl.java  
C:\> javac Server.java

CMD

1. On the client-side, we do the same for the **stub** and the class where we called the **remote object’s methods**.

C:\> javac CalculatorInterface.java  
C:\> javac Client.java

CMD

1. Back on the server-side, we start the **naming service** which makes the names that we bound to the skeleton available to the clients.

C:\> start rmiregistry

CMD

This opens up a **new command-line window** which should remain open for the entire time we want our skeleton to be accessible. Closing the window will prevent any new connections.

If we are not using the default port, we have to use the following command:

C:\> start rmiregistry 4000

CMD

1. Finally, we can actually start the **server**.

C:\> java Server  
Server is waiting...

CMD

1. Running the **client-side code** now allows us to use the methods.

C:\> java Client  
Enter the first number:  
2  
Enter the second number:  
6  
Addition: 8  
Subtraction: -4  
Multiplication: 12  
Division: 0

CMD

The Java implementation of RMI we just saw is very **low-level**, which is why it is this complicated. We will most likely not end up using this, but rather a higher-level implementation such as **Enterprise JavaBeans**, which uses Java RMI underneath the hood.

## CORBA

We will only be looking into a brief overview of CORBA.

CORBA is a product of the **Object Management Group** (OMG), which is a large association of several companies devoted to improving remote object method invocation. As mentioned before, it is not an implementation, but rather a **specification**. An individual implementation of this specification is called an **Object Requests Broker** (ORB), such as RMI ORB, CORBA ORB, etc. Since it is just a specification, CORBA is not limited to a single language.

Unlike **RMI ORBs**, which use a protocol called **Java Remote Method Protocol** (JRMP), **CORBA ORBs** use the **Internet Inter-ORB Protocol** (IIOP), which is based on **TCP/IP**. This protocol is what allows inter-operability between ORBs from different vendors.

Another huge difference with RMI is that CORBA uses a special language called the **Interface Definition Language** (IDL) to define interfaces. A schema specification, also written in IDL, describes the objects that are to be exported for remote usage. An ORB implements those interfaces, handling requests and responses.