

UNIT 5

RMI (Remote Method Invocation)

(5 Hours) — Advanced Java Programming (CAC354)

Important Notice — Please Read

These notes were prepared from the syllabus and general knowledge of Java. **They may contain errors** (typos, outdated API details, or mistakes in code and explanations), so please take care. Verify important points and test all code yourself, and cross-check with your textbook (Core Java Vol. II / Java: The Complete Reference), the official Java documentation, and your teacher's notes.

5.1 What Is RMI?

Remote Method Invocation (RMI) is a Java API (packages `java.rmi` and `java.rmi.server`) that allows an object running in one Java Virtual Machine to invoke methods on an object running in **another JVM** (possibly on another computer) as if it were a local object. It is Java's mechanism for **distributed object computing**, and it is pure Java: both client and server are written in Java.

- Hides the networking details (sockets, protocols) from the programmer.
- Uses Java **serialization** to send objects as parameters and return values.
- Supports **distributed garbage collection** and dynamic class loading.
- Built on **JRMP** (Java Remote Method Protocol); RMI-IIOP allows interoperability with CORBA.

5.2 The Roles of Client and Server

Role	Responsibility
Server	Creates the remote object (implementation), exports it, and registers (binds) it with the RMI registry under a name
Client	Looks up the remote object in the registry by name, receives a stub and calls methods on it
RMI Registry	A simple naming service (default port 1099) that maps names to remote object references

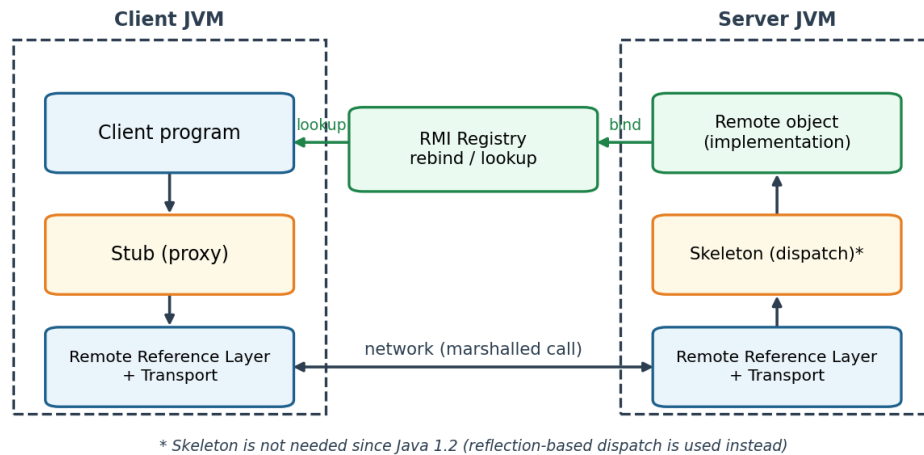


Figure 5.1 — RMI architecture

5.3 Remote Method Calls

When the client calls a method on the stub, the following happens:

- The client invokes a method on the **stub** (local proxy object that implements the same remote interface).
- The stub **marshals** (serialises) the method identifier and parameters and sends them across the network through the remote reference/transport layers.
- On the server, the call is received, the parameters are **unmarshalled**, and the actual method of the remote object is executed.
- The result (or exception) is marshalled and returned to the stub, which unmarshals it and returns it to the client code.

Note: Any network problem can occur during a remote call, so every remote method must declare `throws RemoteException` (a checked exception).

5.4 Stubs and Parameter Marshalling

A **stub** is a client-side proxy for the remote object. It implements the remote interface, so the client code calls it like a normal object. The stub:

- Opens a connection to the remote JVM,
- **Marshals** parameters (writes them to a stream as bytes),
- Waits for the reply, **unmarshals** the return value and gives it to the caller.

Historically the server side had a **skeleton** that unmarshalled parameters and dispatched the call. Since Java 1.2 skeletons are not needed (generic reflection-based dispatch), and since Java 5

stubs are generated **dynamically** at run time (dynamic proxies), so the old `rmic` compiler is no longer required (it was deprecated in Java 9 and removed in JDK 15).

Parameter passing in remote calls

Parameter type	How it is passed
Primitive types	By value (copied)
Serializable objects (not remote)	By value — object is serialised and a copy is created on the other side; changes are not reflected in the original
Remote objects (implement <code>Remote</code>)	By reference — a stub is sent; calls go back to the original object
Non-serializable, non-remote objects	Not allowed — causes a runtime exception (e.g. <code>MarshalException</code> / <code>NotSerializableException</code>)

5.5 The RMI Programming Model

To build an RMI application, follow these steps:

Step	Task
1	Define the remote interface (extends <code>java.rmi.Remote</code> ; each method throws <code>RemoteException</code>)
2	Write the implementation class (extends <code>UnicastRemoteObject</code> or exported explicitly)
3	Write the server : create the object, create/locate the registry and bind the object to a name
4	Write the client : look up the name in the registry and call methods
5	Compile everything; start the registry and server; run the client

5.6 Interfaces and Implementations

5.6.1 Remote Interface

```
import java.rmi.*;

public interface Calculator extends Remote {
    int add(int a, int b) throws RemoteException;
    int multiply(int a, int b) throws RemoteException;
    String greet(String name) throws RemoteException;
}
```

- The interface must be **public** and extend `java.rmi.Remote` (a marker interface).
- Every method must declare `throws RemoteException` (or a superclass such as `IOException` / `Exception`).

- Parameters and return types must be primitives, **Serializable** or **Remote**.

5.6.2 Implementation Class

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

public class CalculatorImpl extends UnicastRemoteObject implements Calculator {

    public CalculatorImpl() throws RemoteException {
        super(); // exports the object (anonymous port)
    }
    public int add(int a, int b) throws RemoteException { return a + b; }
    public int multiply(int a, int b) throws RemoteException { return a * b; }
    public String greet(String name) throws RemoteException { return "Hello, " + name + "!"; }
}
```

UnicastRemoteObject makes the object available for remote calls over TCP (point-to-point) and exports it automatically when constructed. If the class does not extend it, call `UnicastRemoteObject.exportObject(obj, 0)` explicitly (the call returns the stub).

5.7 The RMI Registry

The **RMI registry** is a bootstrap naming service on the server machine. The server **binds** a remote object to a name; clients **look up** that name to obtain a stub. The default port is **1099**. It can be started from the command line with `rmiregistry` or inside the server using `LocateRegistry.createRegistry(1099)`.

Class / Method	Purpose
<code>LocateRegistry.createRegistry(port)</code>	Starts a registry inside the current JVM
<code>LocateRegistry.getRegistry(host, port)</code>	Gets a reference to a (remote) registry
<code>registry.bind(name, obj)</code>	Binds a name; throws <code>AlreadyBoundException</code> if it exists
<code>registry.rebind(name, obj)</code>	Binds or replaces an existing binding
<code>registry.lookup(name)</code>	Returns the stub for a name
<code>registry.unbind(name)</code>	Removes a binding
<code>registry.list()</code>	Names of all bound objects
<code>Naming.rebind</code> / <code>Naming.lookup</code>	Older URL-based form: <code>rmi://host:port/name</code>

Note: For security, the registry only accepts bind/rebind/unbind calls from the same host where it runs; lookup and list can be done from anywhere.

5.8 Parameters and Return Values in Remote Methods

Because the client and server are in different JVMs, objects cannot be shared directly. As described earlier, non-remote objects are copied using serialization. Example of passing a serializable object:

```
import java.io.Serializable;

public class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    private int id; private String name;
    public Student(int id, String name) { this.id = id; this.name = name; }
    public int getId() { return id; } public String getName() { return name; }
}

public interface StudentService extends Remote {
    Student find(int id) throws RemoteException; // returns a COPY
    java.util.List<Student> all() throws RemoteException; // List must be serializable too
}
```

- **Pass by value** for serializable objects: modifications made by the server to its copy are not visible to the client unless the object is returned.
- **Pass by reference** for remote objects: the receiver gets a stub, so method calls go back to the original object.
- The class definition must be available on both sides (or loaded dynamically from a code base).

5.9 Remote Object Activation

Normally the server must be running for the remote object to exist. **Activation** (package `java.rmi.activation`) allows a remote object to be created **on demand**, when a client first calls it, rather than keeping it in memory all the time. A special daemon (**rmid**, the activation system) starts a JVM and creates the object when needed, and it can be deactivated when idle. The class `Activatable` is used instead of `UnicastRemoteObject`.

- Saves resources — objects exist only while needed.
- Survives server crashes/restarts since the object can be re-activated.
- Needs additional set-up: activation descriptor, rmid, security policy.

Note: The activation mechanism is legacy: it was deprecated in Java 15 and removed in Java 17. Mention it as a concept for the syllabus, but it is not available in recent JDKs.

5.10 Simple Client/Server Application Using RMI

Project layout

Calculator.java (interface), CalculatorImpl.java (implementation), CalculatorServer.java (server), CalculatorClient.java (client). All compiled with javac.

Server

```
import java.rmi.registry.*;

public class CalculatorServer {
    public static void main(String[] args) {
        try {
            Calculator calc = new CalculatorImpl();
            Registry registry = LocateRegistry.createRegistry(1099); // start registry
            registry.rebind("CalcService", calc); // bind name
            System.out.println("Server ready...");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Client

```
import java.rmi.registry.*;

public class CalculatorClient {
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry("localhost", 1099);
            Calculator calc = (Calculator) registry.lookup("CalcService");

            System.out.println("5 + 3 = " + calc.add(5, 3));
            System.out.println("5 * 3 = " + calc.multiply(5, 3));
            System.out.println(calc.greet("Nepal"));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Compiling and running

```
javac *.java
java CalculatorServer      # terminal 1 (also starts the registry)
java CalculatorClient      # terminal 2
```

Output:
5 + 3 = 8
5 * 3 = 15
Hello, Nepal!

Note: If the client runs on another machine, use the server's IP address in getRegistry(). You may also need to set -Djava.rmi.server.hostname= on the server and open port 1099 plus the object's port in the firewall.

Example 5.1 — Common errors

List some common reasons why an RMI program fails.

Solution

- **ConnectException**: registry/server is not running, wrong host or port.
- **NotBoundException**: lookup name differs from the bound name or object not bound.
- **ClassNotFoundException** / **UnmarshalException**: interface or serializable class not available on the client.
- **NotSerializableException** / **MarshalException**: a parameter does not implement Serializable.
- **ExportException** / **AlreadyBoundException**: port already in use or name already bound (use `rebind`).

5.11 Comparing RMI with CORBA

CORBA (Common Object Request Broker Architecture) is a language-independent standard from the OMG for distributed objects. Interfaces are described in **IDL (Interface Definition Language)** and objects communicate through an **ORB** using the **IIOP** protocol.

Feature	RMI	CORBA
Language	Java only (Java-to-Java)	Language independent (C++, Java, Python...)
Interface definition	Java interface extending Remote	IDL (Interface Definition Language)
Protocol	JRMP (and RMI-IIOP)	IIOP (via ORB)
Parameter passing	Serializable objects by value; remote objects by reference	Primitive/IDL types by value, objects by reference
Naming service	RMI registry	CORBA Naming Service (COS Naming)
Complexity	Simple, easier to learn	Complex, heavier
Dynamic class loading	Supported (code can be downloaded)	Not supported
Garbage collection	Distributed GC	No built-in distributed GC
Current status	Still part of Java SE	Removed from Java SE in Java 11

Summary: choose RMI when both ends are Java and simplicity is desired; use CORBA when objects written in different languages must interoperate. (Today, web services/REST, gRPC and messaging are more common than both.)

5.12 Short Questions for Revision

- What is RMI? Explain its architecture with a diagram.
- What are stubs and skeletons? What is marshalling?
- What is the RMI registry? How are objects bound and looked up?
- Write an RMI program (interface, implementation, server and client) to perform arithmetic operations.
- How are parameters passed in remote methods?
- What is remote object activation?
- Compare RMI with CORBA.

Key Takeaways — Unit 5

- ✓ RMI lets a Java object call methods of an object in another JVM; the client works with a stub (proxy).
- ✓ Stub marshals parameters, the server unmarshals them, executes and returns the result; all remote methods throw `RemoteException`.
- ✓ Remote interface extends `Remote`; implementation usually extends `UnicastRemoteObject`; server binds the name in the RMI registry (port 1099); client looks it up.
- ✓ Serializable parameters are passed by value; remote objects are passed by reference (as stubs).
- ✓ Activation (legacy) creates objects on demand; it was removed in Java 17.
- ✓ RMI is Java-only and simple; CORBA is language independent, uses IDL/IIOP and is more complex.

— End of Unit 5 —

Reminder: these notes may contain errors — verify against your textbook and run the code yourself.