RMI
(Remote Method Invocation)
Topics

- What is RMI? Why RMI?
- Architectural components
- Serialization & Marshaled Objects
- Dynamic class loading
- Code movement
- Codebase
- ClassLoader delegation
- RMI Security
- Writing RMI Server and Client
- Activation
- HTTP Tunneling
What is RMI?
What is RMI?

- RPC (Remote Procedure Call) between Java Objects

- General RPC behavior
  - Invoke remote methods
  - Pass arguments into methods
  - Receive results from methods

- RPC Evolution
  - Non-object-oriented RPC
  - CORBA (Object-oriented)
  - RMI (Object-based)
What is RMI?

- Differences from other RPC’s
  - RMI is Java-based
  - RMI supports code movement
  - RMI has built-in security mechanism
  - RMI exposure of network failures to application programmers through RemoteException
Why RMI?

- Capitalizes on the Java object model
- Minimizes complexity of distributed programming
- Uses pure Java interfaces
  - no new interface definition language (IDL)
- Preserves safety of Java runtime
- Recognizes differences of remote call from local call
  - partial failure
  - latency
  - no global knowledge on system state
RMI Architectural Components
RMI Architectural Components

- Remote interface
- Stub and Skeleton
- Remote object
Remote Interface

- Java interface
  - Specify *remotely* accessible methods
- Implemented by a class, an instance of which becomes a **remote object**
- Contract between caller of the remote method (RMI client) and remote object (RMI server)
- Extends `java.rmi.Remote` interface
  - Markup interface
Stub & Skeleton
Stub and Skeleton

- A tool (rmic) creates
  - RMI stub
  - (Optionally) RMI skeleton
- Gets created from RMI server implementation (not from RMI interface)
- Later version of RMI automatically creates them during runtime
Stub and Skeleton

• RMI Stub
  ✷ Resides in caller’s local address space
  ✷ Represents remote object to caller (client)
    ▪ Plays the role of proxy of remote object
    ▪ Implementation of Remote interface
    ▪ Caller invokes methods of RMI Stub locally
  ✷ Connects to the remote object
  ✷ Sends arguments to and receive results from remote object
    ▪ Performs marshaling and unmarshaling
Stub and Skeleton

• RMI Skeleton
  ◆ Resides in server’s address space
  ◆ Receives arguments from caller (RMI Client's Stub) and send results back to caller
    ■ Performs marshaling and unmarshaling
  ◆ Figures out which method of remote object to be called
  ◆ From JDK 1.3, RMI Skeleton gets created automatically via reflection
Remote Object

- Implementation of remote interface
- Needs to be exported
  - In order to be ready to receive calls from caller
- Can be exported in two types
  - Non-activatable (extends `java.rmi.server.UnicastRemoteObject`)
  - Activatable (extends `java.rmi.server.Activatable`
RMI Communication Model
RMI Communication Model

- Caller
- Remote Interface
- Stub
- Remote Object
- Skeleton
RMI Control Flow

1. Client object calls method on Stub.
3. Passes call to remote object.
4. Receives call and arguments.
5. Processes call.
6. Returns results.
7. Receives and unmarshals results.
8. Marshals arguments.
9. Receives return from Stub.

Interface definition
Declares method(s)
RMI Control Flow

• Caller (Client)
  1. invokes a method of a remote object

• Stub of the remote object
  1. intercepts the method call
  2. marshals the arguments
  3. makes calls to remote object
RMI Control Flow

• Remote object
  1. Receives the calls via Skeleton
  2. Unmarshals the arguments
  3. Performs the call locally
  4. Marshals the result
  5. Send the result to client

• Stub
  1. Receives result
  2. Unmarshal result
  3. Return result to client
Serialization in RMI
Marshaling and Unmarshaling

- Marshaling is a process of encoding objects to put them on the wire.
- Unmarshaling is the process of decoding from the wire and placing object in the address space.
- RMI uses Java programming lanaguage's serialization and deserialization to perform marshaling and unmarshaling.
  - These terms are used interchangeably.
Serialization in RMI

- Arguments/Results get serialized before being transported by sender
- Arguments/Results get deserialized after being transported by receiver
- Arguments/Results in RMI can be one of the following two
  - Remote object
  - Non-remote object/primitive
Serialization in RMI

- For remote object
  - Remote interface type
  - "Pass by reference" semantics
    - Stub is kind of a reference to remote object
    - Stub gets serialized (instead of remote object itself)

- For non-remote object
  - Normal serialized copy of the object
  - Should be type of java.io.Serializable
  - "Pass by value" semantics

- For (non-remote) primitive
  - Normal copy of the object
// Arguments and Returns are non-remote objects
public interface SayHelloStringRemote extends Remote {
    public String SayHelloString (String message)
    throws RemoteException;
}

// Arguments has both remote and non-remote objects
public interface SayHelloObjectRemote extends Remote {
    public String SayHelloObject (String message,
            SayHelloStringRemote someName)
    throws RemoteException;
}
Serialization

- Serialized copy of an object
  - Stream of bytes
  - Persistently maintains state of an object
    - State of non-static and non-transient variables of the object
  - If the class is unknown to the recipient, it will be downloaded automatically
    - Maintains information on “where to get the class bytecodes”
      - codebase annotation
      - Who performs the codebase annotation?
Serialization

- Serialized copy defines state
- Class files define behavior
- Both can be moved around over the network
  - Collectively this is called "Code movement"
Marshalled Objects
What and Why Marshalled Objects?

- **Container object** of serialized object
  - Constructed by passing object into constructor
  - `get()` method retrieves the deserialized object

- Used when you want to maintain the serialized object without deserializing it immediately
  - Storage service of objects
    - Lookup service
Dynamic Class Loading
Dynamic Class Loading

- Class bytecodes (Class file) get downloaded during runtime
  - When caller does not have the class bytecodes in local classpath
    - RMI Stub needs to be downloaded to RMI Caller’s address space from somewhere
  - Serialized copy of an object contains “where to get class bytecodes” information
    - Codebase annotation
Who Does Provide Codebase Annotation Information?

- By the exporter of the class
- Via Export codebase (RMI codebase) property
  - `java.rmi.server.codebase`
  - Typically via HTTP URL
When Does the Codebase Annotation occurs?

- Whenever an object gets serialized
- For remote object
  - Codebase information of Stub class
- For non-remote object
  - Codebase information of normal class
RMI Server and Client Deployment Scenario

- Both client and server have RMI Remote interface class in their local classpaths
- Server has HelloWorld_Stub class in its local classpath
- Client does not have HelloWorld_Stub class in its localpath
  - He could, but is diminishes the whole purpose of class downloading
- Server exports HelloWorld_Stub class via HTTP server
RMI Server and Client Deployment Scenario

- Client gets HelloWorld_Stub serialized object via Registry
  - Client typically does **not** have HelloWorld_Stub class in its local classpath
  - So it will read the RMI codebase annotation (from the serialized stub object) and will try to download the HelloWorld_Stub class from the location specified in codebase annotation
Code (and Data) Movement
Code (and Data) Movement

- Performed in two phases
  1. Serialized object (Marshalled Object) gets moved
  2. Class files get downloaded

- Code
  - Represented by class files

- Data
  - Represented by state captured in serialized object
Serialized Object

- Contains
  - Values of the fields of the object
  - Name of the class
  - Location of the class
    - Via codebase annotation performed by the exporter of the class
    - RMI codebase property
What is Codebase?

- **Location** where class bytecodes (Class files) reside
Two types of Codebase

- **Import codebase**
  - codebase your local VM uses to load classes it needs
  - specified via `CLASSPATH` or `-cp` option

- **Export codebase (RMI codebase)**
  - codebase remote VMs use to obtain the class files "exported" from your local VM
  - specified via `java.rmi.server.codebase` property
    - Codebase annotation
Behind the Scene Activities

- Any objects marshaled by a server will be annotated with RMI codebase
  - For remote object, the stub object gets marshaled and annotated
- When a client instantiates the object, the bytecodes of the class will be downloaded by RMIClassloader from the location specified as RMI codebase
RMI codebase forms

- Could be in any URI form
  - HTTP (Recommended)
  - FTP
  - FILE (Not recommended)
- Classes can be accessible via
  - JAR
  - Directory path
    - Trailing slash required
RMI codebase examples

- Directories need a trailing slash
  -Djava.rmi.server.codebase="file:/export/home/ btm/classes/
  -Djava.rmi.server.codebase="http://daydreamer:8080/export/home/ btm/root/dir/"

- Jar files do not need a trailing slash
  -Djava.rmi.server.codebase="file:/export/home/ btm/jars/examples-dl.jar"
  -Djava.rmi.server.codebase="http://daydreamer:8080/export/home/ btm/jars/examples-dl.jar"

- You can specify multiple locations
  -Djava.rmi.server.codebase="http://daydreamer:8080/export/home/ btm/jars/examples-dl.jar
  http://daydreamer:8080/export/home/ btm/root/dir/"
Typical Causes of Problems

• The `java.rmi.server.codebase` (RMI codebase) property was not set at all
  ♦ Do not use “localhost”

• RMI codebase was set, but HTTP server is not running

• RMI codebase was set, HTTP server is running, but the class is not present under the proper path in HTTP server

• The port number on which HTTP server is listening is not the same as the port number in the RMI codebase

• The name of the host on which HTTP server is running is not the same as the hostname in the RMI codebase

• If a non-jar URL is being used in the RMI codebase, there is no trailing slash (if class file location is in a jar file, no trailing slash is required)
Implementation Guideline

- Client has remote interface class file in its local classpath (unless it uses reflection)
- The classes that are needed for implementation should be downloadable from the server
  - Stub classes
  - Interface classes
    - Needed when client does not have interface classes in its local path
  - Any other classes that the stub and interface refers to
- Make jar file in the form of xxx-dl.jar
ClassLoader
Delegation
ClassLoader Delegation

• Introduced in JDK 1.2
  ◆ Class files are searched based on classloader hierarchy
    ▪ Bootstrap classloader
    ▪ Extension classloader
    ▪ Application classloader
    ▪ RMI classloader
  ◆ Ask parent classloader first
    ▪ Reason why a class file in local CLASSPATH gets picked up first before the same class file gets downloaded from remote location
Classloader Hierarchy

- Bootstrap Classloader
- Extension Classloader
- Application Classloader
- RMI Classloader

Delegation

- Bootstrap Classpath
- Extension Classpath
- CLASSPATH
- RMI codebase
Example
RMI Security
Java Security

- In Java, SecurityManager handles security control
  - Based on security policy file
  - Security policy define “permission control” based on
    - Where the code came from
    - Who signed the code
  - Examples
    - All code signed by Dave can write to a particular directory
    - Any code downloaded from a particular HTTP server site has no filesystem access
Security Policy Example

• Give all all permission to any code
  grant {
    permission java.security.AllPermission "", "";
  };

• Use only during testing
  ✷ **Never** use it in production environment
Security Policy Example

```java
grant codebase "file:${java.class.path}" {
    // file system dependent permissions for unix file system
    permission java.io.FilePermission "./\*", "read,write,execute,delete";
    permission java.io.FilePermission "/tmp", "read,write,execute,delete";
    permission java.io.FilePermission "/tmp/-", "read,write,execute,delete";
    permission java.io.FilePermission "/var/tmp", "read,write,execute,delete";
    permission java.io.FilePermission "/var/tmp/-", "read,write,execute,delete";
    // uncomment this one if you need lookup to accept file: codebases
    // permission java.io.FilePermission "<<ALL FILES>>", "read";
    permission java.lang.RuntimePermission "modifyThreadGroup";
    permission java.lang.RuntimePermission "modifyThread";
    permission java.net.SocketPermission "*:1024-", "connect,accept";
    // for http: codebases
    permission java.net.SocketPermission "*:80", "connect";
    permission java.net.SocketPermission "224.0.1.84", "connect,accept";
    permission java.net.SocketPermission "224.0.1.85", "connect,accept";
    permission java.net.SocketPermission "java.rmi.server.hostname", "read";
    permission java.util.PropertyPermission "com.sun.jini.reggie.*", "read";
    permission java.util.PropertyPermission "net.jini.discovery.*", "read";
    permission net.jini.discovery.DiscoveryPermission "*";
    // file system dependent permissions for windows file system
    permission java.io.FilePermission ".\\*", "read,write,execute,delete";
    permission java.io.FilePermission "c:\\temp", "read,write,execute,delete";
    permission java.io.FilePermission "c:\\temp\\-", "read,write,execute,delete";
    permission java.io.FilePermission "c:\\windows\\temp", "read,write,execute,delete";
    permission java.io.FilePermission "c:\\windows\\temp\\-", "read,write,execute,delete";
    // Deleted the rest
};
```
RMI Security

• Security is a serious concern since executable code is being downloaded from remote location

• In RMI, SecurityManager has to be installed in order to be able to download any code from remote location
  ◆ Without its installation, RMI will search for class files only from local classpath

• The security policy file further specifies the “permission control”
RMI Security

- RMI client needs to install security manager because it needs to download Stub file of RMI object
- A simple RMI server might not need to install security manager if it does not need to download class files from remote location
  - It is still good practice to install it anyway
Writing RMI Server
Steps of Writing RMI Server

- S1: Define remote interface
- S2: Implement remote interface
- S3: Provide an implementation for each remote method
- S4: Write server class
  - Contains *main()* method
  - Create and export remote object
  - Create and install a security manager
  - Register remote object with RMI registry
S1: Define Remote Interface

- Defines methods that are called remotely
- Must be declared as `public`
- Extends the `java.rmi.Remote` interface
- Each method must declare `java.rmi.RemoteException`
- The data type of any remote object that is passed as an argument or return value (either directly or embedded within a local object) must be declared as the Remote interface type (for example, Hello) not the implementation class (HelloImpl).
// A remote interface for getting increasing numbers

class NextNumber extends Remote {
   public int getNextNumber(int n)
      throws RemoteException;
}

import java.rmi.Remote;
import java.rmi.RemoteException;
S2: Write the implementation

- Implement the remote interface
- Extend one of the two remote classes
  - `java.rmi.server.UnicastRemoteObject`
  - `java.rmi.activation.Activatable`
- Write constructor for the remote object
  - By extending one of the two remote classes above, they are automatically exported
    - You can manually export it as well
  - Throw `RemoteException`
- Install Security Manager
- Register remote objects with RMI registry
S2: Example

// A server object that implements the NextNumber
// remote interface

package core.xa;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.Naming;
import java.rmi.server.UnicastRemoteObject;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.net.MalformedURLException;

package core.xa;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.Naming;
import java.rmi.server.UnicastRemoteObject;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.net.MalformedURLException;
public class NextNumberImpl extends UnicastRemoteObject implements NextNumber {

    public NextNumberImpl() throws RemoteException {

        // Install SecurityManager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
    }
}
try {
    // Bind it with RMI Registry
    String host = InetAddress.getLocalHost().getHostName();
    String url = "rmi://" + host + "/nextNumber";
    Naming.rebind(url, this);
    System.out.println("Server bound to: " + url);
} catch (UnknownHostException ex) {
    System.err.println("Couldn't get local host name");
    System.exit(1);
} catch (RemoteException ex) {
    System.err.println("Couldn't contact rmiregistry.");
    System.exit(1);
} catch (MalformedURLException ex) {
    System.exit(1);
}
S3: Implement Remote Methods

• Implement service logic within the methods

• Do not throw RemoteException
  ◆ They are called locally (same address space) from RMI Skeleton
S3: Example

// Implement remote method
public int getNextNumber(int n) {
    return n+1;
}

S4: Write Server Class

• Setup (or Wrapper) class
  ◆ Contains main() method
  ◆ Create one or more instances of remote objects
// Write main method
public static void main(String[] args) {
    try {
        NextNumberImpl server = new NextNumberImpl();
    } catch (RemoteException ex) {
        ex.printStackTrace();
    }
}
Create one or more instances of a remote object

• Remote object gets exported during instantiation process
  • Remote object is ready to receive incoming RMI calls
Register the remote object with RMI Registry

• RMI Registry is a simple naming service
  ♦ Bootstrap mechanism
  ♦ Typically is used by caller to get the remote reference of the first remote object

• Client gets reference to remote object - actually reference to stub object of the remote object
Writing RMI Client
Steps of Writing RMI Client

• Install security manager
• Get a reference to the remote object implementation
  ◆ The registry returns the Stub instance of the remote object bound to that name
• Invoke remote methods
Example

// A client to test the NextNumber server
package core.xa;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.NotBoundException;
import java.rmi.Naming;
import java.net.MalformedURLException;
public class NextNumberClient {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage: NextNumberClient <url> ");
            System.exit(1);
        }
    }
}
if (System.getSecurityManager() == null) {
    System.setSecurityManager(
        new RMISecurityManager());
}
Remote r = null;
try {
    r = Naming.lookup(args[0]);
} catch (RemoteException ex) {
    System.err.println("Couldn't contact registry.");
    System.exit(1);
} catch (NotBoundException ex) {
    System.err.println("There is no object bound to "+ args[0]);
    System.exit(1);
} catch (MalformedURLException ex) {
    System.err.println("The string "+ args[0] + " is not a valid RMI URL");
    System.exit(1);
}
try {
    if (r instanceof NextNumber) {
        NextNumber nn = (NextNumber) r;
        System.out.println("Next number after 1 is "+nn.getNextNumber(1));
        System.out.println("Next number after 2 is "+nn.getNextNumber(2));
    } else {
        System.err.println("Uh oh, the name "+args[0] + "isn't a NextNumber");
    }
} catch (RemoteException ex) {
    ex.printStackTrace();
}
Building, Deploying RMI Server and Client
Build Process

- Compile
  - Compile client and server in separate directories

- Generate Stub and Skeleton
  - Use RMIC
  - Takes fully qualified Java class name of Implementation class
    - `rmic core.xa.NextNumberImpl`
Example: RMIC

C:\files>rmic core.xa.NextNumberImpl

C:\files>cd core\xa

C:\files\core\xa>dir
5,260 01-04-01 1:36p HelloWorldServiceActivatable.java
  249 01-04-01 1:36p NextNumber.java
1,802 01-04-01 1:36p NextNumberClient.java
1,540 01-04-01 1:36p NextNumberImpl.java
  227 06-21-01 10:56p NextNumber.class
1,719 06-21-01 10:56p NextNumberImpl.class
2,023 06-21-01 10:56p NextNumberClient.class
3,218 06-21-01 10:59p NextNumberImpl_Stub.class
1,640 06-21-01 10:59p NextNumberImpl_Skel.class
Server Deployment Process

- **Start RMI Registry**
  - Has to run on the same machine that has RMI object
    - `rmiregistry`

- **Start HTTP server**
  - Root directory has to directory structure that is comparable to exportable class files
    - `C:\files\core\xa\NextNumberImpl_Stub.class`
    - `C:> java -jar C:\files\jni1_2\lib\tools.jar -port 8081 -dir c:\files\ -verbose`
Server Deployment Process

- Create Security policy file
- Run the RMI server
  - C:\files> java

```java
-Djava.security.policy=c:\files\jini1_2\policy\policy.all
-Djava.rmi.server.codebase=http://hannah:8081/
core.xa.NextNumberImpl
```
Client Deployment Process

- Create Security policy file
- Start HTTP server if needed
  - C:> java -jar C:\files\jini1_2\lib\tools.jar -port 8082 -dir c:\files\ -verbose
  - In this example, no need for this since client is not exporting any classes
- Run the RMI Client
  - C:> cd \files
  - C:\files> java
     -Djava.security.policy=c:\files\jini1_2\policy\policy.all core.xa.NextNumberClient rmi://hannah/nextNumber
Activation
Activation

• Why activatable objects?
  - Service could be shut down inadvertently or intentionally
  - Activatable service gets restarted automatically when system boots up or on-demand basis
    ▪ Activatable service needs to be started (registered with RMID) only once

• Activation system components
  - RMID (Activation system daemon)
  - RMID log file
    ▪ Persistently stores all activatable objects
    ▪ Default is <Directory where RMID gets started>/log directory
  - Activatable services
    ▪ They are run as child processes of RMID
Control Flow of Activation

[A new activatable service with running RMID]

(1) RMID running
(2) A new service registers with RMID and gets a special RMI reference - RMID logs the information in persistent storage
(3) The service (actually the proxy object) registers with the lookup service - the proxy object contains the RMI reference
(4) The service goes inactive (intentionally or inadvertently)

(5) Client, via lookup operation, retrieves the proxy object, which contains the RMI reference
(6) Client Stub talks to the service directly and gets an exception since the service (as an RMI server) is inactive
(7) Client Stub then talks to RMID

(8) RMID restarts the service if necessary in a new VM
(9) Client now can talk directly with the service
Control Flow of Activation

[RMID crash and reboot]

(1) A service is registered with RMID
(2) RMID crashes and reboots
(3) RMID reads the log file and restarted the services (the ones which set the RESTART flag during the registration with RMID).

(5) Client, via lookup operation, retrieves the proxy object, which contains the RMI reference
(6) Client talks to the service directly.
RMID

- As long as RMID is running and RMID log file is persistent, a service can get started on “as needed” basis
RMI Tunneling
RMI Tunneling

• Features
  ♦ Protocol runs over HTTP protocol
  ♦ Allows RMI client within a firewall to talk to an RMI server outside of the firewall

• Limitation
  ♦ RMI server cannot talk back to the RMI client
Review Points
Locating Remote Objects
Remote Communication
Loading Classes

- Client
- Server
- Registry
- Web server

- RMI
- URL protocol
Method Invocation