RMI - Introduction

• RMI stands for “Remote Method Invocation”

• RMI is a framework for developing Java-centric distributed applications

• Using the RMI infrastructure and packages, RMI based Java clients can remotely invoke methods on RMI based Java server objects. These RMI based server objects can be running in separate JVM’s – either on the same machine or on another host
RMI - Introduction

• The conceptual forerunner of RMI is Remote Procedure Calls (RPC's)
  - RMI is an object oriented evolution of this technology

• RMI is conceptually similar to other distributed object architectures such as CORBA and DCOM
  - CORBA and its interoperation with RMI is discussed in a future lesson

• The RMI and CORBA paradigms are merging, via support for IIOP
  - IIOP = Internet Inter-Orb Protocol

RMI - Introduction

• RMI clients communicate transparently with RMI server objects by invoking methods on a client side proxy object

• The proxy object serializes the method parameters and streams them to the appropriate server object instance. Return parameters are unmarshalled in similar fashion

• RMI allows for clients and servers to pass objects as method parameters and return values (as opposed to only simple data types)

• One major difference between RMI and CORBA (other than the Java centric nature of RMI) is RMI's ability to pass objects by value

RMI - Introduction

• If a class used in a method parameter or return value is unknown to the client or server it can be dynamically loaded using RMI codebases

• RMI provides a mechanism for distributed garbage collection to clean up any distributed server objects that are no longer referenced by any distributed clients

• In cases where firewalls restrict network traffic to certain ports, RMI can use HTTP tunnelling
RMI - History

- There have been several releases of RMI:
  - RMI was originally distributed as a separate package to JDK 1.0
  - JDK 1.1 included core support for RMI
  - JDK 1.2 included numerous enhancements, including object activation and the capability for RMI to use custom socket protocols. RMI over IIOP was included as an extension
  - JDK 1.3 brings RMI over IIOP into the standard API set

RMI - Architecture

- The basic RMI architecture is shown in the figure
- The RMI client calls an object implementing an interface that corresponds to the remote interface exposed by the RMI server
- The interface is implemented by an RMI “stub”. The stub marshals the call parameters into a serialized packet of data that is sent via a communications protocol to the RMI server
- On the server side, the data is received (by an RMI “skeleton” for RMI 1.1, or a generic server call handler for RMI 1.2), unmarshalled into the original parameters and used to call the RMI server object implementation provided by the developer

RMI - Architecture (continued)

- Responses are handled in a similar manner. The response information from the server object is marshalled and sent to the client “stub”. The “stub” unmarshals the data and passes it to the RMI client
- The RMI transport layer is responsible for the provision of reliable data transfer between the client and server endpoints. The most widely used RMI transport layers are TCP based
- There are 2 RMI transport protocols in common use:
  - Java Remote Method Protocol (JRMP) – the original RMI protocol
  - Internet Inter Orb Protocol (IIOP) – the standard CORBA protocol
RMI - Packages

- The packages you will encounter when developing RMI based applications are:
  - java.rmi - Core RMI classes and interfaces, in particular the Remote interface and many standard exceptions
  - java.rmi.server - RMI server specific classes and interfaces
  - java.rmi.dgc - RMI distributed garbage-collection (DGC)
  - java.rmi.registry - the RMI registry lookup service
  - java.rmi.activation - RMI Object Activation classes and interfaces (RMI objects activated on client request)
  - javax.rmi.CORBA - all CORBA specific classes used for implementing and using RMI/IIOP objects

RMI - Tools

- RMI comes with a number of tools to support development and the runtime operation of RMI based programs.
  - rmic - Java RMI stub compiler. Used to generate stub and skeleton classes based on a compiled RMI server implementation
  - rmiregistry - Java remote object registry. The registry allows RMI server objects to be registered under a name that can be used by clients to "look up" the server object
  - rmid - Java RMI activation system daemon. This daemon allows server objects to be registered and created on demand on a client request
  - idlj - IDL to Java compiler. IIOP equivalent of rmic - used to generate RMI/IIOP based stubs and skeletons
  - tnameserv - RMI/IIOP naming service. IIOP equivalent of rmiregistry - allows RMI/IIOP objects to be registered and looked up by RMI/IIOP or CORBA clients
  - HTTP server - RMI can use a HTTP Server for dynamically downloading classes to RMI clients or servers that need to load such classes into their environments
RMI - Infrastructure Configuration

• Before an RMI server can make its services available to clients, a runtime infrastructure must be configured and brought online. The infrastructure components that may be needed are:

  § RMI registry and/or RMI/IIOP naming service – you will need to configure and start one (or both) of these so that RMI servers can bind to them and RMI clients can look up server objects

RMI - Infrastructure Configuration (2)

• RMI activation daemon(s) – you will need to start an activation daemon on each host that will create activatable RMI server objects on demand

• HTTP server – you may need to configure and start a HTTP server to serve Java classes to RMI clients and servers that need to dynamically load these classes. This component is optional - it is not necessary if the all relevant classes are loaded by both the client and server programs as part of the configuration of these programs

RMI - Development Process

1. Define the remote interface.
   The first step is defining the interface for the service that you wish to implement

2. Implement the RMI server.
   The next step is to create the RMI server class. This class should implement your remote interface and extend one of the RMI server classes. The different RMI server classes that can be extended allow either preactivated, dormant but activatable or RMI/IIOP behaviour
RMI – Development Process (2)

3. Generate the RMI skeletons and stubs.
Using the rmic tool you can generate RMI server skeletons and RMI client stubs for your RMI server class (which must be compiled first before rmic can be used). Note that skeletons are not created in RMI 1.2, only RMI 1.1.

4. Implement an RMI server registrar.
The next step is to create a class (typically with a main() method) that does one of the following, depending on the type of server class:
• Registers the RMI server with the RMI registry
• Registers the activatable RMI server with the RMI activation daemon
• Registers the RMI/IIOP server with the RMI/IIOP naming service

RMI – Development Process (3)

5. Implement an RMI client.
The final step is to implement an RMI client. The RMI client must first look up the RMI server object. This is done using either the RMI registry or the RMI/IIOP naming service depending on the type of server. The RMI stubs (generated by rmic) are used transparently to the client – the client is returned an interface for the remote object by the registry / naming service.

• The next section will discuss the process in a little more detail. The examples given are based on standard Java RMI, rather than RMI/IIOP – see Perrone & Chaganti for a discussion of the differences when implementing an RMI/IIOP application.

RMI – Remote Interfaces

• The first step to create an RMI application is to define an RMI interface

• The interface you define must extend java.rmi.Remote
  - This interface is a marker interface analagous to the Serializable interface

• Only the methods you want to make available remotely should be defined in this interface
• All methods must throw java.rmi.RemoteException
Example – Remote Interface

```java
/* Thing.java */
package example;
import java.rmi.*;
public interface Thing extends Remote
{
    void thingMethod (String arg) throws RemoteException;
}
```

RMI – Implement the RMI Server

- The next step is to implement the RMI server
- Your RMI server class must implement the remote interface you defined previously
- It must also extend java.rmi.server.UnicastRemoteObject
- The method implementations must throw the correct exceptions defined in your remote interface and UnicastRemoteObject
- Your RMI server code must be thread-safe

Example – RMI Server

```java
/* ThingImpl.java */
package example;
import java.rmi.*;
public class ThingImpl
extends java.rmi.server.UnicastRemoteObject
implements Thing
{
// implementation must have an explicit constructor
// in order to declare the RemoteException exception
public ThingImpl() throws RemoteException {
    super();
}
public void thingMethod(String arg) throws RemoteException {
    // do thing
}
```
RMI - Generate Stubs and Skeletons

- Once your RMI server class has been written, you need to compile your Java source files into Java class files.
- The rmic compiler can now be used to generate your RMI stub and skeleton classes.
- The stub and skeleton classes are generated within the same package as your RMI server.
- For the previous example, you would run:

```
rmic example.ThingImpl
```

RMI - Implement the server registrar

- Now we have a “distributable” server object, it needs to be made available to clients.
- Making the server available is done by registering a remote reference to the server object with a naming service.
- Naming services enable clients to look up distributed objects based on some identifier - a name for the object in the form of a string.
- RMI uses the the RMI registry as its naming service.

RMI - Implement the server registrar

- To register a remote reference with the RMI registry, the registry (rmiregistry) must be running.
- Remote references to instantiated server objects are registered with the naming service using one of:
  - `java.rmi.Naming.bind()`
  - `java.rmi.Naming.rebind()`
Example – RMI Server Registration

/* ThingServer.java */
package example;

import java.rmi.Naming;

public class ThingServer {
    public static void main(String args[]) {
        try {
            // create instance of remote object
            Thing t = new ThingImpl();

            // bind remote object to naming service (call it ThingService)
            Naming.rebind("rmi://localhost:1099/ThingService", t);
        } catch (Exception e) {
            ...
        }
    }
}

Example – RMI Client

/* ThingClient.java */
package example;

import java.rmi.*;

// get a reference to the remote object from the naming service
Thing t = (Thing) Naming.lookup("rmi://localhost:1099/ThingService");

RMI – Implement the RMI client

- Your client will make calls to the RMI stub code you generated using rmic. The RMI stub code implements your remote interface
- A reference to the server object is obtained by the client from the RMI naming service via the java.rmi.Naming.lookup() method
- Server classes used by the client must be able to be loaded by the client
- There are two mechanisms for doing this:
  1. Adding the relevant server stub classes and any classes passed between the server and client to the client CLASSPATH, or
  2. Configuring dynamic RMI class loading
RMI - Summary

- RMI is a part of the core JSDK. It is used for developing distributed applications in Java.
- RMI can interoperate with CORBA objects using RMI/IIOP.
- RMI is not used by developers building J2EE applications directly - however it is important to understand RMI as it provides the foundation for other J2EE services such as Enterprise Java Beans.