Introduction to RMI

RMI Introduction

- Java Remote Method Invocation (RMI)

- **Sources:**
- `http://java.sun.com/products/jdk/rmi/index.html`
  - for JDK 1.1; includes Java RMI tutorial and specification
- `http://java.sun.com/products/jdk/1.2/docs/guide/rmi/index.html`
  - describes RMI enhancements for upcoming JDK 1.2

RMI Introduction

- Java's Remote Method Invocation (RMI)
  - writing distributed applications in Java (only)
- RMI provides the ability to:
  - define types whose instances may be remote
    - i.e., execute on another Java virtual machine
    - typically another address space or node
  - invoke remote objects transparently
  - pass objects by reference or by value as parameters in a remote invocation
  - obtain a reference to a remote object from a simple name service
  - download the code necessary to access a remote object automatically

RMI basics

- Java is a *class-based* object-oriented programming language
  - Supports *single inheritance*
- A Java *interface* is essentially an abstract class
  - a collection of method *definitions* (and constant declarations)
  - methods and constants declared in an interface are implicitly `public`
- An interface can be derived from *one or more* interfaces
- A class can *implement* one or more interfaces
  - as well as being derived from at most one other class
- The interface and class hierarchies are orthogonal

Defining a remote interface

- A type whose instances may be invoked remotely is defined as a *remote interface*
- A remote interface is simply an interface that extends the interface `java.rmi.Remote` (no IDL!!!)
- The methods of a remote interface must be defined to throw the exception `java.rmi.RemoteException`
- Otherwise syntactically *identical* to any other Java interface

Defining a remote interface

- For example, the interface exported by our game server:

Remote interfaces must be `public`

Import package `java.rmi`

```
package DistributedGame;

import java.rmi.*;

/* Interface exported by the game server. */
/* Allows a player to join in a game. */
public interface GameServer extends Remote {
  Game join() throws RemoteException;
}
```

`join` takes no parameters and returns a `Game`

Making a remote object available

- Remote objects are instances of classes that implement remote interfaces
- Such a class simply implements the methods defined in the remote interface
- To be accessible remotely, a remote object must be *exported*
  - i.e., listen for incoming invocation requests
- An object is implicitly exported if its class derives from `java.rmi.server.UnicastRemoteObject`
  - not necessary; an object may also be exported explicitly (see later)

Making a remote object available

For example, an (outline) implementation of the game server interface:

```
package DistributedGame;

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

public class GameServerImpl
  extends UnicastRemoteObject
  implements GameServer
{
  public GameServerImpl() throws RemoteException {
   super();
   ...
  }

  public Game join() throws RemoteException {
   ...
  }
```

UnicastRemoteObject constructor exports the object

Making a remote object available

- Remote objects are typically hosted by server programs
- A server program is simply a Java application that creates one or more remote objects
- Such a server must install an appropriate security manager - typically an instance of `java.rmi.RMISecurityManager`
- A server may advertise references to one or more of the objects that it hosts via the local RMI registry
- The registry allows a binding between a URL and an object reference to be made and subsequently queried by potential clients

Making a remote object available

```
/* the server mainline */
public static void main(String args[]) {

  System.setSecurityManager(new RMISecurityManager());

  try {
   GameServerImpl server = new GameServerImpl();
   Naming.rebind("//mars.dsg.cs.tcd.ie/GameServer",
server);
    System.out.println("GameServer installed.");
  } catch (Exception e) {
   System.out.println("GameServer exception.");
  }
  /* now the server object is waiting for requests */
 }
}
```

create and install a security manager

create a server object and register it with the registry

Accessing a remote object

- From a client's perspective access to a remote object is transparent...
- ...except that the client has to be prepared to handle `java.rmi.RemoteException` when invoking a remote object
- If `java.rmi.RemoteException` is raised the caller has no guarantee as to how far the invocation proceeded before failing
- The client will normally obtain a reference to its first remote object from the registry

Accessing a remote object

- For example, an implementation of an applet that wants to join a game:

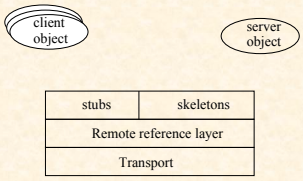
```
package DistributedGame;

. . .
import java.rmi.*;

public class TicTacToe extends Applet  implements . . . {

public void init() {
 . . .
 try {
  server = (GameServer)Naming.lookup("//"
            + getCodeBase().getHost() + "/GameServer");
  game = server.join();
 } catch (RemoteException e) {
  showStatus("Applet initialisation failed.");
  stop();
 }
}
```

Note cast to remote *interface* type

Invocation of remote object is syntactically identical to any other invocation

RMI details

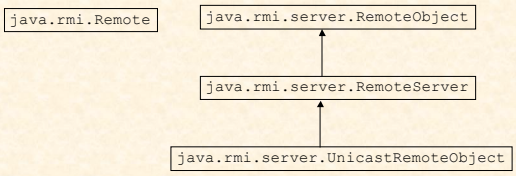
- **RMI Architecture 1**
- In principle, a three layered architecture:



RMI details

- Main class hierarchies:

Implements remote reference layer
- semantics of remote objects (`equals`)
Only implemented for singleton objects



RMI details

- A class can implement several remote interfaces as well as implementing methods that do not belong to any remote interface
  - such methods cannot be invoked on a remote object
- Clients always refer to remote objects via their remote interface types not their implementation class type
- A reference to a remote object can be passed as a parameter or returned as a result of any method invocation
- In particular, remote objects are passed *by reference* in remote method invocations
- **When passing a remote object as a parameter, the stub for the remote object is passed**

Concurrency control

- For example,
- Given two references, `r1` and `r2`, to a remote object (transmitted in different invocations):
  - `r1 == r2` is `false!!!!` (different stubs)
  - `r1.equals(r2)` is `true` (stubs for the same remote object)

RMI details

- In fact, the parameters to or return value from a remote method can be of any Java type that is *serialisable*
  - i.e., non-remote objects can be passed and returned
- Non-remote objects are passed by value in remote method invocations
  - i.e., a copy of the object is passed
- **Effectively, a new object is created for each invocation**

RMI details

- For example,
- Given variables, `l1` and `l2`, referring to the same non-remote object transmitted in different invocations:
  - `l1 == l2` is `false` (different copies)
  - `l1.equals(l2)` is `true/false`
  - (depending on whether the state of the source object changed between the invocations)

Example 2

- The original version of our TicTacToe game has the players poll the game to determine when it is their go
- An alternative is to have the game make a callback to the player whenever the opponent makes their move
- Now player must also be implemented as remote objects (capable of being invoked remotely)
- We will need the following new interfaces:

Example 2

```
/* Interface exported by the game. */
/* Allows a player to particpate in a game with another
player. */
public interface Game extends Remote {
  /* return a handle for the game */
  String getName() throws RemoteException;
  /* determine if caller has first move */
  boolean firstMove(Player player) throws RemoteException;
  /* handle a move by the caller */
  void reportMove(Player player, Point move) throws
RemoteException;
}

/* Callback interface exported by each player. */
public interface Player extends Remote {
  /* handle a move by the remote opponent */
  void reportMove(Point move) throws RemoteException;
}
```


Example 2

- The main implementation problem is that the client applet can't extend `UnicastRemoteObject` as before
- The client must take responsibility for exporting itself explicitly

Example 2

```
package DistributedGame;

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

public class TicTacToe extends Applet implements Player,... {

public void init() {
 . . .
 try {
  UnicastRemoteObject.exportObject(this);
  server = (GameServer)Naming.lookup("//"
            + getCodeBase().getHost() + "/GameServer");
  game = server.join();
 } catch (RemoteException e) {
  showStatus("Applet initialisation failed.");
  stop();
 }
}
. . .
```


Tutorial

- Outline the design of an online travel booking service
- Support creation and booking of full itineraries
  - Flights
  - Hotels
  - Shows
  - Etc.
- Discuss how this might be constructed using technologies you know about
- Discuss problems you might encounter