Remote Method Invocation

Based on Notes by D. Hollinger
Also based on Sun’s Online Java Tutorial
Network Programming Paradigms

• Sockets programming: design a protocol first, then implement clients and servers that support the protocol.

• RMI: Develop an application, then (statically) move some objects to remote machines.
  – Not concerned with the details of the actual communication between processes – everything is just method calls.
Call Semantics

• Method Call Semantics – what does it mean to make a call to a method?
  – How many times is the method run?
  – How do we know the method ran at all?
• RMI does a great job of providing natural call semantics for remote objects/methods.
  – Simply a few additional Exceptions that you need to handle.
  – Objects implementing the Remote interface are passed by reference. Non-remote (serializable) objects and primitive types are passed by value.
Finding Remote Objects

• It would be awkward if we needed to include a hostname, port and protocol with every remote method invocation.

• RMI provides a *Naming Service* through the RMI Registry that simplifies how programs specify the location of remote objects.
  - This naming service is a JDK utility called `rmiregistry` that runs at a well known address (by default).
RMI Adds a few layers

Client App. <-> Server App.
Stubs <-> Skeleton
Remote Reference <-> Remote Reference
Transport <-> Transport
Remote Object References

• The client acquires a reference to a remote object.
  – This part is different from creating a local object.

• The client calls methods on the remote object
  – No (syntactic) difference!
  – Just need to worry about a few new exceptions.
Overview of RMI Programming

- Define an `interface` that declares the methods that will be available remotely.
- The `server` program must include a `class` that implements this `interface`.
- The `server` program must create a remote object and register it with the naming service.
- The `client` program creates a remote object by asking the naming service for an object reference.
Java Interfaces

• Similar to Class
• No implementation! All methods are abstract (virtual for C++ folks).
• Everything is public.
• No fields defined, just Methods.
• No constructor
• an Interface is an API that can be implemented by a Class.
Interfaces and Inheritance

- In Java a class can only extend a single superclass (single inheritance).
- A class can implement any number of interfaces.
  - end result is very similar to multiple inheritance.
Sample Interface

```java
public interface Shape {
    public double getArea();
    public void draw();
    public void fill(Color c);
}
```
public class Circle implements Shape {
  private double radius;
  private Point center;

  // define a constructor and other
  // methods

  // MUST define the methods:
  //  getArea();
  //  draw();
  //  fill(Color c);
}

Implementing an Interface
Server Details – extending Remote

- Create an interface that extends the java.rmi.Remote interface.
  - This new interface includes all the public methods that will be available as remote methods.

```java
import java.rmi.*;
public interface MyRemote extends Remote {
    public int foo(int x) throws RemoteException;
    public String blah(int y) throws RemoteException;
    ... ...
}
```
How the interface will be used

Remote Interface

Your Interface

RemoteServer Class

UnicastRemoteObject

provides methods needed by

extends

Class implementing your remote service

extends

extends

implements

extends
Server Details – Implementation Class

- Create a class that implements the interface.
  - The class should extend UnicastRemoteObject*

- This class needs a constructor that throws RemoteException!

- This class is now used by rmic to create the stub and skeleton code.

*It doesn’t have to extend UnicastRemoteObject, there is another way…
public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {

    public MyRemoteImpl() throws RemoteException {
    }

    public int foo(int x) {
        return (x+1);
    }

    public String blah(int y) {
        return ("Your number is " + y);
    }
}

Generating stubs and skeleton

• Compile the remote interface and implementation:

  > javac MyRemote.java MyRemoteImpl.java

• Use rmic to generate MyRemoteImpl_stub.class, MyRemoteImpl_skel.class

  > rmic MyRemoteImpl
Server Detail – main()

• The server `main()` needs to:
  – create a remote object.
  – register the object with the Naming service.

```java
public static void main(String args[]) {
    try {
        MyRemoteImpl r = new MyRemoteImpl();
        Naming.bind("joe", r);
    } catch (RemoteException e) {
        . . .
    }
}
```
Client Details

• The client needs to ask the naming service for a reference to a remote object.
  – The client needs to know the hostname or IP address of the machine running the server.
  – The client needs to know the name of the remote object.

• The naming service uses URIs to identify remote objects.
Using The Naming service

• `Naming.lookup()` method takes a string parameter that holds a URI indicating the remote object to lookup.
  
  \[
  \text{rmi://hostname/objectname}
  \]

• `Naming.lookup()` returns an `Object`
• `Naming.lookup()` can throw
  – `RemoteException`
  – `MalformedURLException`
Getting a Remote Object

```
try {
    Object o =
    Naming.lookup("rmi://monica.cs.rpi.edu/joe");

    MyRemote r = (MyRemote) o;
    // . . . Use r like any other Java object!
} catch (RemoteException re) {
    . . .
} catch (MalformedURLException up) {
    throw up;
}
```
Starting the Server

• First you need to run the Naming service server:
  
  \texttt{rmiregistry}

• Now run the server:
  
  \texttt{java MyRemoteImpl}
Sample Code

• There is sample RMI code on the course homepage:
  – BankAccount: remote access to a bank account
  – SimpleRMI: remote integer arithmetic
  – AlternativeSimpleRMI: Implementation class doesn’t extend UnicastRemoteObject
  – RemoteSort: remote sort server – uses Java List objects