Remote Method Invocation

The JAVA Distributed Object Environment

RMI – What's the Deal?

- In the world of JAVA…
  - All JAVA programs are collections of objects
  - There exists a hierarchy of authority
  - They must live inside a "JVM" (simulated computer)
- In the real world…
  - We have networks of computers
  - We have collaborative computing
- Why not have collaborative virtual computing?

OOP is Inherently Distributable

- JAVA Object Lifecycles
  - Objects are defined by the programmer
  - Object “templates” are created at compile time
  - Instances of objects are brought into existence by code imperatives at run time
  - The instances are then used for doing “work”
- At any point in time, does a program have any knowledge of “where” an object template or instance lives?

How does it work?

- Start with sockets
- Build a listener (server)
  - Serves up object templates
  - Allows for client to create instances on server
  - Needs a “name space” mechanism
  - Needs a security mechanism
- Build a client
  - Create a library for users to interact with the server

Key Issues that come up…

- Traditional programming methods require that the program has full knowledge of all objects
- Even if we are sharing a single object…
  - What fields does it have that we can access?
  - What methods does it have that we can invoke?
  - Who (which clients) are allowed to see this object?
- If we are sharing many objects…
  - How do we prevent name space collisions?
  - How can we export an object by collection?

How RMI handles the issues…

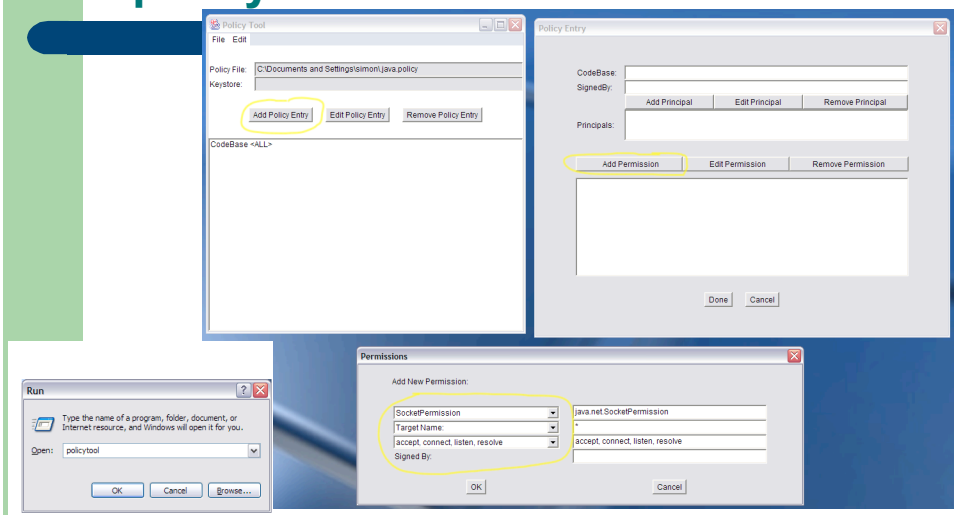
- How do we prevent unauthorized access
  - We have to have a *security policy*
- How do we present object collections
  - Use an object manager with *name binding* services
  - The program is called *rmiregistry*
- What does an object offer (fields/methods)
  - The *interface* for the object must be publicly shared between the client and server

JAVA Security Policies

- The JDK/JRE comes with a policytool
- This is used to create/edit .java.policy
  - In UNIX, it belongs in your home directory
  - In Win9X, it belongs in C:\WINDOWS
  - In WinNT/2K/XP, it belongs in your profile directory

```
grant {
  permission java.net.SocketPermission "*:1024-65535", "accept, connect, listen, resolve";
  permission java.io.FilePermission "<<ALL FILES>>", "read";
};
```


"policytool" under Windows



The RMI Registry

- This keeps a mapping between “collections” names (projects) and actual server classes
- Set your CLASSPATH to your code base
  - UNIX: export CLASSPATH=/home/me/myfiles
  - Windows: set CLASSPATH=C:\mystuff
- Startup the registry
  - UNIX: nohup rmiregistry &
  - Windows: start rmiregistry (keep window open)
- Automatically handled by Forte/Netbeans

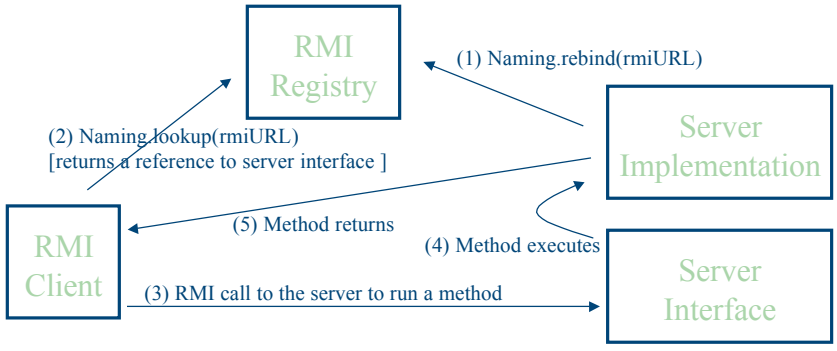
Remote Object Interfaces

- RMI objects must list exported methods into an interface that extends java.rmi.Remote
- The RMI object the must:
  - extend java.rmi.server.UnicastRemoteObject
  - implement TheInterfaceThatExtendsRemote
- All methods that can be called remotely must:
  - Be present in the interface
  - Declared as “throws java.rmi.RemoteException”

The Steps to RMI Outlined:

- Make sure you have a proper .java.policy
- Write your code
- Compile your code
  - javac for your regular code
  - rmic to generate stubs and skeletons
- Start your rmiregistry
- Run your server
- Run your client

The Basic RMI Picture



RMI – Defining the interfaces

```
import java.rmi.*

public interface Server
                extends java.rmi.Remote {

  public void doSomething()
                throws RemoteException;

}
```


Implementing the RMI Objects

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

public class ServerImpl
        extends UnicastRemoteObject
        implements Server { // Server is the interface

 public void doSomething() throws RemoteException {
    // PUT YOUR CODE HERE
 }

 // NEED TO PUT SPECIAL CODE FOR MAIN HERE

}
```

Implementing a Server's main

```
public static void main(String [] args) {

    String rmiName = ”myProjectName";

    try {
      Server theServer = new ServerImpl();  // POLYMORPHISM
      Naming.rebind(rmiName, theServer);
      System.out.println(“bound as " + rmiName);
    } catch (Exception e) {
      System.out.println("Error binding as " + rmiName);
      System.out.println(e);
    }

}
```


Implementing a Client's main

```
public static void main(String [] args) {

    String rmiURL = "rmi://server.com/myProjectName";
    Server theServer = null;

    try {         // NOTICE THE CAST BELOW
      theServer = (Server)Naming.lookup(rmiURL);
      theServer.doSomething();
    } catch (Exception e) {
      System.out.println("Couldn't find " + rmiURL);
      System.out.println(e);
    }

}
```

Compiling/Running a RMI program

- Clients need to have an idea of what's available in order to compile properly
  - JAVAC checks to see if the method names are valid
- Stubs and skeletons for this purpose can be generated with the "rmic" compiler
  - rmic on classes that extend UnicastRemoteObject
- Forte/NetBeans handles this automatically

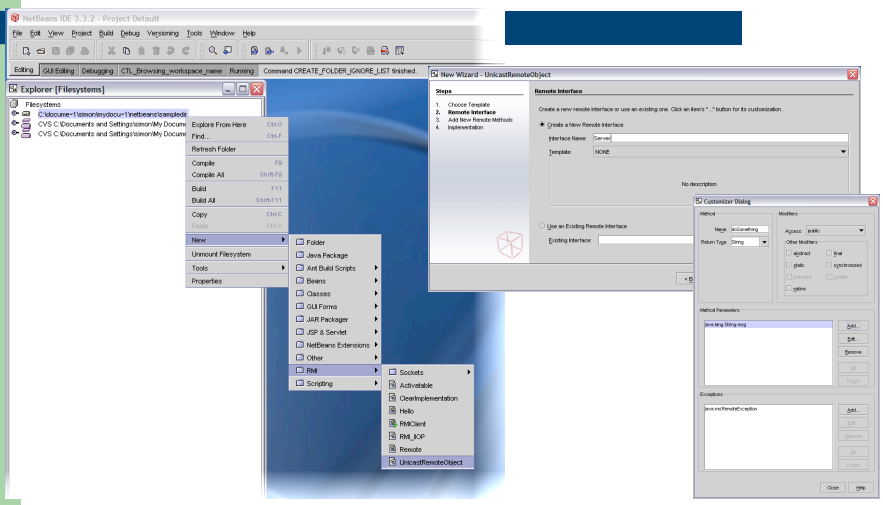
IDE To The Rescue

- There are Forte/NetBeans modules for just about every extension of JAVA, including RMI
  - These modules are now part of the standard install
  - They used to be "pay extra for these modules"
- Use the "UnicastRemoteObject" template for the RMI server
  - This actually creates two files (interface and class)
- Use the "RMIClient" template for the client

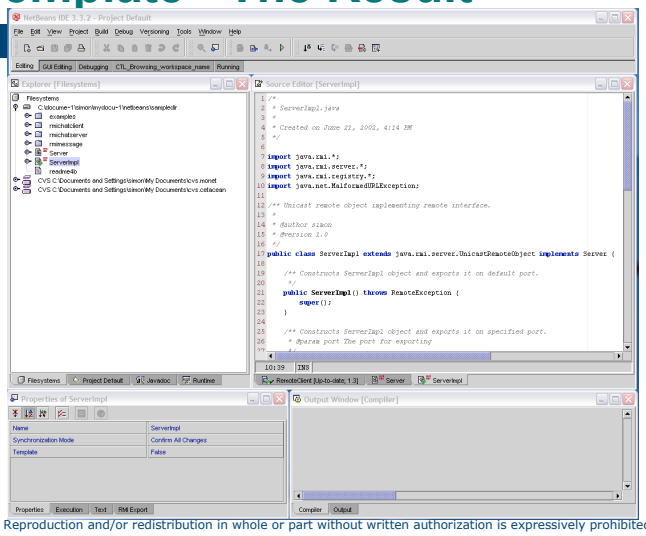
Forte/NetBeans will automatically…

- Synchronize your interface with your implementation on the server side
- Startup an RMI registry on the server
- Provide easy to edit static protected variables for per-project settings (like the RMI name)
- Automatically run rmic to build the stubs and skeletons

Using the RMI template



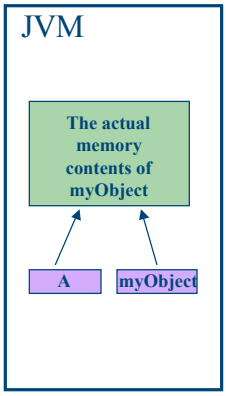
RMI Template - The Result



Call by what?

- In JAVA, everything is call by value
  - This is even true of objects!
  - The reference variables are copied when passed
- This is no different with RMI
  - Parameters and return values are duplicated before they are passed across the network
  - The objects that reference variables point to are *serialized* and transferred
  - A *clone* (that no longer references the original) is then created on the JVM with the implementation

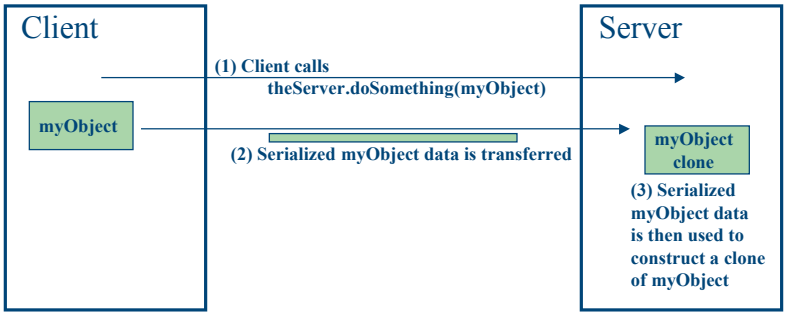
Normal Method Invocation




```
public class whatever {
   public void callMe(Object A) {
      A.doSomeOtherStuff();
   }
   public static void main(…) {
      Object myObject = new Object();
      myObject.doStuff();
      callMe(myObject);
   }
}
```

Reference variables are cloned but point to the same memory location.

RMI Parameter Passing



myObject and myObject clone are **NOT referencing the same object anymore!**

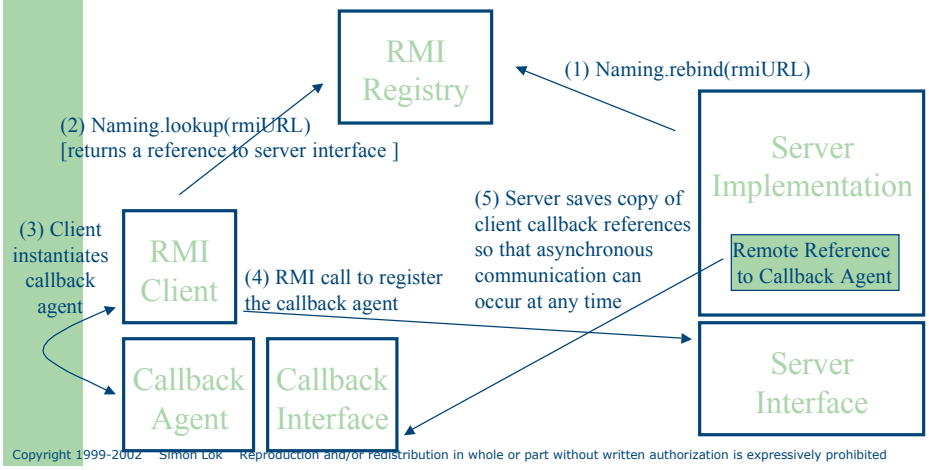
Passing References in RMI

- Classes must implement java.io.Serializable
  - This interface doesn't have any methods or fields
  - It's a marker for the JAVA to know to serialize
- Passing objects by reference is also possible
  - Basically the object must be a RMI "Server"
  - It must extend java.rmi.server.UnicastRemoteObject
  - It must implement an interface that extends Remote
  - Methods that are called in a JVM other than the one the object was created in must be in the interface

Asynchronous Communication

- The client can call the server at any time.
- The server "responds" by executing the implemented method that the client called.
- Let's say the method takes a long time, why can't the server call the client asynchronously?
- Because it doesn't have the a reference to the client to call methods on!
- Setup RMI callbacks to do this.

The RMI Callback Picture



RMI Callback Framework: Server

```
public interface Server {
   public registerClient(Callback theClient) throws RemoteException;
}

public class ServerImpl extends UnicastRemoteObject
                        implements Server {
  Vector theClients = new Vector();
  public registerClient(Callback theClient) throws RemoteException {
     theClients.addElement(theClient);
  }
  public broadcastMessage() throws RemoteException {
     for(int j = 0; j < theClients.length(); j++) {
        ((Callback)theClients.get(j)).recvMessage("whatever");
     }
  }
  public static void main(String[] args) {
     // DO RMI INITIALIZATION AND BINDING HERE
  }
}
```

RMI Callback Framework: Client

```
public interface Callback {
   public recvMessage(String theMessage) throws RemoteException;
}

public class CallbackImpl extends UnicastRemoteObject
                          implements Callback {
  Client myClient = null;  // MUST INIT THIS GUY USING CONSTRUCTOR
  public recvMessage(String theMessage) throws RemoteException {
     myClient.recvMessage(theMessage);
  }
}

public class Client {
  public Client() {
     Server theServer = (Server)Naming.lookup(myURL);
     theServer.registerClient(new CallbackImpl(this));
  }
}
```


In Summary

- RMI is …
  - an extension of sockets that is easier to use
  - a way to make JAVA objects network accessible
  - based on client/server networking principles
- RMI overhead includes …
  - security policy setup
  - setting up the RMI registry
  - creating interfaces, compiling stubs and skeletons
  - special framework if asynchronous communication between server and client is desired