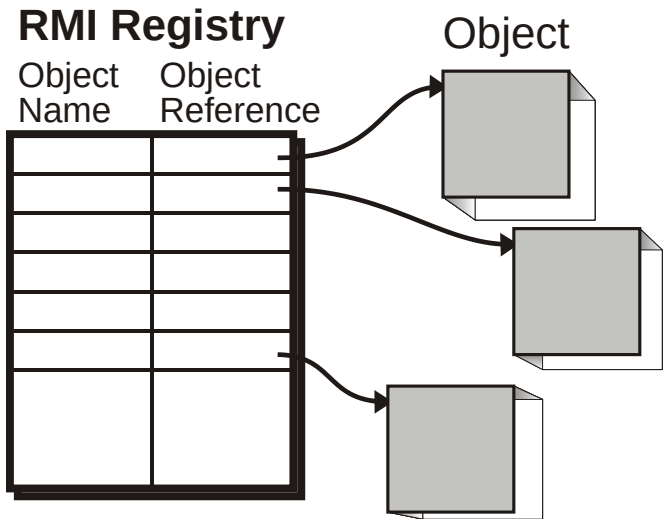
Network Computing Lecture 8

Professor Louise E. Moser

Spring 2012

More on RMI

- The **RMI Registry** is a simple server that runs on the server machine and allows Java server programs on the server machine to register server object names, along with their server object references
- A Java client program wishing to invoke a remote server object supplies a name to the RMI Registry
- The RMI Registry returns a remote reference to the remote server object



The RMI Registry

- You can start the RMI Registry as a server on port 1099 by typing

  rmiregistry &

- Or, you can start an application-specific registry on some other port, say 2164, by typing

  rmiregistry 2164 &

The RMI Registry

- Normally, you would start the RMI Registry from the directory that contains the files for the classes that you declared to be remote, as well as their *Stub* and *Skel* class files
- If you start the RMI Registry from a directory that does not contain those class files, you must tell the RMI Registry where they are
- When you start the Java interpreter, you can use the -d option to supply a URL for the directory containing those class files
  - EX: If *MyServerImpl_Stub* and *MyServerImpl_Skel* are in *foo.bar.edu/Mystuff*, you start the Java system with

    java -d java.rmi.server.CodeBase=rmi://foo.bar.edu/Mystuff
  - This allows the *_Stub* and *_Skel* class files to be retrieved at runtime

Registering Objects in the RMI Registry

MyServerImpl mine = new MyServerImpl();
Naming.bind*("MyName", mine);*

- For a class that is declared to be remote, i.e., extends *UnicastRemoteObject*, you create an instance of that class
- Then, you bind that object to a name in the Registry
  - ***Naming*** is a class in *java.rmi.**
  - *"MyName*" is a short form URL, which expands to, say, "*rmi://foo.bar.edu/Mystuff/MyName*"
- You can bind an object to a name in the Registry only if the object is local to the computer on which the Registry resides
  - ***Naming.bind()*** adds a name to the Registry and binds an object to that name, provided that the name is not already there
  - ***Naming.rebind()*** adds a name to the Registry and binds an object to that name. If the name is already bound, the new object overwrites the old object
  - ***Naming.unbind()*** removes a name from the Registry

Exceptions When Registering an Object

- Examples of exceptions are:
  - ***RemoteException*** - Could not locate the Registry
  - ***MalformedURLException*** - Could not parse the URL string
  - ***UnknownHostException*** - Could not locate the host
  - ***ServerNotActiveException*** - Server not running
  - ***AlreadyBoundException*** - Object has already been bound into the Registry
- There is not much you can do about these exceptions at run time, except to print an error message

Finding Objects in the RMI Registry

```
MyServer mine;
try {
  mine = (MyServer)Naming.lookup(
                              "rmi://foo.bar.edu/Mystuff/MyName");
}
catch ....
```

- Because the RMI Registry is on a remote machine, when you invoke the method *Naming.lookup()*, you are making a remote method invocation
- If you start a registry on a port number other than 1099, say 2164, you need to include that port number in the URL, as in

  *"rmi://foo.bar.edu**:2164**/Mystuff/MyRegistry"*

Finding Objects in the RMI Registry

- You can obtain a remote object of type *Registry*, i.e., a reference to the Registry on the remote computer

```
Registry reg =
        java.rmi.registry.LocateRegistry.getRegistry(
                                        "foo.bar.edu/MyStuff");
...
MyServer mine = (MyServer)reg.lookup("MyName");
...
MyServer oldmine = (MyServer)reg.lookup("MyOldName");
...
YourServer goldmine = (YourServer)reg.lookup("YourName");
```

- Here, "*MyName*" can be just a name in the Registry, rather than a complete URL, because you are already in contact with the Registry on the remote computer

The Sequence of Actions for the Server

1. For a simple Hello application program, the server interface *Hello.java* and the server implementation *HelloImpl.java*
2. Put *Hello.java* and *HelloImpl.java* on the server computer
3. Run *javac* on *HelloImpl.java* to produce *HelloImpl.class*
4. Run *rmic* on *HelloImpl.class* to produce *HelloImpl_Stub.class* and *HelloImpl_Skel.class*
5. Start the Registry on the server computer
6. Start *HelloImpl.class* on the server computer

   (During initialization, *HelloImpl* registers at the Registry)

Thus,

javac -d somedirectory HelloImpl
rmic HelloImpl
rmiregistry &
java HelloImpl

The Sequence of Actions for the Client

1. Program the client *MyClient.java*
2. Put *MyClient.java* and *Hello.java* on the client computer
3. Run *javac* on *MyClient.java* to produce *MyClient.class*
4. Put a copy of *HelloImpl_Stub.class* on the client computer
   If you don't put a copy of the stub to the client computer, the JVM will look for it on the server computer at run time, as discussed in Lecture 7
5. Start *MyClient.class* on the client computer

Thus,

javac -d somedirectory MyClient

ftp HelloImpl_Stub.class

java MyClient

Security

- Because RMI uses program code from remote machines, security restrictions must be imposed
- These restrictions are enforced by a *SecurityManager*
- The *SecurityManager* is invoked at the beginning of *main()* by the command

  System.setSecurityManager(new RMISecurityManager());
- This command is included in the *main()* of the server, and also in the *main()* of the client, to allow dynamic downloading, if the client and the server are **not** applets
  - An applet already has a (much more stringent) security manager, so you do not need to issue this command for an applet
- The *SecurityManager* restricts what a client can do
- It is possible to provide your own custom security manager

Exceptions When Invoking a Remote Method

- When you invoke a method of a remote server object, there are lots of different kinds of exceptions that can be sent back to the client
- You can't do much about most of these exceptions, except to print an error message

```
MyServer mine;
try {
    mine = (MyServer)Naming.lookup(
                                "rmi://foo.bar.edu/Mystuff/MyName");
}
catch (NotBoundException e) {
    out.println("Can't locate MyName in the Registry");
}
catch (RemoteException e) {
    out.println("Can't locate the Registry");
}
catch (UnknownHostException e) {
    out.println("Can't locate the host");
}
```

Serialization and Externalization

- Remote Method Invocation (RMI) provides formats for packaging method invocations, their parameters, and their results, into messages that can be shipped to a remote computer
- If the parameters are simple variables, such as *int*, this is easy
- *Strings* and *Vectors* are variable length and must be sent as a length followed by data
- Some objects are only meaningful locally, for example, *mousex* and *mousey* report the position of the local mouse
  - There is no point in shipping their values to another computer
  - Such values are declared as ***transient*** and are not shipped across the network

private ***transient*** *int mousex*
private ***transient*** *int mousey*

Serialization

- The *Serializable* interface does **not** provide any methods that the programmer can use for shipping objects across the network
- For an object of a class that extends the *Serializable* interface, Java "pickles" the object automatically, i.e., puts it into a byte-code format that allows it to be shipped across the network (or stored in a file on disk)
- To pickle an object, Java uses the code that you wrote for the class

Externalization

- Many objects are declared to contain other objects
  - Shipping such objects across the network can be complex
- One of the primary objectives of RMI is to avoid shipping big objects from one computer to another computer
  - Instead, the big objects stay put and other computers ship method invocations to those remote objects
- However, if you need to ship big objects efficiently, you can use the *Externalizable* interface
- The *Externalizable* interface provides two methods
  - *readExternal()*, which reads from a stream
  - *writeExternal()*, which writes to a stream
- You must define the implementations of these two methods

An Example of Externalization

Consider an ObjectA that contains ObjectB and ObjectC

Data Structure	Externalization
ObjectA { *ObjectB b;* *ObjectC c;* *}*	*writeExternal(Stream out) {* *out.writeHeader("ObjectA");* *b.writeExternal(out);* *c.writeExternal(out);* *}*
ObjectB { *Int x;* *Int y;* *}*	*writeExternal(Stream out) {* *out.writeHeader("ObjectB");* *out.writeInt(x);* *out.writeInt(y);* *}*
ObjectC { *Int z;* *String s;* *}*	*writeExternal(Stream out) {* *out.writeHeader("ObjectC");* *out.writeInt(z);* *out.writeString(s);* *}*

- The formats that Java actually uses are very complicated
  - You should try to avoid needing to know about them

The Customer/Warehouse Example

- This (rather elaborate) example illustrates both RMI and applets (from the book Core Java, Vol. II, pp. 258-263)
  - We will focus on the use of RMI by the applet (see the handout)
- The applet, named *WarehouseApplet*, displays a GUI
  - The customer enters his/her age, sex and hobbies, and then clicks submit
- The applet packages the customer information into a Customer object and invokes the *find()* method of the Warehouse object remotely using RMI
- The Warehouse returns a list of products that match the Customer
- The applet displays the products and the customer selects one
- The applet invokes the *getImageFile()* method of a Product object remotely using RMI and then displays that image

The Customer Class

- The *Customer* class is a simple class that records the customer's age, sex, and hobbies
- The *Customer* class exists on both the client and the server computers
- The directive ***implements Serializable*** instructs the Java compiler to generate code that can pickle an object of this class so that it can be shipped across the network as a parameter or result
- The method *hasHobby()* searches a *String* to find a match

The ProductImpl Class

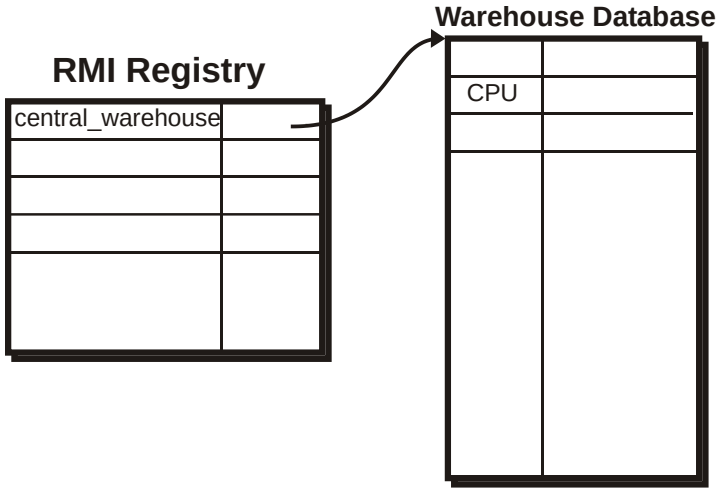
- Each product has a name, an age range, a sex, a hobby, and an image
- The products reside on the server
- The *Product* interface extends *Remote*, and the *ProductImpl* implementation extends *UnicastRemoteObject* and implements the *Product* interface
  - When a product is returned to a remote client, what gets shipped is a reference to a remote product object
- The *Product* interface provides two remote methods *getDescription()* and *getImageFile()*
- The *ProductImpl* class provides a local method *match()* that determines whether a customer matches the product
  - *match()* is local, because it is called only from *WarehouseImpl*

The WarehouseImpl Class

- The warehouse resides on the server computer, along with the products
- The *Warehouse* interface provides a remote method *find()*, which returns a vector of products that match a particular customer as its result
- The *WarehouseImpl* class keeps a private vector of products
- The *WarehouseImpl* class provides a local method *add()*, which can be used to add a product to the vector of products

The WarehouseServer Class

- The *WarehouseServer* class creates an instance *w* of *WarehouseImpl* and registers it in the RMI Registry with the name "*central_warehouse*"
  - Note the *RMISecurityManager* at the beginning of *main()* of the *WarehouseServer* class
  - The *central_warehouse* is registered in the RMI Registry, but the products are not registered in the RMI Registry, because they are in the central warehouse, not in the RMI Registry
- The *WarehouseServer* fills the warehouse with products by invoking the *add()* method of the warehouse



The WarehouseApplet Class

- This code looks complicated, but it is mostly just the GUI
- At the bottom of the first page is the code to obtain a remote reference to the warehouse, with the name "*central_warehouse*", in the RMI Registry
- On the second page is *actionPerformed*, which is invoked by clicking a button on the GUI
  - It creates a new instance *c* of *Customer* to hold the customer's data
  - It then invokes the *find()* method of the warehouse remotely, which returns a vector of suitable products, which the applet displays
- When the user clicks on an item in the list displayed by the GUI, the *itemStateChanged* listener is invoked
  - The applet invokes the product's *getImageFile()* method remotely to obtain an image of the product, and then displays it