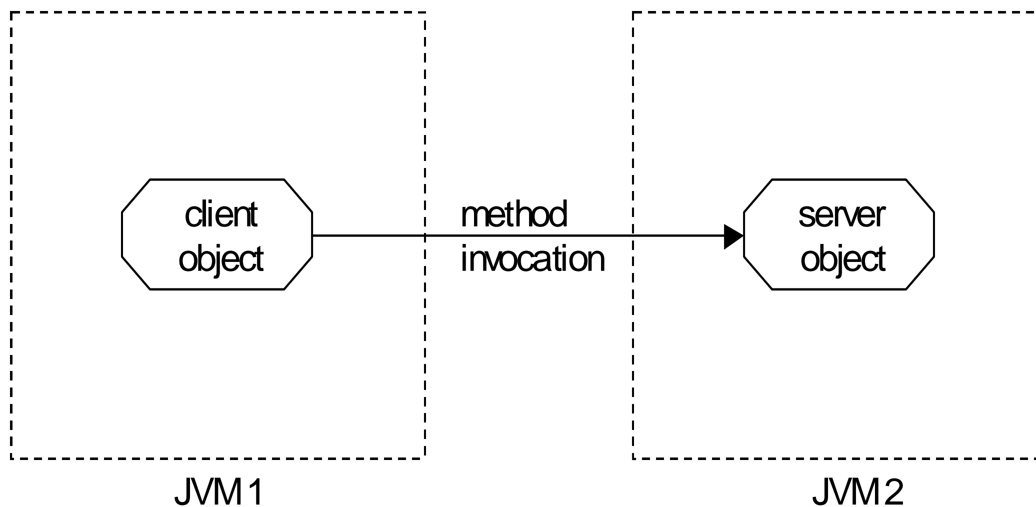
CIS/EE 794R/694R **Spring 2004**
Prof. Paul Sivilotti

Java RMI Example: Hello World

Distributed: Wednesday, Apr 21st

OVERVIEW

This simple example program is a variant on the canonical "Hello World" program, modified for Java RMI. The program is distributed. That is, one object (the "client") invokes a method on another object (the "server") in a *different* Java Virtual Machine. This remote invocation returns a value back to the client, a string (i.e., the string "Hello World"). The client then prints this string to the screen. The server has a single method, `sayHello()`, that takes no arguments, and returns a string. This is the method invoked by the client.



This document sketches a step-by-step procedure for developing this example program. It is a *supplement* to the more complete information at Sun's JDK Web site:

`http://java.sun.com/j2se/1.4.2/docs/guide/rmi/getstart.doc.html`

(See the class web page for a link to this resource.) The only substantial difference between this guide and what you will find on the web at Sun's site is that in this document, the client is a stand-alone Java program, rather than a Java applet. The use of applets is not the point here.

In this document, we assume the user developing the server has login id "brutus" and the user developing the client has login id "buckeye". We also assume that the server is being run on a host named "beta.cis.ohio-state.edu". Presumably neither of these

assumptions is strictly true in your particular case. So, you need to modify the relevant steps to take into consideration your particular login id and hostname.

BEFORE YOU BEGIN

Make sure you are running Java 1.4. You can do this in the CIS environment by subscribing to JDK-current (ie type "subscribe" and follow the menu to select JDK-current). After subscribing, you need to login before the changes take effect. You can check that you are correctly subscribed by typing "which java". You should see something like: /usr/local/j2sdk1.4.0_01/bin/java.

SERVER SIDE DEVELOPMENT

1. Implement the Server

   1.1. Write the remote interface

   - Create the directory: ~brutus/src/examples/hello
   - In this directory, define the remote interface Hello.java
     (see handouts from class website)

   1.2. Write the implementation of the remote object

   - In the same directory, write the remote object implementation, HelloImpl.java (see handouts from class website)
   - Note the implementation inherits from UnicastRemoteObject, which implements basic object functionality for remote objects.
   - Also note that a constructor always makes an implicit (or explicit) call to its super class (UnicastRemoteObject) constructor, which exports the object (i.e., makes it available for RMI calls). Because of this implicit call, the constructor must explicitly allow RemoteException to be thrown.
   - In the main function, a security manager is created and installed, which defines what actions are permitted by, for example, the Java class loader responsible for dynamic loading of code.
   - Also in the main function, after creating an instance of the server object, this implementation instance is registered in the RMI registry. The usual format for the string used to identify the implementation instance is something like:
     ```
     //myhost:1234/HelloServer
     ```
     but, implicitly, the local host and port 1099 are used in the first two fields.

2. Compile the Server Source

   2.1. From ~brutus/src type
   ```
   javac examples/hello/HelloImpl.java
   ```
   This creates HelloImpl.class in the src/examples/hello directory.

3. Generate Stubs (and Skeletons)

   3.1. Choose a directory under your WWW directory in which network-accessible classes will be placed
   ```
   mkdir ~brutus/WWW/classes
   ```

3.2. From ~brutus/src type

```
        rmic -d ~brutus/WWW/classes
examples.hello.HelloImpl
```

This creates the necessary stubs and skeleton files for the remote object. It also creates a subdirectory off of WWW/classes (specified by the –d flag) called examples/hello (specified by the package name of the target java file). In this new subdirectory, it places the generated stubs and skeletons.

4. Publish the Required Files by Making Them Network Accessible

4.1. The client must be able to obtain the interface. This is usually done with a jar file, but for this example it is sufficient to place the class files in a place that is accessible by the client.

4.2. Place the class file for the remote interface (Hello.class) in the directory created in the previous step

```
     cp ~/src/examples/hello/Hello.class
~/WWW/classes/examples/hello
```

4.3. Make sure the files and directories are publicly accessible (readable for directories and readable/searchable for directories)

```
     chmod -R og=u ~/WWW/classes
     chmod -R og-w ~/WWW/classes
```

5. Write a Security Policy File

5.1. Create a text file in ~brutus/src (typically called policy.txt) that specifies what downladed code can do

5.2. The security manager will read this file to determine whether requested actions are permitted.

5.3. For examples, see policy.txt and policy2.txt at handouts under class website

6. Start rmiregistry

6.1. Make sure CLASSPATH does not include the directory where the code that will be downloaded by clients resides (ie ~brutus/WWW/classes). This information will be specified when the server is started.

```
        unsetenv CLASSPATH
```

6.2. Start up the process

```
        rmiregistry &
```

By default, this runs on port 1099. You can choose a different port with, for example:

```
        rmiregistry 2000 &
```

7. Start the Server

7.1. From the ~brutus/src directory, type:

```
        java -classpath .:$HOME/WWW/classes
             -Djava.rmi.server.codebase=
http://www.cis.ohio-state.edu/~brutus/classes/
             -Djava.rmi.server.hostname=
beta.cis.ohio-state.edu
```

```
-Djava.security.policy=$HOME/src/policy.txt
examples.hello.HelloImpl
```

- The classpath flag points to the stubs and skeletons. Notice the space after classpath (no =).
- The codebase property tells clients where downloadable files are. Don't forget the trailing "/"! It is required for clients to be able to find the downloadable files. Also, do NOT use a space after the = sign.
- The hostname property is used to guarantee that java can determine the local hostname. If the java run-time can correctly make this determination , it is not needed. Again, do NOT place a space after the = sign.
- The policy property specifies the location of the policy text file for the security manager. Again, there is no space after the = sign.

Common Errors Encountered During Server Side Development

1. Port Already in Use
   - When: starting rmiregistry
   - Error Message Generated:
     ```
     RegistryImpl.main: an exception occurred: Port already
     in use: 1099
     ```
   - Problem: The specified port is already in use, so the rmiregistry process cannot make use of it. There may be another rmiregistry processes already running or something completely different that happens to use the port you requested.
   - Solution: Make sure it is not your own rmiregistry process that is already running by typing ps (or ps –af) to look at the current processes. If it is, you can kill this rmiregistry with kill –9 <process-id>. If someone else's process is already occupying the port you requested, restart rmiregistry with a new port:
     ```
     rmiregistry 1150 &
     ```
     If you change the port of the rmiregistry process, remember to change the registration of the Hello remote object (with Naming.rebind) to reflect this new port. For example:
     ```
     Naming.rebind ("//:1150/HelloImpl", obj);
     ```
2. java.lang.ClassNotFoundException
   - When: attempting to bind or rebind a remote object to a name in the registry.
   - Problem: the registry can not locate the remote object's stubs or other classes needed by the stub.
   - Note: the remote object's stub implements all the same interfaces as the remote object itself, so those interfaces, as well as any other custom classes declared as method parameters or return values, must also be available for download from the specified codebase.

- Possible causes:
  1) The codebase property was misspelled (ie java.rmi.server.codebase).
  2) Incorrect codebase given (e.g., forgetting the trailing "/", misspelling the path to the classes, or having a space after the "=" sign).
  3) Forgetting to move all the necessary ("published") classes to the location specified in the codebase.
  4) Forgetting to make all the "published" classes world-readable.
  5) Forgetting to make the codebase directory world-accessible (x permission for directories).
  6) Rmiregistry was finding the files in its CLASSPATH.
- Solutions:
  1) Check misspelling of the property name.
  2) Check the spelling and trailing "/" of directory given as codebase. Make sure there is no space after the "=" sign.
  3) Check that all required files are in the codebase directory.
  4) Check that the files in the codebase directory are world-readable, using ls –l. Add permissions as needed with chmod.
  5) Check that the directory tree for codebase is world-accessible, using ls –l. Add permissions as needed with chmod.
  6) Make sure classpath is unset before starting rmiregistry.

3. Connection Refused
   - Problem: Server is not bound to the rmiregistry. (rmiregistry is not running)
   - Solution: Check that rmiregistry is, in fact, running. Check the port is available to run rmiregistry. (eg. If your port is not available, exception will occur).

CLIENT SIDE DEVELOPMENT

1. Write a Client that Uses the Remote Interface

   1.1. Create a directory for the client source code
   ```
   mkdir ~buckeye/src/examples/helloclient
   ```

   1.2. Obtain the remote interface from the server

   - The remote interface will often be contained in a jar that can be downloaded and upacked. Regardless, the client must obtain the class file in order to write code that imports (and uses) the remote interface. Place this remote interface (Hello.class) in a directory that is visible to the client code. The directory ~buckeye/src/examples/hello is a good choice.

   1.3. Write the client implementation, say Client.java.
   (see handouts under class web site)

2. Compile the Client

   - From `~buckeye/src`, type:

     ```
     javac examples/helloclient/Client.java
     ```

   2.1. In general, you may need -classpath flag to indicate where to find examples.hello.Hello.class file. For example:

     ```
     javac -classpath $HOME/remotestuff/
     examples/helloclient/Client.java
     ```

     But in our case here, we placed Hello.class under the same directory (`~buckeye/src`) as Client.java, so there is no need for this –classpath information.

3. Run the Client

   ```
   java -classpath .:$HOME/src
        -Djava.security.policy=$HOME/src/policy.txt
        examples.helloclient.Client beta.cis.ohio-
   state.edu
   ```

Common Errors Encountered During Client-Side Development

1. ClassNotFoundException

   1.1. When: When attempting to lookup a remote object in the registry.

   1.2. Problem: If you receive this exception in a stacktrace resulting from an attempt to run your RMI client code, then your problem is the CLASSPATH with which your RMI registry was started.

2. Connection refused

   - Error message generated:
     java.rmi.UnmarshalException: Return value class not found; nested exception is:
     java.lang.ClassNotFoundException: MyImpl_Stub
     at sun.rmi.registry.RegistryImpl_Stub.lookup(RegistryImpl_Stub.java:109
     at java.rmi.Naming.lookup(Naming.java:60)
     at RmiClient.main(MyClient.java:28)

   - Problem: No rmi registry (eg. Client trying to bind to wrong host, port or both)

   - Solution:

     2.1.1. Use the ps command to verify that rmiregistry is running on the server machine

     2.1.2. Verify that the port used by the client corresponds to the port with which the rmiregistry was started (default 1099)

     2.1.3. Verify that the client is using the correct hostname to locate the rmiregistry process.