1. Java Remote Method Invocation (RMI)

- This lecture was developed by Russ Tront, Instructor, School of Computing Science, Simon Fraser University
- email: tront@sfu.ca
1.1 Introduction

The Java language has extensive data communications facilities built right into the language. You don’t just call function libraries provided by your operating system. Instead there is extensive language support for reading and writing to sockets, for calling member functions of Java object instances running in Java virtual machine processes on other machines, and for downloading code from other machines. This downloading is not just of applets (though that is what Java’s distributed computing features are perhaps most well know for). In addition:

- there is support for downloading proxy stubs which a client program needs to make calls to member functions of object instances running on other machines. The stub marshals the parameters into a packet to send to the other machine.
- there is extensive support for security checks at all points.
- there is even support in Java 1.2 and especially 1.3 for interoperating with a competing distributed object technology called CORBA.

This section of the course will not discuss security except what is necessary to get our demonstration programs running (not a small feat under the new Java 1.2). It will mainly introduce you to the basic concepts and practices of Remote Method Invocation (RMI).

One of the interesting things about remote procedure calls is that your calling program is NOT statically ‘linked’ (in the ‘compile’, then ‘link’, then ‘run’ sense) to the called code. Instead, at run time, dynamic linking is done to determine exactly where within the destination program’s code is the entry point of the called function.

One of the advantages of dynamic linking is that you can update either the caller or the callee, without needing to re-statically link them together.

An Aside: How would you like to have to buy new copies of all your applications when you installed a new version of Windows (because certain functions were now located at a different byte address within Windows)? Or visa versa. Fortunately, most operating systems interact with their applications using a very special dynamic link mechanism peculiar to that brand of CPU.

Unlike C/C++, Java never uses static linking but instead only uses dynamic linking even for ‘within program’ calls. And because of Java’s portable nature, this dynamic linking is not CPU dependent. Because remote function calls requires dynamic linking, and because Java already used dynamic linking, it was quite easy to add the RMI feature to the Java technology.
1.1.1 The Basic Architecture of RMI

There are 4 major software parts needed to call the member function of an object instance residing in a different Java Virtual Machine (JVM) than the calling object. (Usually this different JVM is on a different CPU somewhere else on the network, but for testing purposes, it may be just another JVM on the same machine. Note that when you start a program using the command:

> java myprogram

this starts up a new virtual machine and starts the main method of the class 'myprogram' running within it. If you do this twice, in two different command windows, you will have two JVMs running on your machine at the same time, for testing purposes.)

The calling code (be it an instance method or just a static method like main()) is in the same program/VM as the stub (also called proxy) instance. This ‘proxy’ instance is a ‘stand-in’ for the actual remote instance. When you call a remote instance, RMI magically arranges it so that you actually call the local stub instance, which has all the same methods as the remote object! Each stub method knows how to send (also called marshal) all the parameters for the call out over a communications ‘socket’ across the network to the skeleton. The skeleton knows how to unmarshal the incoming data. The skeleton is running in the same program as the remote object, and it can thus makes an actual (up) call using the unmarshalled parameters to the actual remote object.

When the remote method is done, it does a normal function return to the skeleton. The skeleton then marshals any return parameters and values in preparation for transmission back to the stub. The stub, once it receives the return values, then does a normal kind of function return to the calling code. The calling code is halted until the complete remote call mechanism finishes. This is therefore called a ‘synchronous’ call: the caller proceeds only when (in synchronicity with) the remote procedure returning.

Notes:
- The program that the remote object is running within is often called the ‘server’.
- The remote object instance is sometimes called the ‘servant’.
- If you know FOR SURE that all Java technology used in your system (including in your web browser if you are using applets) is JDK 1.2 or later, you do not need the skeleton. Java 1.2’s
VM will apparently synthesize a skeleton on the fly for you at the initial time of the receipt of the marshalled data. Java 1.2’s RMI Compiler ‘rmic’ by default still produces a skeleton anyway, and that is likely wise until we get into the new millennium.

- In the competing COM technology, Microsoft uses the single term ‘proxy’ for what in Java is called either the ‘proxy’ or the ‘stub’. They do this so that they can unfortunately use the term ‘stub’ for what in Java and in CORBA technologies is called the ‘skeleton’!

OK, so now you have seen the basic entities involved in, and the step involved in, a remote procedure call. In particular, this is a object-oriented remote procedure call, because the stub, skeleton, and servant are object instances. The caller could be either an instance function or just a static function, but the called function cannot be static. The called function MUST be a method belonging to an object instance!

1.1.2 The Server

Server programs in Java 1.1 and earlier must be running BEFORE the remote call arrives! This is not unlike how socket servers work (unless running the Unix super server named ‘inetd’). You should be warned that if a computer is to have many server programs and servant objects running, and each is only called once a week, it is a terrible waste of RAM resources and TCP ports. It is better if seldom called servers and objects are started/activated only when a call arrives, and are otherwise in ‘stasis’ on disk.

In Java 1.2 there is a brand new feature called Remote Object Activation (ROA) which will allows you to have your servers and servants in stasis on disk between calls. Of course you have to wait a considerable fraction of a second for them to activate when they have not been recently used. I won’t discuss ROA much (for more information consider The Activation Tutorial at http://www.javasoft.com/products/jdk/1.2/docs/guide/rmi/
1.1.3 The Registry

Finally, an essential part of the Java RMI system is the rmiregistry.exe program. This is not to be confused with a Microsoft operating system registry (though the latter is used by Microsoft’s competing DCOM technology for object registration in addition to normal operating system stuff).

The RMI registry maintains a list of ‘advertised names’ for remotely-available objects on that machine. It also stores other information about each (e.g. listening port for each). The CORBA technology calls this the ‘Naming Service’ rather than the registry service. As mentioned above, Microsoft uses the operating system’s registry for DCOM object lookup.

When a servant object starts, because it inherits functionality from the java.rmi.server.UnicastRemoteObject class, it automatically grabs a random TCP port whose number is >1023 and starts listening for incoming calls. In order for a client to find out which port a servant instance is listening on, the servant must notify the registry of its presence and port number. This is done by the server (or servant) using a call to the function:

```
Naming.rebind("advertised_name",
               servant_variable );
```

This notifies the registry that a particular servant is to be advertised with a particular name string. Any client object which knows that name string can inquire about it to the registry using the following call to the java.rmi.* package:

```
Naming.lookup(
               "rmi://hostname:1099/advertised_name");
```

This function returns a ‘remote reference’ to the remote servant object which will allow the client to find out everything it needs to know about the servant: its port, the server program process that it is hosted within, and even the stubs that it needs to actually call the servant instance. In fact, the stub’s code can even be AUTOMATICALLY downloaded to the client if the client doesn’t have them (even if the client is an ocean away!). It can then dynamically link to those new stubs at run time! This capability comes from the RMIClassLoader.

The only key thing that the client programmer must have ahead of (run) time is the source code for the remote interface definition.

In a Java RMI client, once the remote reference and stub code is in hand, the client can then call the remote object’s remote methods exactly as if the remote object were local! Note that it cannot call any of the remote object’s non-remote methods, nor the remote object class’s static methods.

For correct execution, things must thus generally be started in this order:
1. rmiregistry.exe
2. java theServerClassName
3. java theClientClassName
Note that either of the server or the servant can do the actual rebind. For example, sometimes the servant’s constructor is programmed to register itself!

1.2 Interfaces

Java provides a very nice way of specifying the functions a class must implement without defining the class itself. I.e. without specifying how the class is implemented, what is the class name, what classes it inherits from, and what statements it has in its function bodies. An interface is like an abstract class, except that a class that implements an interface does not need to inherit from it (it only has to state that it implements that interface).

Sun states that interfaces are useful for the following:
- Capturing member function similarities between ‘unrelated’ classes without artificially forcing an inheritance relationship on or between them.
- Declaring methods that one or more classes are expected to implement.
- Non-inheritance-based polymorphism.
- Revealing an object's programming interface without revealing its class.

There are a few other differences between an abstract class, and an interface.
- An interface cannot have member attributes, except public constants. An interface generally only advertises functions!
- The method bodies of ALL methods must be absent. There should not even be empty braces{} for the body, nor like in C++ a ‘=0’ present.
- Most methods and constants should be declared public, because the purpose of an interface is to make these things publicly
specified! The only exception is if the users of those non-public members are in the same package as the server (rare, as why would you put them in the interface then!).

Java RMI uses interfaces to specify what instance functions are available to remote clients. In particular, it is not possible for a client to access a remote instance’s member attributes directly. This is because a remote object’s RAM address, and the addresses of its member attributes, have no meaning to a client program running on another machine anyway. Since distant programs thus only communicate by function call, a remote instance could provide get() and set() remote functions for each attribute that it wants remote clients to be able to indirectly access.

Here is an example of a (remote) interface:

```java
//DownCounter Interface
public interface DownCounter
    extends java.rmi.Remote{
    public void setCount(int newValue) throws java.rmi.RemoteException;
    public int decrement() throws java.rmi.RemoteException;
}
```
1.3 Using an Interface

Students are often not familiar with how to use an interface. Here is an example:

```java
DownCounter d = new Airplane();
```

The variable `d` is a reference to some kind of object that implements the above two functions in the DownCounter interface. This variable could, as shown above, refer to an instance of an Airplane class which has an altitude that can be set and decremented, OR equally validly (as we shall see later) refer to an instance of the DownCounterImpl class. The `d` variable can be thought of polymorphically, even though interestingly the various kinds of things it might refer to at different moments during execution are NOT related by inheritance! The programmer is nonetheless assured that, no matter where the reference was obtained (do you trust your sources?) and irrespective of the underlying instance type, he/she can still execute a statement like:

```java
d.decrement();
```

1.4 Remote Interfaces

‘Remote’ interfaces have special criteria which applies to them over and above that of regular interfaces. As shown in the DownCounter interface in the previous section:

- The remote interface itself must be declared public (assuming that the client is not in the remote object’s package).
- The remote interface must extend the standard java.rmi.Remote interface (note that interfaces can ‘inherit’ from other interfaces, but this process just adds all the member functions together into one big interface). For RMI, interface Remote is currently just be an empty interface serving as a remotable flag. (Note: In future there may be remote methods that every RMI servant must have, but that you as an application programmer needn’t be aware of. Since you normally write your servant class by inheriting from a Java-supplied base class, Sun would supply the declaration of such future functions in the Remote interface and the function bodies in the standard base class.)
- Every member function must declare that it could throw a java.rmi.RemoteException (or a super class of that exception type). Also, every member function of an interface is by default individually public and it is NOT necessary to declare each one public as I did in the example above (in fact, Sun Microsystems is now starting to discourage it).

In addition, the parameters of each remote method must satisfy the following criteria:
- Must be either a primitive type like int, or an object that is declared to implement the ‘Serializable’ interface, or a remote object. Most classes fall under one of these criteria except those
classes (like a file handle) which are unique to a particular operating system or CPU. These criteria allow the parameters to either be marshaled into a serialized stream of bytes and sent over the network to the called remote object. Or in the case of a remote object parameter, just its reference is serialized rather than the whole object.

- The type of any parameters that are themselves remote objects (or parameters that are local objects containing embedded references to remote objects), must have those entities be declared as their interface type and not as the actual class type that implements that (other) remote interface. (We will see the difference between an interface type and an implementation class type later).

Note that if you want to test your server with someone else’s client, either you the servant author, or the both of you should write one shared interface file, then put that on each of your machines to allow independent development of your server and client code.

### 1.5 Servant Class and Server Program

The servant instance must be an instance of a particular class. The DownCounter interface name is not a class name, and is not a class (obviously, because it has no code that implements the member implementation functions). Though it is common to write both a servant implementation class and also a separate server program class (containing the main() function), it is equally common in simple cases to merge the two into one class as shown below.

```java
// DownCounterImpl.java, a RMI servant/server.
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class DownCounterImpl      
extends UnicastRemoteObject        
implements DownCounter {
  private int theCount;
  public DownCounterImpl()throws RemoteException{
    super();   //optional; will be called anyway.
    theCount = 10; //default initial count.
  }
  public void setCount(int newValue) throws 
    java.rmi.RemoteException {
    theCount = newValue;
  }
  public int decrement() throws 
    java.rmi.RemoteException {
    theCount--;
    return theCount;
  }
```
public static void main(String args[]) {
    // This is the main program of the server.
    // First, create and install security manager
    // which is required for JDK 1.2+
    System.setSecurityManager(
        new RMISecurityManager());

    try {
        DownCounter dcInstance =
            new DownCounterImpl();
        System.out.println(
            "DownCounter instance now listening.");
        Naming.rebind(
            "AdvertisedDownCounterName",
            (DownCounter)dcInstance);
        System.out.println(
            "DownCounter instance registered.");
    } catch (Exception e) {
        System.out.println("Exception: " +
            e.getMessage());
        e.printStackTrace();
    }
} // end of main.
} // end class.

You can name your server program class anything you want, and if you have a separate servant class, you can name it anything you want. Most programmers in creating the server class append either “Impl” or “Server” to the end of the remote interface’s name. If your servant and server are separate classes, then the servant often uses the suffix “Impl” or “Servant” and the main class has the suffix “Server".
The remotely-callable servant class must:

- declare that it implements at least the one remote interface.
- inherit its remotable nature from class UnicastRemoteObject (this class knows how to talk to skeletons (or may actually have the skeletal functionality in it)).
- define a no argument constructor. It is optional that this constructor call super(), as the super class constructor will be called in any case. But many programmers put the call to super() in to remind them that this is where the remote object actually gets bound (in the socket sense) to a listening port and starts listening. Unless you request a special port, a random one above 1023 will be chosen.
- the constructor must declare that it can throw a RemoteException.
- the servant class must provide implementation code for at least the methods that can be invoked remotely.

The main function of the server class:

- must be started with the -Djava.rmi.server.codebase= command line parameter exactly correct so as to grant the registry and the client access to download the stubs. You may get a ClassNotFoundException if this is not correct. This may happen even if the client and server are on the same machine, but often happens once you try to run them on separate machines.
- must create servant instance(s), and have it/them bind to an advertised name(s) in the rmiregistry. Generally, you either assign the created instance to a variable with the remote interface type, or cast it to the remote interface type when registering it (you do not need to do both, as I did above purely for demonstration purposes). That remote interface type is all the registry and client really need to know about the servant and its stub; they don’t know the servant’s actual class! A client could actually call two different servants which are of different classes yet support just the one remote interface definition! Note that remote reference variables somehow contain the IP address and TCP port number of where the referred-to object’s skeleton is listening.
- for Java 1.2 and later, create and install a security manager. In Java 1.2, security can be implemented in a very precise and fine grained manner exactly as desired by the programmer. The default behavior of the RMISecurityManager class is very restrictive (similar to an applet security manager). In fact, it will not even allow the server/servant to get a socket port on its own machine. This will be discussed further in the section on running the server.
1.6 The RMI Client

The client can be named anything you want. It needs access when compiling to the remote interface file (either put a copy of the interface file in the same directory as the client, or use the Java -classpath feature to point to it). The client also needs a security manager, because it does not want to let any downloaded stubs do bad things if they contain viruses sent by a malicious server.

The client uses the static Naming.lookup() call to ask the registry about the advertised name. The request is often in the form of a URL style name. The first (optional) part is the protocol: rmi. The next part contains the machine name where the correct registry is located (the server’s machine name). If the registry and server are on the client machine, you might be able to use the word “localhost” instead of the machine name (should work even if the local machine doesn’t have a name). The machine name may default to the local machine if the machine name is absent. Finally, the third part is the advertised name you are looking for.

```java
// DownCounterClient.java
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class DownCounterClient
{
    public static void main(String args[])
    {
        // Create and install the security manager
        System.setSecurityManager(
            new RMIInvocationHandler());
        try
        {
            DownCounter remoteCounter =
                (DownCounter) Naming.lookup(
                    "rmi://" + args[0] + "/"
                    + "AdvertisedDownCounterName");
            System.out.println(
                "DownCounter found in registry.");
            System.out.println(
                "Beginning remote invocations.");
            for (int i = 0 ; i < 5; i++)
            {
                System.out.println("Returned value = "
                    + remoteCounter.decrement());
            }
        } catch(Exception e)
        {
            System.err.println("System Exception"+ e);
            e.printStackTrace();
        }
    } //end of main.
} //end of DownCounterClient.
```
1.7 Compiling and Generating Stubs/Skeletons

You should be able to compile these programs from a DOS or NT Command Prompt window, by typing:

```
> javac DownCounter.java
> javac DownCounterClient.java
> javac DownCounterImpl.java
> rmic DownCounterImpl
```

The last command runs the RMI compiler on the servant .class (not .java) file. This tool notices that the servant .class file implements a remote interface, and thus **rmic in turn generates two extra .class files:**

- DownCounterImpl_Stub.class and
- DownCounterImpl_Skel.class.

Note that you don’t get to see the source files for the stub and skeleton classes unless you specifically ask the rmic command to generate them (you don’t need the sources anyway, but you might want to peek at them out of interest!).

(Note that rmic uses the classpath environment variable to find the DownCounterImpl.class file. So unfortunately you may need to add your working directory to your classpath so it can find the necessary file. Also note that rmic is fussy about package subdirectories, and also that rmic’s behavior with respect to paths and packages is changing in JDK 1.3 to make it more similar to the javac compilers assumptions about search directories.)

You will see in the following section that the stub class file should be made available by the server for download to the client (which might not have a copy).

Rather than using command line tools like I did above, you can use a Java development environment like Borland’s JBuilder, or Semantic Visual Café, or others. But beware: Microsoft’s Visual Studio/Visual J++ does NOT support RMI. Microsoft is being sued by Sun for their lack of support for the ‘complete’ Java system. Microsoft does not support RMI or the JFC Swing GUI classes because they compete directly with their DCOM and WFC product features. I have read that you can simply add the RMI classes into Visual J++, but they do not come with J++.

I have found a few freeware development environments that sit right on top of Sun’s raw JDK. I like these because you can get the latest Java features very quickly, rather than having to wait for Borland or Semantic to modify their development environments. See for instance:

- TextPad from www.textpad.com
- Kaw from www.tek-tools.com
- Jpad Pro from www.modelworks.com

The first is the simplest, a basic text editor that has two menu items: ‘Compile Java’ and ‘Run Java’. It also reports the list of errors from a compile, and double clicking on an error message will take you to the exact line in the file you just compiled. See the Appendix of this section for more info on setting up TextPad.
1.8 Running the Programs

1.8.1 Configuring the Security Manager

First, to run RMI in Java 1.2 or later, you will need to include calls in your programs to install a security manager. The security manager checks for policy files to tell it what is permissible.

The default behavior of the RMISecurityManager class is very restrictive (similar to an applet security manager). In fact, it will not even allow the server/ servant to get a socket port on its own machine. To allow this, you must additionally edit the file jdk1.2.2\jre\lib\security\java.policy (see the note below) to add the permission:

```java
grant {
    permission java.net.SocketPermission "*:1024-65535", "connect,accept";
    // permission java.io.FilePermission "d:\Russ\Java\test\-", "read";
    permission java.net.SocketPermission "*:80", "connect";
};
```

- The first permission grants the server permission to accept incoming connections on a port and to also connect out (like to the registry) on any port above 1023.
- The second permission is necessary to allow the registry and client to get access to where the stubs are located.
- The last permission is commented out, but gives an example of allowing the registry and the client to instead get your stubs from a web server also running on this computer but on port 80.

When starting your server, you must use a:
-Djava.rmi.server.codebase=
clause to indicate to the server where you have stored the stubs that it might have to give out to calling clients.

NOTE 1: The java security policy file is located by looking in
{java.home}\lib\security\java.policy
where {java.home} on Windows NT is a Windows registry variable. Contrary to the documentation from Javasoft at http://java.sun.com/products/jdk/1.2/docs/guide/security/PolicyFiles.html this registry variable is NOT the location that the JDK is installed, but is instead the location of where the Java Run Time Environment (JRE) is installed. ONLY if the JRE is not installed does it then look at the {java.home} registry variable associated with the JDK. Not only that, but it ADDITIONALLY then inserts a subdirectory /jre between the JDK’s {javahome} variable and the /lib/security/java.policy. I.e. it therefore looks in:

```bash
{java.home}\jre\lib\security\java.policy
```

To see whether you have a JRE registry variable, start the Windows NT regedit command line program, and look for HKEY_Local_Machine\Software\Javasoft\JRE\1.2\{java.home}.
If you can’t find that, look for:
HKEY_Local_Machine\Software\Javasoft\Java Development Kit\1.2\{java.home}.
and to the latter append a \jre before appending the \lib\security\java.policy.

NOTE 2: If instead you want to give each program a different policy, you can specify the policy file location on the command line when starting a Java a program. Use the:
–Djava.security.policy=someDirectory
command line option. By the way, this option only works if the
policy.allowSystemProperty is true (the default). This latter
property in turn can be found in the file:

{java.home}/lib/security/java.security
or wherever the jre can be found.

Note 3: Interesting, Java also allows different users to have
different policies based on {user.home}.

1.8.2 Having a Running Communications Stack

Second, to run RMI, you have to have a TCP communications
stack installed on your computer. If you have a network
connection, this is likely the case already. If you have an old
Windows 95 machine that you have never connected to a modem
or network, you may have to install TCP.

In addition, you have to have an ‘active’ connection or RMI will
not work. I.e. the TCP stack has to be active. The simplest way
to do this is to be connected to a network or to dial out to your ISP
with your modem for the moments that you are actually running
the programs. You will likely need to do this even if you are
running both the client and the server on the same machine!

There are ways around this. If you are running Windows NT you
can configure a loopback adapter pseudo-device. See the Java
Developer Forum Archives and search for “isolated NT
Workstation” or go directly to:
http://forum.java.sun.com/forum?13@23.q9swaDA5abY^0@.ee7
871a/0
Alternately, if you have a spare serial port on your PC, see the
http://java.sun.com/products/jdk/1.2/docs/guide/rmi/faq.html
1.8.3 Starting Your Programs

Create three separate DOS or NT Command Prompt windows. In the first, simply do the following:

>set classpath=
>rmiregistry

Don’t be surprised if this daemon produces no output.

NOTE: Even though rmiregistry is not a Java program (it is a native executable), when you start rmiregistry, there:

• Must be NO classpath environment variable.
• Or, the classpath must NOT any classes (e.g. stubs) you want downloaded to your remote client.

This makes sure that the rmiregistry does not use a classpath to find your server’s stubs, but instead uses the codebase that you indicate when you start your server. If you wrongly have some of the classes to be downloaded available in the registry’s classpath, they will not be able to be downloaded to a REMOTE client via the server’s java.rmi.server.codebase property. The registry in this situation does not even bother (!) to send in the ‘looked up’ reference the correct location of the stubs to the client. If your client gets a ClassNotFoundExeption, this may be your problem.

In the second command window, from a directory where the Java Virtual Machine can find your server, start the server with a single line command similar to the following:

>D:\test>java -Djava.rmi.server.codebase=file:/d:/test/DownCounterImpl

DownCounter instance now listening.
DownCounter instance registered.

Note the above was a single line command that unfortunately must be ‘wrapped’ to fit on this page width. Leave no blanks except after the word ‘java’ and another before the server name. The first and last slashes are REQUIRED and MUST be forward slashes! Java seems to not care if the middle slash is forward or backward! If your stubs are on a different machine which is an applet web server, you would want a codebase something like http://machineName/~tront/classes/

Note also the two lines of output from the server above. The server main program may actually end, but the operating system process that it is running in does not! Due to the Java’s DISTRIBUTED automatic garbage collection, the server process will not end until all references to the remote object instance are gone. Currently, the registry program still holds a remote reference to the remote instance in the server process’s memory heap, so this relieves us from having to insert any ‘keep alive’ code near the end of the server main function.

Now in a third Command window, run the client. Here is the client being started from an appropriate directory, and the results.

D:\test>java DownCounterClient localhost

DownCounter found in registry.
Beginning remote invocations.
Returned value = 9
Returned value = 8
Returned value = 7
Returned value = 6
Returned value = 5
>
At this point, the client actually ends and you get a new prompt. You can run the client again, and verify the servant is still in existence:

D:\test>java DownCounterClient localhost
DownCounter found in registry. Beginning remote invocations.
Returned value = 4
Returned value = 3
Returned value = 2
Returned value = 1
Returned value = 0
>

Notice that the count continued down from where it last left off!

Finally, you can shut down the registry and server with Cntl-C in their respective windows.

At this point it would be wise to try this distributed application with the client running on a different machine. It would also be wise to try creating a common remote interface file shared between you and your neighbor, then see if his/her client will work with your server, and visa versa!

1.9 Serialization

When you make a call from a client on one machine to a server on another, you may possibly pass as a parameter a reference to second remote object on yet a third machine. This is OK, but is not serialization.

```
Client

meth1(remClass paramObject);

Class ServantImpl ... {
    Void meth1(remClass theParam){
        // now call the third object.
        theParam.otherMethod();
    }
}
```

The servant method will use this remote reference as it sees fit. If the servant method decides to in turn call a remote method on the remote object on the third machine, then the server will need the stubs for the paramObject class located on the third machine. If the stubs are already available on the second machine fine. If not,
they will automatically be downloaded from the third machine because the remote reference originally passed contains the full location info for the actual parameter object! This is not serialization.

Alternately, Java RMI has the capability of passing a non-remote object (one that does NOT implement the interface java.rmi.Remote) as a parameter to any servant’s remote methods. What does this mean?

Well, it means passing an object by value. Primitive types like int and float are always passed by value. For object instance though, instead of passing a reference to the object as a parameter, the actual member attribute data of the passed object is sent. Later, if necessary, the Java byte codes for the actual member functions (not stubs) of the parameter class are downloaded I believe. In essence, that instance and its implementation code are ‘serialized’ into a stream of bits and shipped off to the destination of the call. This is called ‘moving behavior’ or ‘mobile objects’, and is a very exciting concept. It allows a program to rove through out the Internet! Who knows what wonderful (and possibly bad) things this might allow us to do in future.

The ONLY thing that determines whether a parameter object is passed by reference, or by serialized value, is whether the parameter class is declared to support the interface java.rmi.Remote or not. This is not always very obvious when reading calling or called code, so good programmer commenting is wise in this area.

Note that most objects are serializable, but not all. To make sure the programmer has thought about whether he/she should let the compiler serialize an instance of a particular class, the compiler will NOT serialize an instance unless its class is declared to implement the interface java.io.Serializable.

To summarize:
• If a parameter does implement the java.rmi.Remote interface, it is sent by reference, and the location of the object is passed. The stubs can be gotten by the server from that third location.
• If a parameter does NOT implement the java.rmi.Remote interface, it is sent by serialized value. The actual member attribute data is sent, and the original location of the parameter’s class (which may not be where the serialized bits are coming from for this call) is sent so that the full member function implementations (NOT stub implementations) can be downloaded if necessary.
1.10 Exercises

Your instructor will likely make the code from the above examples available. Good exercises are:

1. Try running the code as is.
2. Try running the client on one machine and the server/servant on another.
3. Add code to the servant so that it throws an exception when the count falls to zero. Of course the client should have a custom try/catch block added to handle this (remote!) event gracefully.
4. Add code to the servant so that it calls a remote method exported from the client (!) to inform the client that the count has reached zero. Instead of having the client advertise this method in its registry, it would be better for the client to pass a remote reference to itself (i.e. a reference to a callback instance) in an initial call to the server.
5. Calls to a servant from different client virtual machines will execute in different server threads. To beautifully demonstrate this, change the decrementing code so that it copies theCount variable into a local variable, then does:
   ```java
   try {
       Thread.sleep(5000);
   } catch (Exception e) {
   }
   
   then decrements the local variable and uses that to update and return theCount. Now try two clients calling the servant AT THE SAME TIME, and watch for race/critical section anomalies! Next, demonstrate that you can get rid of these anomalies simply by using the elegant Java ‘synchronized’ keyword on the servant (not interface) method signature. This makes each servant instance into its own protected section.
6. Add code to move the call to rebind from the server main into the servant class constructor. The server can then create several/many servant instances without having to handle the rebind of each. Since each instance should have a different advertised name, perhaps the servant class will append a digit to the advertised name of each successive class, or maybe you will add a non-default constructor that accepts the advertised name of the instance to be constructed in that constructor’s parameter.
1.11 Appendix - Details on Using TextPad

TextPad from www.textpad.com is not much more than a simple text editor that sits on top of the Sun Java Development Kit. If you have a JDK already installed on your computer, it allows you to add two menu items to the tools menu (not toolbar): ‘Compile Java’ and ‘Run Java’. (Version 4 might additionally have one that runs a Java Applet, though the Run Java item in Version 3.2.5 will run either an applet or an application).

You can also add new commands to the tool menu. I typically add another which invokes the RMI compiler: rmic.

If TextPad did not detect your Java JDK when TextPad was installed, or you installed the Java JDK after TextPad, you will have to configure TextPad to work with Java. Select Configure|Preferences|Tools|Add|JDK Commands, then click on Apply. If you are not able to do this, likely you have not set the DOS/Windows path variable to where the javac compiler is available (Help|Help Contents|How To|Use With Other Applications|Java Development Kit). Also, if you can’t jump directly from error messages to the source line in error, it is likely that the Configure|Preferences|Tools|Compile Java regular expression did not get set right. See Help|Help Contents|How To|Use With Other Applications|Compilers for the correct regular expression and other details.

To add your own custom menu item for the RMI compiler ‘rmic’, use Configure|Preferences|Tools|Add|Program and set the following:
1. Find the rmic command in the jdk1.2.2/bin directory and select that as the command. Click on Apply.
2. Expand the Tools tree and check the Command is set right.
3. Set the Parameters to $BaseName.
4. Check that the Initial Folder is $FileDir
5. Make sure the Save and the Capture check boxes are checked.

Note: TextPad is not really aware of the relationship of packages to subdirectories, so avoid using named packages with TextPad until you really understand these. This is particularly true with rmic. Rmic requires on the command line the package qualified name, like:

>rmic myPackage.myClassName

even if this class is in the simply-named file myClassName.class. Also, rmic from jdk1.2.2 (but not 1.3) puts the stubs and skeletons in the directory it is run from, not in the directory from which the servant class file was obtained from! If the servant class imports other classes from other packages (as the Sun Java Tutorial RMI trail does), you will probably need to add a -classpath as shown below:

>rmic -classpath myprojdir;d:\otherdir myPackage.myClassName

Also, you can add “-d .” to the rmic command, and in jdk1.2.2 that surprisingly seems to put the generated stubs and skeletons
back in the directory from which the servant class was found (rather than in the current directory which you would think that the dot means)! Note: Java 1.3 will reverse some of this odd behavior to make it more normal.

Since the Java support in TextPad is just used to create DOS command lines, if you are working on a multi-package project you can add as needed some of these options to your TextPad customized java tool Parameter or Initial Folder tool fields.