REQUIREMENTS

Use Java socket to implement a simplified RMI system.

Assumptions:

- No consideration of different protocol types.
- JVM on all nodes.
- Each port can have at most one remote object listening.
- Each remote object implements exactly one remote interface.
- The parameters of the remote methods are either of primitive types or local objects.

Requirements:

- >>>>> java StartRegistry
  can be used to start the registry process to listen at a specific port 11111.
- The registry process maintains a table of registered (name, reference) pairs.
- Class *MyUnicast.java* is responsible for assigning anonymous port numbers for exporting remote objects to listen to incoming calls.
- Class *MyUnicast.java* is responsible for creating new threads to listen to incoming calls.
- *exportObject()* receives the (local) object reference as a parameter. It could also receive as a parameter, the remote interface that the object implements. The second parameter is for simplification.
- The return of *exportObject* is a remote object reference.
- Class *MyRegistry.java* should implement *rebind* and *lookup* methods.
- *StartRegistry.java*, *MyUnicast.java*, and *MyRegistry.java* should not contain any information specific to its upper layer applications e.g. Pi computation, prime number computation, etc. Java reflection could be used.
- A *remote object reference* should contain the information of a) the (ip, port number) of the place where the *remote object* is listening; b) the name of the *remote interface*.
- A *remote object reference* should implement its *remote interface*.
- The *server stub* should listen at the assigned port and for each incoming method invocation request, perform the communication, marshalling, unmarshalling, and method invocation.

- Upon user's method invocation request, the *client stub* performs the communication, marshalling, and unmarshalling.

- Server stub and client stub are not dynamically generated. They can be different programs. They are for specific *remote interfaces*.

RMI applications (testing):

- pi computation implemented at server side
- prime number computation implemented at server side
- computeEngine with pi computation or prime number computation at client side

TAKING STEPS

1. Use Java socket to implement a client/server application to simulate the RMI mechanism: a) client passes method name and parameters to the server; b) server calls the method implementation, and returns the result of Pi computation.

   Pi_Client is deployed to both sides.

   The class of Pi implementation is on the server side. There is no transfer of class definitions.

   Get your files properly organized:

   Pi (interface)
   Pi_Client
   Pi_Server
   Server
   Client

2. Implement *MyUnicast*, *MyRegistry*, and *MyStartRegistry*.

   For *MyUnicast*, use one additional thread for each exported remote object.

   Implement a registry to listen at a specific port 11111, and communicate with client and server via *rebind* and *lookup*.

3. Implement the above for *prime computation* example. Compare it with the one for Pi computation. Use Java reflection to build library code.

4. Implement the above for the compute engine example.

5. (optional) Compare the following code and implement the automated generation of these files.

   Pi_Client/Prime_Client
   Pi_Server/Prime_Server

   Implement the passing of remote object reference.

SIMPLIFICATION

A *remote interface* must be declared *public*. It must extend interface *java.rmi.Remote* (this interface does not contain any methods).

You may not need this. You can assume that there is only remote method in each remote interface. You do not need to do any error checking.

Each *remote method* must declare *java.rmi.RemoteException* (or a superclass of it) in its throws clause, in addition to any application-specific exceptions.

You do not have to generate or pass exceptions.

Remote object implementation class may have methods that are not in its *Remote interface*. These can only be invoked locally.

You can assume that local methods do not appear in the implementation class of a remote interface.

If a remote object is being used remotely, its type must be declared to be the type of the *Remote interface*, not the type of the *Remote class*.

Same.

If class definitions for *Serializable classes* need to be downloaded from another machine, then the security policy of your program must be modified. Java provides a security manager class called RMISecurityManager for this purpose.

You do not have to work on security issues.

The *server program* should create one or more instances of a remote object, *export* the remote object**,** and *register* at least one of the remote objects with the RMI *remote object registry*.

Same.

With the use of *UnicastRemoteObject*, we can create a remote object that uses RMI's default socket-based transport for communication and runs all the time at an anonymous port on the server computer.

Use socket-based mechanism.

Create a separate thread at the anonymous port. You need to generate and assign an anonymous port number for each invocation of *exportObject(...)*.

You can assume that there will be only one invocation at a time for each *remote object*, so you do not have to consider threading issues.

Web server could be used for communication.

Not needed. Use socket-based mechanism.

A *remote reference* is actually a reference to a stub, which is a client-side proxy for the remote object.

Same.

Arguments to or return values from *remote methods* can be of almost any type, including: (i) primitive types; (ii) *remote object* (implements *Remote*); (iii) local object (implements *Serializable*)

You need to implement the parameter passing at least for primitive types and local objects. You can predefine a list of primitive types that you handle.

You do not have to check for type errors.

Certain values e.g. file descriptor, thread, cannot be passed.

Same.

It is not necessary for both *client* and *server* to have access to the definition of the *Remote class*. The *server* requires the definition of both the *Remote class* and the *Remote interface*, but *client* only uses *Remote interface*.

Same.

If the *client* and *server* programs are on different machines, then class definitions of *Serializable* classes may have to be downloaded from one machine to the other. Such a download could violate system security.

Download without considering security issues.

If a *serializable object* is passed as a parameter (or return value), the value of the object will be copied from one address space to the other.

If a *remote object* is passed as a parameter (or return value), the object stub will be copied from one address space to the other.

Same.

A class that is both *Remote* and *Serializable* is considered poor design.

You do not have to consider this. Assume that all classes will be either *Remote* or *Serializable*, but not both.

All of the *Remote interfaces* and *classes* should be compiled using *javac*. Once this has been completed, the stubs and skeletons for the *Remote interfaces* should be compiled by using the *rmic* stub compiler.

Same. Your server stub and client stub could be different programs.

The class definition of a stub must be available to the client, either locally (put on client part before the communication) or via download (put on server part but dynamically downloaded to client part).

Same. Implementation of the dynamic downloading is required.

We can dynamically download class definitions because both sides run Java programs, and we have JVM.

Same.

The generated stub class implements exactly the same set of remote interfaces as the remote object itself.

Simplification: each stub class for only one remote interface.

The object registry allows one to obtain a *remote object* using only the name of the *remote object*. The name of a *remote object* includes the following information:

- IP of the machine that is running the object registry.
- The port to which the object registry is listening. If the object registry is listening to the default port 1099, then this does not have to be included in the name.
- The local name of the *remote object* within the object registry.

The registry is run on the same machine of the server programs. The default port number is 11111. Local name of the remote object could be simply the remote interface name.

The result of *Naming.lookup* must be cast to the type of the *Remote interface*.

Same.

The *RMI object registry* only accepts requests to bind/unbind objects running on the same machine, so it is never necessary to specify the name of the machine when one is registering an object.

Same.

Users can start a registry on a port different from 1099, e.g.

start rmiregistry 2001 (in Window NT)

In this setting, you should also use:

Naming.rebind("//myhost:2001/HelloServer", obj)

Use default 11111. You can assume that only 11111 is used.

We should start the *rmiregistry* first, then the server program, followed by the client program. E.g.

>>>>> rmiregistry (Solaris)
>>>>> start rmiregistry (Windows 95 & NT)

>>>>> java HelloServer
>>>>> java HelloClient

Same.

The *ComputeEngine* example used *type variable* for the return value of *Task*.

Type variable is not required. You can assume that *Task* always returns integer.

The interface *Compute* and interface *Task* are downloaded from the server to the *registry* at runtime.

Same.