A Typical RMI Application

- Client and Server run on different machines
- Remote Object(s) registered in rmiregistry by Server
- Remote Object(s) look’d up by Client
- When necessary, code transferred from web server to point of use
  - Both Client and Server can make code network accessible
- Operations on Remote Objects carried out by RMI
Case Study

• This example taken directly from the Java RMI tutorial
  – http://docs.oracle.com/javase/tutorial/rmi/index.html

• Editorial note:
  – Please do yourself a favor and work through the tutorial yourself
  – If you get the tutorial to work, you’ll have no problems with project 4 or with the RMI portion of the final exam
  – For a webserver, I use apache running on my laptop.
  – You can also use
    • http://terpconnect.umd.edu
  – You can also use a simple RMI webserver:
    • http://www.oracle.com/webfolder/technetwork/java/core/basic/rmi/class-server.zip
Compute Server Application

- **Goal**
  - Execute object methods on a remote machine
  - Often because local resources aren’t sufficient
- **Real-life example: Amazon EC2**
  - Large computing infrastructure -- somewhere in the cloud
  - Users push many different kinds of work to these rented machines
    - Examples: Justin.tv, Zillow.com, NY Times (PDF conversion)
package compute;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Compute extends Remote {
    <T> T executeTask(Task<T> t) throws RemoteException;
}

• Any class that implements Compute is a Remote object
  – Its Remote methods can be called from any JVM
  – Its implementation does not leave the JVM in which it was created
• executeTask() is a Remote method
  – It must throw RemoteException
package compute;
public interface Task<T> {
    T execute();
}

• Task doesn’t implement Remote
  – Why not?
• execute() method returns an instance of type T
  – Method not required to throw RemoteException
Implementing Compute Engine

• Our implementation of the Compute interface will be called *ComputeEngine*

• In general, a Remote interface impl should:
  1. Declare the Remote interfaces being implemented
  2. Define the constructor for the Remote object
  3. Implement each Remote method in the Remote interfaces
Further Requirements for Servers

• The server needs to create and to set up the Remote objects
  – The setup procedure often done in main() method of the Remote object
    • but can be done anywhere

• The setup procedure should
  1. Create and install a security manager
  2. Create one or more instances of a Remote object
  3. Register at least one of the Remote objects with the RMI registry
• The ComputeEngine class is declared as
  public class ComputeEngine implements Compute {

Define the Constructor

• **ComputeEngine** has a single, 0-arg constructor
  
  ```java
  public ComputeEngine() {
      super();  // optional
  }
  ```
Implement Each Remote Method

• Compute has a single Remote method, `executeTask()`:
  
  ```java
  public <T> T executeTask(Task<T> t) {
    return t.execute();
  }
  ```

• Client provides `ComputeEngine` with a Task object
  – Which implements the Task's `execute()` method

• `ComputeEngine` executes the Task and returns the result
Implement the Setup Procedure

• Create and install a security manager
• Create one or more instances of Remote objects
• Register at least one of the Remote objects with the RMI registry
Create and Install a Security Manager

- Security Manager determines whether downloaded code has access to the local file system or can perform any other privileged operations
- Without a security manager, RMI will not download classes (other than from the local class path) for objects received as parameters, return values, or exceptions in Remote method calls
  ```java
  if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
  }
  ```
- Policy files can grant specific permissions
  - if you want to modify SecurityManager’s default perms
The main method creates an instance of ComputeEngine

- `Compute engine = new ComputeEngine();`

Note engine’s type is Compute, not ComputeEngine

- The interface is available to clients, not the implementation
- At runtime, you’ll pass the stub, not the actual implementation

The main method exports the Remote object (activates it)

- `Compute stub = (Compute) UnicastRemoteObject.exportObject(engine, 0);`
Make the Remote Object Accessible

• To invoke a Remote object, caller needs a reference to it
• Can get it from the program (return value, data field, etc.)
• Can look it up in an RMI registry
  – The RMI registry is a simple Remote object naming service
Make the Remote Object Accessible

• Start the registry
  – From the command line as a separate process, or
  – From within your Server program

• If registry is started within server, it will be shut down when program shuts down

• If not, you’ll need to remember to shut it down manually
Add Remote Object to Registry

- The `java.rmi.Naming` interface is API for binding, or registering, and looking up Remote objects in the registry
- The `ComputeEngine` class creates a name for the Remote object
  ```java
  String name = "Compute";
  ```
- Then finds the registry
  ```java
  Registry registry = LocateRegistry.getRegistry();
  ```
Add Remote Object to Registry

- Then adds Remote object to the registry
  ```java
  registry.rebind(name, stub);
  ```
- Application can bind, unbind, or rebind Remote object references only with a registry running on the same host
- Once the Remote object is registered, the setup procedure exits
Creating a Client Program

• Two separate classes make up the client in our example
  – ComputePi
  – Pi
• ComputePi gets a reference to a Compute object, creates a Task object, and then requests that the task be executed
• Pi implements the Task interface, calculating Pi to the required degree of precision
ComputePi

- Begins by installing a security manager
- Constructs the name used to look up Compute Remote Object
- Uses Registry.lookup() to look up the Remote Object by name in the remote host's registry
- Creates a new Pi Object
ComputePi

- Invokes `executeTask()` on the Compute Remote Object
- `executeTask()` returns an Object of type `java.math.BigDecimal`
- Program prints out the result
• Calculates Pi
• Implements Serializable. Why?
  – It’s computationally expensive which is why you want to run it on a (presumably) fast compute server
• Think of the application as logically having 4 directory trees

• Server
  – Application directory – (server code written and compiled here)
  – Web accessible location – (client downloads server code from here)

• Client
  – Application directory (client code written and compiled here)
  – Web accessible location - – (server downloads client code from here)

• Editorial note:
  – You have to put all the code in the right places each time you make changes
    • So use a makefile!
  – Ultimately you should put client and server code in separate directory trees / separate machines
    • Otherwise you may not know if things are really working
• Compile interface classes, build a jar file
  – Move jar file to developer-accessible locations
  – Everyone shares these files – don’t change them

• Build Server classes
  – (add classpath info to the following command lines)
  – cd ServerDevDir
  – javac engine/ComputeEngine.java

• For this example, no server classes will be downloaded
Compiling

• Build the Client classes
  – cd ClientDevDir
  – javac client/ComputePi.java client/Pi.java
  – mkdir ClientWebDir/client
  – cp client/Pi.class ClientWebDir/client/

• Client class is now web-accessible
Running Application

• Copy policy file to some directory
  – On Unix I put the file in ./java.policy

• Start the RMI registry (done by program in our example)
  – rmiregistry portNum &

• Start the server
  
  ```
  java -classpath ServerDevDir/  
  -Djava.rmi.server.useCodebaseOnly=false  
  -Djava.security.policy=java.policy  
  -Djava.rmi.server.logCalls=true  
  -Dsun.rmi.server.logLevel=VERBOSE  
  -Dsun.rmi.loader.logLevel=VERBOSE  
  engine.ComputeEngine regPort
  ```
Running Application

• Start the client (on another machine)
  
  ```
  java 
  -classpath ClientDevDir/ 
  -Djava.rmi.server.codebase=http://ClientWebServer/ClientWebDir/ 
  -Djava.security.policy==java.policy 
  -Djava.rmi.server.logCalls=true 
  -Dsun.rmi.server.logLevel=VERBOSE 
  -Dsun.rmi.loader.logLevel=VERBOSE 
  client.ComputePi serverName 20
  ```

• Should produce
  
  – 3.14159265358979323846

• Note: don’t forget trailing “/” on codebase (no “/” for jar files)