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://java.sun.com/docs/books/tutorial/rmi
- 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 5 or
    with the RMI portion of the final exam
  - For a webserver, I use apache httpd running on my laptop.
  - You can also use
    - http://terpconnect.umd.edu
  - You can also use a simple RMI webserver:
    - http://java.sun.com/javase/technologies/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 clouds
  - Users push many different kinds of work to these rented machines
    - Examples: Justin.tv, Zillow.com, NY Times (PDF conversion), Twitter

Compute Interface

```java
class 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
Task Interface

```java
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 compute interface is 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 of the remote interfaces
Further Requirements for Servers

• The server will create and publish one or more remote objects
  – The setup procedure is often done in main() method of the remote object, but can be done anywhere

• The setup procedure should
  – Create and install a security manager
  – Create one or more instances of remote objects
  – Register at least one of the remote objects with an RMI registry

Declare the Remote Interfaces

• 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
Create & Export the Remote Object

- The main method creates an instance of ComputeEngine
  - `Compute engine = new ComputeEngine();`
- Note (for clarity) the 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
- 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
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 or creates the registry
  ```java
  Registry registry = LocateRegistry.getRegistry();  // Registry is external
  Registry registry = LocateRegistry.createRegistry(1099);
  ```
- 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 some 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
- Invokes executeTask() on the Compute remote object
- executeTask() returns an object of type java.math.BigDecimal
- Program prints out the result

**Pi**

- Calculates Pi
- Implements Serializable. Why?
  - It’s computationally expensive which is why you want to run it on a fast compute server
Compiling

• Application has 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
    • Automation is helpful. Use some kind of makefile
  – Ultimately you need to put client and server code in separate directory trees / separate machines
    • Otherwise you may not know if things are really working

Compiling

• 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/
• Note that the download directory matches the package structure – client/Pi.class
• 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, otherwise use
  – rmiregistry portNum &
• Start the server
  
  java -classpath ServerDevDir/ \ 
  -Djava.security.policy=java.policy \ 
  -Djava.rmi.server.logCalls=true \ 
  -Dsun.rmi.serverLogLevel=VERBOSE\ 
  -Dsun.rmi.loader.logLevel=VERBOSE\ 
  engine.ComputeEngine

• Note: don’t forget trailing “/” on codebase (no “/” for jar files)
• Some helpful troubleshooting tips at:
  – http://java.sun.com/j2se/1.4.2/docs/guide/rmi/codebase.html
Running Application

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

• Should produce
  
  3.14159265358979323846