RMI - Eclipse
CSCI 201L

Jeffrey Miller, Ph.D.
jeffrey.miller@usc.edu

HTTP://WWW-SCF.USC.EDU/~CSCI201
Outline

- Remote Method Invocation
- Program
RMI Overview

- RMI is a Java-implementation of RPC referred to as a distributed object application
- An RMI server typically creates some remote objects, makes references to those objects accessible, and waits for clients to invoke methods on those objects
- An RMI client obtains a remote reference to one or more remote objects on a server and invokes methods on them
  - RMI clients can locate remote objects through an RMI registry, assuming the RMI server has registered its remote objects with it
- The details of remote communication between server and client are handled by RMI
  - Remote communication looks like regular Java method invocations to the programmer
- The client can pass a class to a remote server and have it execute methods on that class
RMI Dynamic Code Loading

- RMI has the ability to download the definition of an object’s class if the class is not defined in the client’s JVM
  - All of the types and behaviors of an object can be transmitted to a remote JVM
  - New types and behaviors can be introduced into a remote JVM, thus dynamically extending the behavior of an application
RMI Application Example

Here are the steps for setting up an RMI application
  › This example is adapted from http://docs.oracle.com/javase/tutorial/rmi/overview.html

1. Write the code
   1.1 Write the remote interface
   1.2 Write the server code
   1.3 Write the client code
2. Compile the code
3. Make the classes network accessible
4. Start the RMI server and the application
RMI Application Example – Step 1.1

- Step 1.1 – Write the remote interface
  > The example we will see here allows a client to submit a task to a server program, the server program to run that task, and the results of that task returned to the client

  > **Compute** is the remote interface that allows tasks to be submitted to the engine
    - Since **Compute** inherits from `java.rmi.Remote`, its method `executeTask(Task<T>)` can be invoked from another JVM

  > **Task** is the client interface that defines how the compute engine executes a submitted task
    - The **Task** interface is the parameter to the `executeTask` method in the **Compute** interface

  > Objects are passed from client to server serialized, so the class implementing the **Task** interface and the parameterized return type `T` must both be `Serializable`
RMI Application Example – Step 1.1

Compute.java
1   package compute;
2
3   import java.rmi.Remote;
4   import java.rmi.RemoteException;
5
6   public interface Compute extends Remote {
7     <T> T executeTask(Task<T> t) throws RemoteException;
8   }

Task.java
1   package compute;
2
3   public interface Task<T> {
4     T execute();
5   }
RMI Application Example – Step 1.1

- Create `Task.java` and `Compute.java` in the `compute` package
Since RMI can assume the `Task` objects are written in Java, implementations of the `Task` object that were previously unknown to the server are downloaded by RMI into the server’s JVM.

This means that clients are able to define new types of tasks to be run on the server without that code needing to be explicitly installed on the server.

The server code in the `ComputeEngine` class implements the `Compute` interface and enables different tasks to be submitted to it by calls to the `executeTask` method.

This method just executes the task’s `execute` method and returns the results to the remote client.
RMI Application Example – Step 1.2

- Step 1.2 – Write the server code
  - A class that implements a remote interface needs to provide an implementation for each remote method in the interface
  - The server program needs to create the remote objects and export them to the RMI runtime, making them available to receive incoming remote invocations
  - A security manager must be created and installed so the RMI runtime knows what can be executed on the server
  - Remote objects are passed by reference from a client
  - Other parameters that are not remote objects are passed by value
package engine;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;

public class ComputeEngine implements Compute {
    public ComputeEngine() {
        super();
    }
    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Compute engine = new ComputeEngine();
            Compute stub = (Compute) UnicastRemoteObject.exportObject(engine, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind(name, stub);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
            System.err.println("ComputeEngine exception: ");
            e.printStackTrace();
        }
    }
}
RMI Application Example – Step 1.2

- Create `ComputeEngine.java` in the `engine` package
Step 1.3 – Write the client code

- The client for this program needs to define the task that it wants the server to perform ($\pi$)
  - This means the client needs to create a class that implements the Task<T> interface
- The client has another program (ComputePi) that will obtain a reference to the newly-created Task<T> object and request it to be executed on the server
  - This means that it needs to contact the RMI registry and submit the Task to be executed by calling the executeTask(Task<T>) method on a Compute object
package client;

import compute.Task;
import java.io.Serializable;
import java.math.BigDecimal;

public class Pi implements Task<BigDecimal>, Serializable {

    private static final long serialVersionUID = 227L;

    /** constants used in pi computation */
    private static final BigDecimal FOUR = BigDecimal.valueOf(4);

    /** rounding mode to use during pi computation */
    private static final int roundingMode = BigDecimal.ROUND_HALF_EVEN;

    /** digits of precision after the decimal point */
    private final int digits;

    /**
     * Construct a task to calculate pi to the specified
     * precision.
     */
    public Pi(int digits) {
        this.digits = digits;
    }

    /**
     * Calculate pi.
     */
    public BigDecimal execute() {
        return computePi(digits);
    }

    /**
     * Compute the value of pi to the specified number of
     * digits after the decimal point using Machin’s formula.
     * pi/4 = 4*arctan(1/5) - arctan(1/239)
     */
    public static BigDecimal computePi(int digits) {
        int scale = digits + 5;
        BigDecimal arctan1_5 = arctan(5, scale);
        BigDecimal arctan1_239 = arctan(239, scale);
        BigDecimal pi = arctan1_5.multiply(FOUR).subtract(
            arctan1_239).multiply(FOUR);
        return pi.setScale(digits, BigDecimal.ROUND_HALF_UP);
    }

    /**
     * Compute the value, in radians, of the arctangent of
     * the inverse of the supplied integer to the specified
     * number of digits after the decimal point.  The value
     * is computed using the power series expansion
     * arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 + ...
     */
    public static BigDecimal arctan(int inverseX, int scale) {
        BigDecimal result, numer, term;
        BigDecimal invX = BigDecimal.valueOf(inverseX);
        BigDecimal invX2 = BigDecimal.valueOf(inverseX * inverseX);
        numer = BigDecimal.ONE.divide(invX, scale, roundingMode);
        result = numer;
        int i = 1;
        do {
            numer = numer.divide(invX2, scale, roundingMode);
            int denom = 2 * i + 1;
            term = numer.divide(BigDecimal.valueOf(denom),
                scale, roundingMode);
            if ((i % 2) != 0) {
                result = result.subtract(term);
            } else {
                result = result.add(term);
            }
            i++;
        } while (term.compareTo(BigDecimal.ZERO) != 0);
        return result;
    }

    /**
     * Construct a task to calculate pi to the specified
     * precision.
     */
    public Pi(int digits) {
        this.digits = digits;
    }

    /**
     * Calculate pi.
     */
    public BigDecimal execute() {
        return computePi(digits);
    }
}
package client;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;
import compute.Compute;

public class ComputePi {
  public static void main(String args[]) {
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new SecurityManager());
    }
    try {
      String name = "Compute";
      Registry registry = LocateRegistry.getRegistry(args[0]);
      Compute comp = (Compute) registry.lookup(name);
      Pi task = new Pi(Integer.parseInt(args[1]));
      BigDecimal pi = comp.executeTask(task); // makes remote procedure call
      System.out.println(pi);
    } catch (Exception e) {
      System.err.println("ComputePi exception:");
      e.printStackTrace();
    }
  }
}
RMI Application Example – Step 1.3

- Create **Pi.java** and **ComputePi.java** in the **client** package
RMI Application Example – Step 2

- Step 2 – Compile the code
  - Eclipse will automatically compile the code, so nothing else is required on this step
RMI Application Example – Step 3

- Step 3 – Make the classes network accessible
  - Since all of the files are in the same project in Eclipse, they will all be accessible to the other classes
  - Nothing else is required on this step
RMI Application Example – Step 4

- You need to create a *server.policy* file to specify the security a client will have running a program on the server.
- You also need to create a *client.policy* file to specify the security a server will have in an object it returns to the client.

**server.policy**

```java
grant {
    permission java.security.AllPermission;
};
```

**client.policy**

```java
grant {
    permission java.security.AllPermission;
};
```
RMI Application Example – Step 4

- Create new files server.policy and client.policy
  - Note: In the “New” menu, there is an item called “File”
RMI Application Example – Step 4

- Step 4 – Start the RMI server and the application
  - The RMI registry is executed by running a program that came with the JDK called rmiregistry
  - To run this within Eclipse, go to “Run->External Tools->External Tools Configuration…”
RMI Application Example – Step 4

- Step 4 – Start the RMI server and the application
  - Click the “New Launch Configuration” button and name it “rmiregistry”
  - For the “Location,” click “Browse File System” to find the rmiregistry program in the bin directory of your JRE directory
  - For the “Working Directory,” click “Browse Workspace” to find the bin directory of your project
  - Click “Apply” then “Run” – the RMI registry should be running but there is no output
RMI Application Example – Step 4

- Step 4 – Start the RMI server and the application
  - We need to add a Java environment variable before running the server
  - Click the arrow next to the green Run button and click “Run Configurations”
  - Type “ComputeEngine” as the “Name,” your project as the “Project”, and “engine.ComputeEngine” as the “Main class”
RMI Application Example – Step 4

- Step 4 – Start the RMI server and the application
  - On the “Arguments” tab, type the following as the “VM arguments”
    -Djava.security.policy=${workspace_loc}/CSCI201Spring2015-RMI/server.policy
  - Note: Replace “CSCI201Spring2015-RMI” with your project’s name
  - Click “Apply” then “Run” and you should see the ComputeEngine running
RMI Application Example – Step 4

- **Step 4 – Start the RMI server and the application**
  - We need to add a Java environment variable and command line parameters before running the client
  - Click the arrow next to the green Run button and click “Run Configurations”
  - Type “ComputePi” as the “Name,” your project as the “Project”, and “client.ComputePi” as the “Main class”
RMI Application Example – Step 4

- Step 4 – Start the RMI server and the application
  - On the “Arguments” tab, type the following as the “VM arguments”
    -Djava.security.policy=${workspace_loc}/CSCI201Spring2015-RMI/client.policy
  - Note: Replace “CSCI201Spring2015-RMI” with your project’s name
  - Type “localhost 55” as the “Program arguments,” which are the command line arguments read through the args variable in the program
  - Click “Apply” then “Run” and you should see ComputePi running
Outline

- Remote Procedure Calls
- Remote Method Invocation
- Program
Program

- Modify the RMI application we wrote today to have the server determine the value of Pi instead of the client sending the code to be executed. The server will perform the calculation and return the value of Pi based on the number of digits in a variable to the client.
  
  In other words, the client should not know how to compute Pi to the number of digits specified – the server should do all of that.