Distributed Systems

02r. Part 1: Java RMI Programming Tutorial

Paul Krzyzanowski
Rutgers University
Fall 2016

Java RMI

- RMI = Remote Method Invocation
- Allows a method to be invoked that resides on a different JVM (Java Virtual Machine):
  - Either a remote machine
  - Or same machine, different processes
    - Each process runs on a different Java Virtual Machines (JVM)
    - Different address space per process/JVM

RMI provides object-oriented RPC (Remote Procedure Calls)

Participating processes

- **Client**
  - Process that is invoking a method on a remote object
- **Server**
  - Process that owns the remote object
  - To the server, this is a local object
- **Object Registry** (rmiregistry)
  - Name server that associates objects with names
  - A server registers an object with rmiregistry
  - URL namespace

```
        rmi://hostname:port/pathname
e.g.:   rmi://crapper.pk.org:12345/MyServer
```

Port number

Classes & Interfaces needed for Java RMI

- **Remote**: for accessing remote methods
  - Used for remote objects
- **Serializable**: for passing parameters to remote methods
  - Used for parameters
- Also needed:
  - RemoteException: network or RMI errors can occur
  - UnicastRemoteObject: used to export a remote object reference or obtain a stub for a remote object
  - Naming: methods to interact with the registry

Remote class

- **Remote** class (remote object)
  - Instances can be used remotely
  - Works like any other object locally
  - In other address spaces, object is referenced with an *object handle*
    - The handle identifies the location of the object
  - If a remote object is passed as a parameter, its handle is passed

Serializable interface

- **java.io.Serializable** interface (serializable object)
  - Allows an object to be represented as a sequence of bytes (marshaled)
  - Allows instances of objects to be copied between address spaces
    - Can be passed as a parameter or be a return value to a remote object
    - Value of object is copied (pass by value)
  - Any objects that may be passed as parameters should be defined to implement the **java.io.Serializable** interface
    - Good news: you rarely need to implement anything
    - All core Java types already implement the interface
    - For your classes, the interface will serialize each variable iteratively

Remote classes

- Classes that will be accessed remotely have two parts:
  1. interface definition
  2. class definition
- **Remote interface**
  - This will be the basis for the creation of stub functions
  - Must be public
  - Must extend `java.rmi.Remote`
  - Every method in the interface must declare that it throws `java.rmi.RemoteException`
- **Remote class**
  - implements Remote interface
  - extends `java.rmi.server.UnicastRemoteObject`

Super-simple example program

- Client invokes a remote method with strings as parameter
- Server returns a string containing the reversed input string and a message

Define the remote interface

SampleInterface.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface SampleInterface extends Remote {
    public String invert(String msg) throws RemoteException;
}
```

- Interface is public
- Extends the Remote interface
- Defines methods that will be accessed remotely
  - We have just one method here: *invert*
- Each method must throw a RemoteException
  - In case things go wrong in the remote method invocation

Define the remote class (Sample.java)

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.server.*;

public class Sample
  extends UnicastRemoteObject
  implements SampleInterface {

    public Sample() throws RemoteException {  }
    public String invert(String m) throws RemoteException {
        // return input message with characters reversed
        return new StringBuffer(m).reverse().toString();
    }
}
```

- Defines the implementation of the remote methods
- It implements the interface we defined
- It extends the java.rmi.server.UnicastRemoteObject class
  - Defines a unicast remote object whose references are valid only while the server process is alive.

Next…

- We now have:
  - The remote interface definition: SampleInterface.java
  - The server-side (remote) class: Sample.java
- Next, we'll write the server: SampleServer.java
- Two parts:
  1. Create an instance of the remote class
  2. Register it with the name server (**rmiregistry**)

Server code (SampleServer.java)

- Create the object

```
new Sample()
```

- Register it with the name server (rmiregisty)

```
Naming.rebind("Sample", new Sample())
```

- *rmiregistry* runs on the server
  - The default port is 1099
  - The name is a URL format and can be prefixed with a hostname and port: "`//localhost:1099/Server`"

Server code: part 1 (SampleServer.java)

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class SampleServer {
    public static void main(String args[]) {
        if (args.length != 1) {
            System.err.println("usage: java SampleServer rmi_port");
            System.exit(1);
        }
```


Server code: part 2 (SampleServer.java)

```
        try {
            // first command-line arg: the port of the rmiregistry
            int port = Integer.parseInt(args[0]);

            // create the URL to contact the rmiregistry
            String url = "//localhost:" + port + "/Sample";
            System.out.println("binding " + url);

            // register it with rmiregistry
            Naming.rebind(url, new Sample());
            // Naming.rebind("Sample", new Sample());
            System.out.println("server " + url + " is running...");
        }
        catch (Exception e) {
            System.out.println("Sample server failed:" +
                                                e.getMessage());
        }
    }
}
```


Policy file

- When we run the server, we need to specify security policies
- A security policy file specifies what permissions you grant to the program
- This simple one grants all permissions

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


The client

- The first two arguments will contain the host & port
- Look up the remote function via the name server
- This gives us a handle to the remote method

```
SampleInterface sample = (SampleInterface)Naming.lookup(url);
```

- Call the remote method for each argument

```
sample.invert(args[i]));
```

- We have to be prepared for exceptions

Client code: part 1 (SampleClient.java)

```
public class SampleClient  {
  public static void main(String args[]) {
    try {
      // basic argument count check
      if (args.length < 3) {
        System.err.println(
          "usage: java SampleClient rmihost rmiport string... \n");
        System.exit(1);
      }

      // args[0] contains the hostname, args[1] contains the port
      int port = Integer.parseInt(args[1]);
      String url = "//" + args[0] + ":" + port + "/Sample";
      System.out.println("looking up " + url);

      // look up the remote object named "Sample"
      SampleInterface sample = (SampleInterface)Naming.lookup(url);
```


Client code: part 2 (SampleClient.java)

```
    // args[2] onward are the strings we want to reverse
    for (int i=2; i < args.length; ++i)

          // call the remote method and print the return
          System.out.println(sample.invert(args[i]));

    } catch(Exception e) {
      System.out.println("SampleClient exception: " + e);
    }
  }
}
```

Compile

- Compile the interface and classes:

```
javac SampleInterface.java Sample.java
javac SampleServer.java
```

- And the client…

```
javac SampleClient.java
```

(you can do it all on one command: `javac *.java`)

- Note – Java used to use a separate RPC compiler
  - Since Java 1.5, Java supports the dynamic generation of stub classes at runtime
  - In the past, one had to use an RMI compiler, *rmic*
  - If you want to, you can still use it but it's not needed

Run

- Start the object registry (in the background):

```
rmiregistry 12345 &
```

  - *An argument overrides the default port 1099*
- Start the server (telling it the port of the rmi registry):

```
java -Djava.security.policy=policy SampleServer 12345
```

- Run the client:

```
java SampleClient svrname 12345 testing abcdefgh
```

  - Where svrname is the name of the server host
  - 12345 is the port number of the name server: rmiregistry, not the service!
- See the output:

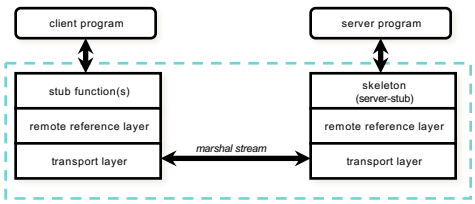
```
gnitset
hgfedcba
```


RMI
A bit of the internals

Interfaces

- Interfaces define behavior
- Classes define implementation
- RMI: two classes support the same interface
  - client stub
  - server implementation

Three-layer architecture



Server - 1

- **Server creates an instance of the server object**
  - extends UnicastRemoteObject
  - TCP socket is bound to an arbitrary port number
  - thread is created which listens for connections on that socket
- **Server registers object**
  - RMI registry is an RMI server (accepts RMI calls)
  - Hands the registry the client stub for that server object
    - contains information needed to call back to the server (hostname, port)

Client - 1

- **Client obtains stub from registry**
- **Client issues a remote method invocation**
  - **stub class creates a RemoteCall**
    - opens socket to the server on port specified in the stub
    - sends RMI header information
  - **stub marshals arguments over the network connection**
    - uses methods on RemoteCall to obtain a subclass of ObjectOutputStream
    - knows how to deal with objects that extend java.rmi.Remote
      - serializes Java objects over socket
  - **stub calls RemoteCall.executeCall()**
    - causes the remote method invocation to take place

Server - 2

- **Server accepts connection from client**
- **Creates a new thread to deal with the incoming request**
- **Reads header information**
  - creates RemoteCall to deal with unmarshaling RMI arguments
- **Calls *dispatch* method of the server-side stub (skeleton)**
  - calls appropriate method on the object
  - sends result to network connection via RemoteCall interface
  - if server threw exception, that is marshaled instead of a return value

Client - 2

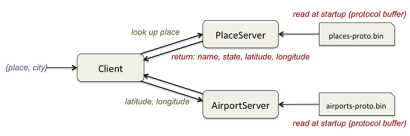
- The client unmarshals the return value of the RMI
  - using RemoteCall
- value is returned from the stub back to the client code
  - or an exception is thrown to the client if the return was an exception

Part 2: Project Overview

Paul Krzyzanowski
Rutgers University
Fall 2016

Assignment Summary

- Find the five airports closest to a given location
- One Client
- Two Servers
  - Place Server: get information about a location (latitude, longitude)
  - Airport Server: find airports near a given latitude, longitude
- Data is stored in Google Protocol Buffer format
  - Each server reads it at startup



Assignment

- The assignment uses Java RMI
- It does not have to be multithreaded
- You may work in groups up to 4
  - The larger the group, the more polished I expect your work to be
  - Group size > 1: submit a beautiful-looking project report
- The assignment is due on Sunday October 16
  - Start early
  - During this time, you will also have written assignments and an exam

Implementation hints

Key Components

- The amount of code you will write is very small
- There are three parts that you need to get working
  1. Reading the places and airports databases
  2. Client-server communication
  3. Computing distances
- Any of these, especially 1 & 2, might cause confusion
- Start early
- Solve ONE problem at a time
- Then put it all together

Google Protocol Buffers

- Go through the tutorial
  - https://developers.google.com/protocol-buffers/docs/javatutorial
- Download pre-built protocol buffer compiler from:
  - https://github.com/google/protobuf/releases
  - For example:
    - protoc-3.0.2-osx-x86_64.zip
    - protoc-3.0.2-linux-x86_64.zip
  - This you will get the protocol buffer compiler in **bin/protoc**.
  - You can also build from source

Step 1

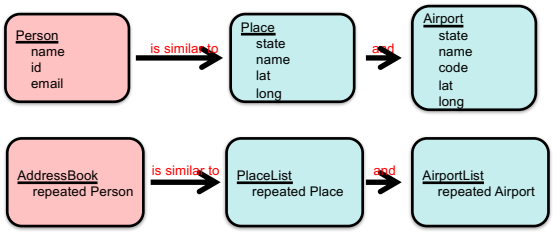
- Make sure you can read the Google Protocol Buffer files
- Download or build:
  - Protocol Buffer compiler: protoc
  - A bunch of Java support classes
    - You can assemble them into one file: protobuf.jar

```
$ cd protobuf-3.0.2/java/core/src/main/java
$ protoc --java_out=core/src/main/java -I../src \
    ../src/google/protobuf/descriptor.proto
$ javac *.java
$ jar cvf protobuf.jar com/google/protobuf
```

  - Or download protobuf.jar from the assignment link
- Go through the tutorial – *ignore the assignment for now*
  - See the link: *Try the tutorial for your favorite language*

Step 1a: Tutorial

- The tutorial is in the *examples* directory in the source package
- The example is similar to what is needed for the assignment
  - Similar structures and examples of reading (and writing)
- If you cannot do the tutorial, you will not be able to do the assignment!



Step 1b: Test program: Places

- Write a small program to read and print the list of places

```
PlaceList pl = PlaceList.parseFrom(new FileInputStream(fname));
for (Place p: pl.getPlaceList()) {
    System.out.println(
        "state: " + p.getState() + " "
        + "place: " + p.getName() + " "
        + "lat: " + p.getLat() + " "
        + "lon: " + p.getLon());
}
```

- Make sure protobuf.jar is in your CLASSPATH
- You should see output like

```
state: AL place: Abbeville city lat: 31.566367 lon: -85.2513
state: AL place: Adamsville city lat: 33.590411 lon: -86.949166
state: AL place: Addison town lat: 34.200042 lon: -87.177851
state: AL place: Akron town lat: 32.876425 lon: -87.740978
```

Step 1c: Test program: Airports

- Write a small program to read and print the list of airports

```
AirportList al = AirportList.parseFrom(new FileInputStream(fname));
for (Airport a: al.getAirportList()) {
    System.out.println(
      "state: " + p.getState() + " " + "name: " + p.getName() + " "
      + "code: " + p.getCode() + " " + "lat: " + p.getLat() + " "
      + "lon: " + p.getLon());
}
```

- Make sure protobuf.jar is in your CLASSPATH
- You should see output like

```
state: AL name: Anniston code: ANB lat: 33.58 lon: -85.85
state: AL name: Auburn code: AUO lat: 32.67 lon: -85.44
state: AL name: Birmingham code: BHM lat: 33.57 lon: -86.75
state: AL name: Centreville code: CKL lat: 32.9 lon: -87.25
```


Step 2a: Write a skeletal standalone program

- You know you can read the protocol buffer data
- Don't worry about RMI for now
- Write standalone programs
  - Create *Places* and *Airports* classes (pick names you like)
  - Places
    - Constructor reads in the places database
    - main() can be a test function that takes a place name, looks it up, and prints results
  - Airports
    - Constructor reads in the airports database
    - main() can initially be a test function that looks up an airport

Step 2b: Refine the skeletal program

- Modify your *Airports* main() to look for closest airports
- Take latitude & longitude as parameters
- Find the 5 closest airports
  - Use the formula in the assignment to compute great circle distance
    $d = 60 \cos^{-1}(\sin(lat_1) \sin(lat_2) + \cos(lat_1) \cos(lat_2) \cos(lon_2 - lon_1))$
  - You don't need a clever algorithm
    - Just go through the list of airports
    - Compute the distance
    - See if each new distance should displace your list of *n* shortest distances
  - Print the results
    - Check that the results look right!

Step 3a: Make sure you can use RMI

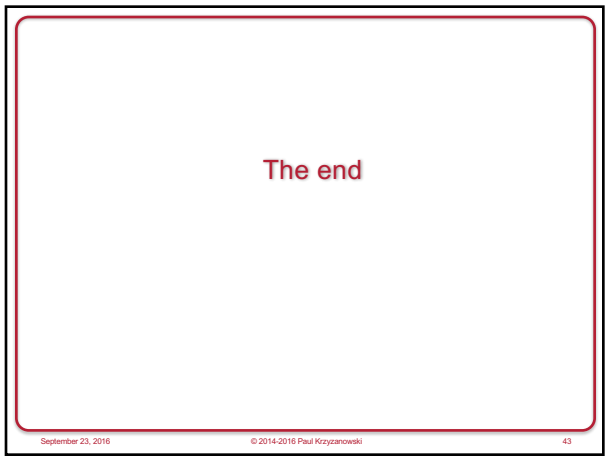
- Again, ignore the assignment for now
- Download the RMI sample program
- Compile and run it
  - This will make sure you have no problems with RMI
  - … and no problems with CLASSPATH

Step 3b: Define Interfaces

- Define interface
- AirportsInterface (pick a name)
  - takes latitude & longitude and returns a list of airport info structures
- PlacesInterface (pick a name)
  - takes a place name and returns latitude & longitude

Step 3b: Create servers, client & add RMI

- Create servers for Airports & Places
  - Copy the sample RMI server
  - All it does is
    - Get a port from the command line
    - Instantiate the class
    - Register it with rmiregistry
- Your client will:
  - Call Naming.lookup to look up the Places & Airport servers
  - Places p = places.findplace(place_name)
  - AirportInfo closest[] airports.nearest(p.lat, p.long)
  - Iterate through the list and print the results


The end
September 23, 2016
© 2014-2016 Paul Krzyzanowski
43