Remote Method Invocation

Prof. Dr. Ralf Lämmel
Universität Koblenz-Landau
Software Languages Team

Non-101samples available here:
https://github.com/101companies/101repo/tree/master/languages/AspectJ/javaRmiSamples

http://101companies.org/wiki/
Contribution:javaRmi
Elevator speech

What if the objects of interests are no longer located on the same computer, i.e., the objects are distributed instead? How to communicate between different computers in an OOP-friendly manner? Basically, we want to continue to hold references to objects (perhaps remote objects), and send them messages (i.e., perform method calls) as before.

We will be looking into Java RMI as a technology of choice. Alternative options include Corba, WebServices, and SOA.
Non-distributed programming

Caller and callee are on the same machine.

Object1 (Client)  Method Invocation  Object2 (Server)

- Arguments are evaluated.
- Caller location is pushed onto stack.
- Callee method is executed.
- Result is returned.
- Caller location resumes.

MyMachine

(C) 2010-2013 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)
Distributed programming
Caller and callee are on different machines.

In demo code, client and server may be on the same machine, perhaps even on the same JVM, but the invocation is handled (conceptually) over the network.
A first look at the “hello world” of RMI

DEMO

See package “helloworld” of javaRmiSamples (see 1st slide). Start the server as described in “Makefile”. Start the client by running the “Client” class. Observe how 42 was remotely incremented.
A more interesting scenario: a banking application to access account data remotely at a teller (ATM).
Communication between different Java runtimes on the same machine or different machines, eventually involves the network layer of the OS.
Stubs and skeletons

- Client invokes a remote method on “stub” object.
- The stub sends the call to the server-side “skeleton”.
- That is, the stub:
  - opens a socket to the remote server,
  - marshals the method parameters,
  - forwards the data stream to the skeleton,
  - awaits and unmarshals result data stream from skeleton.
- The skeleton contains a method that:
  - receives the remote calls,
  - unmarshals the method parameters,
  - invokes the actual remote (by now, local) implementation, and
  - marshals the result and delivers it back to the stub.
Remote-method invocation

Object1 (Client)

Remote Object Stub: serves as Remote Reference

odd(3) serves as Remote Reference

true

Host mymac.foo.edu

01101

10010

true

Remote Object Skeleton

odd(3)

true

Object2 (Server)

Host myserver.bar.edu

Same interface as remote object

Serialized method name, parameters and return value
Parameters and return values

- Remote objects – by reference
- Serializable objects – by copy
- Others – cannot be passed (exception)
How to refer to / look up remote objects?

We need a naming service! A server-side directory that associates remote objects with names (URLs)

<table>
<thead>
<tr>
<th>Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>“X”</td>
</tr>
<tr>
<td>“Y”</td>
</tr>
<tr>
<td>“Z”</td>
</tr>
</tbody>
</table>

Remote Object A

Remote Object B

Remote Object C

Host myserver.bar.edu
Naming service cont’d

Host mymac.foo.edu

lookup("Y")

Remote reference to server

Host myserver.bar.edu

Naming

"X"

"Y"

"Z"

ObjectY
Designing a client/server app
(Summary)

- Server
  - Bind 1+ “service” objects via registry
- Client
  - Looks up and interacts with service objects
- For example (see “helloworld” demo):
  - Service: increment operation
Designing a client/server app  
(Detailed steps)

1. Design an interface for service (remote object).
2. Implement the service.
3. (Generate stub and skeleton classes.)
4. Implement a server to contact for binding.
5. Implement a client to invoke service.
6. Start the server.
7. Run the client.
Step 1:
Design an interface for service (remote object).

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

public interface Service extends Remote {
    public long inc(long x)
        throws RemoteException;
}
```

This is a regular interface except for the special base interface and the special throws declaration.
Step 2: Implement the service.

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

public class ServiceImpl
    extends UnicastRemoteObject
    implements Service {

    // Needed for serialization
    private static final long serialVersionUID = 6102178242852627613L;

    // Needed because of exception
    public ServiceImpl() throws RemoteException {
        super();
    }

    public long inc(long x) throws RemoteException {
        return ++x;
    }
}
```
Step 4: Implement a server to contact for binding.

```java
import java.rmi.Naming;

public class Server {
    public Server() {
        try {
            Service s = new ServiceImpl();
            Naming.rebind("rmi://localhost/Service", s);
        } catch (Exception e) {
            System.out.println("Trouble: " + e);
        }
    }

    public static void main(String args[]) {
        new Server();
    }
}
```

Remote object

Register with naming service
Name format for binding

\[ \text{rmi://<host_name>} \]
\[ [:<name_service_port>] \]
\[ //</service_name> \]

Thus, URLs of a specific form are used for addressing remote objects.
Availability of registry

- Binding requires that the registry is locally running.
- Registry can be started programmatically as follows:

  ```java
  import java.rmi.registry.LocateRegistry;
  import java.rmi.registry.Registry;
  LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
  ```

- Default TCP/IP port: 1099
Usage of “localhost”

• localhost may fail to work in binding.
• This depends on the network configuration.
• Alternatively, one can use the following idiom:

```java
InetAddress addr = InetAddress.getLocalHost();
String hostname = addr.getHostName();
Naming.rebind("rmi://"+hostname+"/Service", s);
```
Step 5: 
Implement a client to invoke service.

```java
import java.rmi.Naming;

public class Client {

    public static void main(String[] args) {
        try {
            Service s = (Service) Naming.lookup("rmi://localhost/Service");
            System.out.println(s.inc(41));
        } catch (Exception e) {
            ...
        }
    }
}
```
Step 6: Start the server.

JavaRmi> java helloworld.Server

One may want to run the server (and the client) with a security policy.

Server must be killed, e.g., with CTRL-C

If you are using Eclipse, you cannot (easily) run both server and client simultaneously from within Eclipse. One of the two would be started from the command line.
Step 7:
Run the client.

JavaRmi> java helloworld.Client
42
Components of an RMI-based application

- Services of remote objects (interfaces)
- Implementation of services (classes)
- Proxies for client-side (classes, generated)
- A server that binds services (class)
- A client that looks up services (class)
- The RMI registry (component of RMI layer)
- Security policies for client and server (optional)
Security of RMI

• Useful constrains:
  – Service provision (“May an app provide services?”)
  – Naming lookups (“May an IP address look up services?”)
  – Remote invocations (“May an IP address invoke ...?”)

• Use a non-default security manager:
  System.setSecurityManager(new SecurityManager());

• Assign a security policy to an application:
  java -Djava.security.policy=mySecurity.policy Server
An RMI-based banking app

DEMO

– A non-C/S-based version as a reference: package “banking.local”.
– The C/S-based version: package “banking.remote”.
– Think of a refactoring to derive the latter from the former.
This implementation is interesting in so far that it readies all data objects for RMI. Further, the operation “total” is provided as a service, but “cut” is not. Thus, the client must implement “cut”, which essentially means that all company, department, and employee objects end up as proxies on the client.
Summary

• RMI easily blends with OOP.
  – A simple form of distributed programming is enabled this way.
  – Client/Server applications are enabled this way.

• RMI – semantics and concepts:
  – Use local proxy objects for access to remote objects.
  – Un-/marshal arguments and results for messages on the wire.
  – Bind objects to names (URLs) that can be used for lookup.

• RMI – programming idioms:
  – Remote objects are looked up from server-side registry.
  – Remote objects may also be returned by RMI calls.
  – Remote objects may be created by factory methods.

• Omissions:
  – Multiplicity of clients
  – Other distribution technologies: Corba, WebServices, SOA, HTTP, ...