Remote Objects and RMI

Instructor: Dr. Tongping Liu

Outline
- Distributed/Remote Objects
- Remote object reference (ROR)
- Remote Method Invocation (RMI)
- Case study and example: Java RMI
- Other issues for remote objects
  - Factory method; Transient vs. Permanent objects;
  - Callback objects; Distributed Garbage collection;

Object-Oriented Programming
- Fundamental idea: encapsulation
  - both data and methods
- Interfaces – define the signature of a set of methods
- Object reference – access object’s data/method
- Method invocation – may affect target objects; may need chain invocation
- Data/variable – could be directly access
- Exceptions – thrown when an error occurs

Distributed Object

Client process:
- disObj1.m1(…);
disObj1.m2(…);
disObj2.m3(…);
disObj2.m4(…);

Machine a

Machine b

Machine c

Obj: m3; m4;

Distributed Object

Distributed Object

Client machine
Remote Objects vs. Message Passing

Client:
Sum = math.add(n1, n2);

Server:
int state;
object math;
int add(int n1, int n2)
{
… …
}

Create a socket
Connect it to server
Put n1, n2 in a msg
Send msg to server
Read/wait reply msg
Extract result from the msg

Create a socket
Bind it to a port
Accept a connection
Read/wait for a msg
Extract n1, n2 from the msg
Compute result
Put it in a reply msg
Send reply msg to client

Remote Object Model

- Remote objects -- can receive remote invocations; having state information.
- Remote object reference – identify remote objects in distributed environments
- Remote interface -- specifies methods to be invoked remotely
- Process contains objects (local/remote)
  - Local objects: accept only local invocations
  - Remote object: accept both local/remote invocations
  - Remote invocation – different processes (same or different hosts)
- Exceptions – application level

An Example: Remote Object

- Object may implement both remote & local interface
  - Other processes: invoke only methods in remote interface
  - Same process: local object reference and invoke methods in local interface

Objects and Method Invocations

- Objects receiving remote invocations (service objects) are remote objects, e.g., B and F
- Object references are required for invocation, e.g., C must have E’s reference or B must have D’s reference

What are the local object references?
Outline
- Distributed/Remote Objects
- Remote object reference (ROR)
- Remote Method Invocation (RMI)
- Case study and example: Java RMI
- Other issues for remote objects
  - Factory method; Transient vs. Permanent objects;
  - Callback objects; Distributed Garbage collection;

Remote Object Reference (ROR)
- **Uniquely** identify remote objects in distributed systems
- **Invoke remote methods of a remote object**
- Remote object references may be passed as input arguments or returned as output arguments.

Compared with ordinary object reference, what additional information is needed for remote object reference?

<table>
<thead>
<tr>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet address</td>
<td>port number</td>
<td>time</td>
<td>object number</td>
<td>interface of remote object</td>
</tr>
</tbody>
</table>

Interfaces
- Interface for local objects
  - Specify methods and/or data that can be accessed
  - Do not specify an implementation
- Interface for remote objects
  - Specifies methods for remote invocation
  - **Input** and **output** parameters are also specified and parameters may be objects

Parameters for Remote Methods
- **Primitive types**
  - pass by value
- **Ordinary objects**
  - passed by copy (e.g. using Java serialization; the object must implement the java.io.Serializable interface).
- **Remote objects**
  - use the remote object reference (ROR)

**Why not pass a copy of the remote object?**

1. The copy of a remote object can not perform its function
2. The object proxy and implementation is auto downloaded
Outline

- Distributed/Remote Objects
- Remote object reference (ROR)
- Remote Method Invocation (RMI)
- Case study and example: Java RMI
- Other issues for remote objects
  - Factory method; Transient vs. Permanent objects;
  - Callback objects; Distributed Garbage collection;

RPC Review

RMI Overview

- How do clients know where the remote objects are?
- Binding…
  - RMI register: the string name of the object, the remote object itself
  - The registry returns to the caller a reference (called stub) to the remote object.
  - Invoke methods on the object (through the stub).

Client Side: HelloClient

```java
//HelloClient.java
import java.rmi.*;
import java.rmi.server.*;
public class HelloClient{
    public static void main(String[] argv) {
        try {
            HelloInterface hello = (HelloInterface) Naming.lookup("/localhost/Hello");
            System.out.println(hello.say());
            System.out.println(hello.add("Here is added information!!!"));
        } catch (Exception e) {
            System.out.println("HelloClient exception: "+ e);
        }
    }
```
**RPC vs. RMI**

**Similarity:** Marshaling and parameter passing

**Difference:**
- RPC: C based, structure based semantics. RMI: java and object-oriented
- RPC: call remote functions, passed everything. RMI: remote/system-wide object reference and invoke methods. We can also pass and return objects that can be distributed among many JVM instances, much more powerful.
- RMI can support dynamic invocations: a client can call a remote procedure even if the signature of the remote procedure or the name of the service are unknown until runtime.

---

**Architecture to Support Remote Objects**

**Remote Reference Module (RRM)**

**Server side:**
- Create remote object reference (ROR)
- Maintain remote object reference table
- Entry for remote objects
- Map between remote reference and local reference

**Client side:**
- Create proxy object when first get ROR
- Maintain remote object reference table
- Entry for local proxy (client side)
- Map between remote reference and local reference

**Communication Module**

- Carry out request-reply protocol
- Provide certain invocation semantics
  - Retry request
  - Duplication message filtering
  - Reply message history cache
- Interact with remote reference module (server side)
  - Get remote object’s local reference
  - Pass message and local reference to appropriate dispatcher on server side
Middleware for Remote Objects

- Layer between application and communication/remote reference modules
- Automatically create proxy, skeleton and dispatcher from remote interface definition
- Client side: one proxy for each remote object
  - (kind-of) Implement the methods in remote objects, provide the transparency to the client
- Server side: one dispatcher and one skeleton
  - Dispatcher accepts message and select appropriate method in the skeleton: methodID
  - Skeleton: Marshall/unmarshall messages and invokes corresponding method in the remote object

Server/Client Programs and Binder

- Server program
  - Dispatcher, skeleton
  - Servant class: implement methods for remote objects
- Client program
  - Proxy
  - Use binder to get remote object reference
- Binder: kind of naming service
  - Mapping between text name and remote object reference
  - System wide register/look up service
How to Use Remote Objects: Server Side

- Step 0: start binder
- Step 1: server start communication and remote reference module
- Step 2: server create remote objects, add them to remote object table, and obtain remote object reference from remote reference module
- Step 3: server publish the remote objects to naming service → bind the remote object reference with a name; wait for invocation requests

How to Use Remote Objects: Client Side

- Step 4: client start communication and remote reference module
- Step 5: client contact naming service for desired remote object reference
- Step 6: client remote reference module create proxy
- Step 7: client call methods in proxy → mashall parameters
- Step 8: client locate remote object through remote reference module
- Step 9: Send method invocation request (contain remote object reference) through communication module
**How to Use Remote Objects: Server Side**

- Step 10: server get invocation requests (contains remote object reference) through communication module
- Step 11: server consult with remote reference module and get local reference for the remote object
- Step 12: server hand the request to the dispatcher/skeleton of the remote object’s class \( \text{which method (method id)} \)
- Step 13: server call method in skeleton \( \text{unmarshall parameters in request and invoke the real method of the remote object} \)
- Step 14: perform operation in remote object and return results to skeleton
- Step 15: server marshall results in skeleton and send out message (remote object reference) through communication module

**How to Use Remote Objects: Client Side**

- Step 16: get result message (contain remote object reference) in communication module
- Step 17: obtain proxy reference from remote object reference and hand the result message to the proxy
- Step 18: unmarshall the results in proxy and return results to the caller

Complete a Remote Method Invocation

Why do people say RMI is simpler than sockets?

RMI = Sockets + Object Serialization + Some Utilities
User don’t need to worry about communication

**Outline**

- Distributed/Remote Objects
- Remote object reference (ROR)
- Remote Method Invocation (ROR)
- Case study and example: Java RMI
- Other issues for remote objects
  - Factory method; Transient vs. Permanent objects;
  - Callback objects; Distributed Garbage collection;
Case Study: Java RMI

- Define a remote interface
  ```java
  public interface HelloInterface extends Remote {
      public String add(String s) throws RemoteException;
      public String say() throws RemoteException;
  }
  ```

- Server side:
  - Implement the interface `Hello.java`
  - Develop the server `HelloServer.java`

- Client side:
  - Develop a client `HelloClient.java`

- Run the RMI registry, the server, and the client

Servant Class: Implement Remote Interface

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

public class Hello extends UnicastRemoteObject implements HelloInterface {
    private String message;
    public Hello (String msg) throws RemoteException {message = msg; }
    public String add(String more) throws RemoteException{
        message = new String (message + more); return message;  }
    public String say( ) throws RemoteException {return message; }
}
```

Server: Create Servant Object and Bind

```java
//HelloServer.java
import java.rmi.*;
public class HelloServer{
    public static void main(String args[]){
        try {
            Naming.rebind ("Hello", new Hello("Hello, world!");
        System.out.println ("Hello Server is ready.");
        } catch (Exception e) {
        System.out.println ("Hello Server failed: "+ e);
    }
}
```

The binder: RMI Registry

- For server
  - Void `rebind` (String name, Remote obj)
  - Void `bind` (String name, Remote obj)
  - Void `unbind` (String name, remote obj)

- For Client
  - Remote `lookup` (String name)
  - String[] `list()`
Run the Server

- Compile the interface and remote class
  - `javac HelloInterface.java` and `Hello.java`
- Compile server
  - `javac HelloServer.java`
- Generate skeletons & stubs (old Java compiler)
  - `rmic Hello` to `Hello_Skel.class` and `Hello_Stub.class`
- Start RMI registry
  - `rmiregistry` (default at port 1099)
- Start Hello server
  - `java HelloServer`

Implementations of Client

- First, get remote object reference
  - It can retrieve a remote object reference from `RMiregistry` on the machine where the remote object resides
  - It received a remote object reference from a previous call
- Invoke methods specified in remote interface
- Compile the client code
  - Needs the interface file in order to compile

Client Side: HelloClient

```java
//HelloClient.java
import java.rmi.*;
import java.rmi.server.*;
public class HelloClient {
    public static void main(String[] argv) {
        try {
            HelloInterface hello = (HelloInterface) Naming.lookup("/localhost/Hello");
            System.out.println(hello.say());
            System.out.println(hello.add("Here is added information!!!");
        } catch (Exception e) {
            System.out.println("HelloClient exception: "+ e);
        }
    }
}
```

Java RMI: Compile and Run the Client

- Compile the interface class
  - `javac HelloInterface.java`
- Compile client
  - `javac HelloClient.java`
- Start Hello client
  - `java HelloClient`
Summary of Java RMI Steps

- Design the remote interfaces being implemented
- Server create and to register the remote objects.
  - Implement remote methods in the remote interfaces
  - Create and install a security manager
    - Regulates whether the server itself might download classes when accessing other remote servers
  - Create one or more instances of a remote object
  - Register at least one of the remote objects with the RMI remote object registry (or another naming service)
- Client program to use the methods of remote object
  - Looks up server in remote object registry and gets ROR
  - Use normal method call syntax for calling remote method

Other Issues for Remote Objects

- When/who creates remote objects?
  - Remote Object Factory method
- Can server contact client?
  - Call-back

Outline

- Distributed/Remote Objects
- Remote object reference (ROR)
- Remote Method Invocation (RMI)
- Case study and example: Java RMI
- Other issues for remote objects
  - Factory method; Transient vs. Permanent objects;
  - Callback objects; Distributed Garbage collection;