Remote Method Invocation (RMI) and Distributed Observers in Java

Theodore Norvell

The Proxy Pattern

- The client calls the proxy, which forwards the call (somehow) to the actual subject.

RMI and the Proxy pattern

- RMI uses the Proxy pattern to distribute objects across a network.
- Recall that in the Proxy pattern a proxy and a subject share a common interface.
- In RMI, objects call methods in a proxy (aka stub) on its own machine
  - The proxy sends messages across the network to a “skeleton” object
  - The skeleton calls the subject object.

One Remote Method Call.

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Client calls stub</td>
</tr>
<tr>
<td>1</td>
<td>Stub messages skeleton</td>
</tr>
<tr>
<td>2</td>
<td>Skeleton calls server (subject)</td>
</tr>
<tr>
<td>3</td>
<td>Call returns</td>
</tr>
<tr>
<td>4</td>
<td>Skeleton messages proxy</td>
</tr>
<tr>
<td>5</td>
<td>Call returns</td>
</tr>
</tbody>
</table>

Issues

- Concurrency
  - If there are multiple clients, the server may field multiple calls at the same time.
  - So use synchronization as appropriate.
- Argument passing
  - Arguments are passed by value or “by proxy” not by reference.
- Proxy and Skeleton generation
  - Proxy and Skeleton classes are automatically derived from the server class (program rmic)
- Lookup
  - Objects are usually found via a registry (program rmiregistry)
**Nitty-Gritty**

- The proxy and the server share an interface.
  - This interface must extend java.rmi.Remote.
  - Every method in the interface should be declared to throw java.rmi.RemoteException.
  - RemoteExceptions are thrown when network problems are encountered, or when server objects no longer exist.
- The server typically extends class java.rmi.server.UnicastRemoteObject.
  - The constructor of this class throws a RemoteException.
  - Therefore, so should the constructor of any specialization.

**Argument Passing Revisited**

- Most arguments and results are converted to a sequence of bytes; the bytes are sent over the net.
  - Therefore the class should implement the java.io.Serializable interface.
  - A clone of the argument/result is constructed on the other side.
  - The effect is pass by object value, rather than by object reference.
- But objects that extend java.rmi.server.RemoteObject instead have a proxy constructed for them on the other side.
  - I call this “pass by proxy”. Essentially pass by reference.
  - So each argument, result, exception type should be a primitive type, Serializable, or extend RemoteObject.

**An Example – Distributed Othello**

- Othello is a two person board game.
- It uses the observer pattern so that:
  - when the (game state) model changes, all observers are informed of the change.
- I wanted to put the model on one machine and the observers on other machines.
- Hence I implemented the observer pattern with RMI.
- This is example othello-2 on the website.

**The Observer Pattern**

- Subject alerts Observers of changes of state.
  - notifyObservers:
    - for all o in Observers
      - o.update()
- Observer
  - update()
- Observable
  - addObserver(Observer)
  - removeObserver(Observer)
  - notifyObservers()
- Subject
  - getState()
  - setState()

**Conceptual Model for Othello**

- DoubleBufferedView
- Animator
- RemoteGameModel
- Server Host
- Client Host 0
- Client Host 1
- Observes
  - DoubleBufferedView
  - Animator
  - RemoteGameModel
**Detailed View**

- **DoubleBufferedView**
  - Observes
  - Animator
  - Skeleton

- **Remote Game Model’s Proxy**
  - RemoteGameModel
  - Animator Proxy

**The Observer Pattern in Othello**

- **Observer**
  - Subject
  - RemoteGameModel (instance of model)

- **RemoteGameModelInterface**
  - RemoteGameModelInterface:getGameState()

- **RemoteConcurrentObserver**
  - RemoteConcurrentObserver:observe()

**Proxies for the Remote Game Model**

- Generated Proxy Class
  - RemoteGameModel_Stub
  - RemoteGameModel

- Animator
  - update()

- Remote Game Model’s Proxy
  - Communicates with
    - Skeleton
    - RemoteGameModel

**Proxies for the Animator**

- Generated Proxy
  - Animator_Stub
  - update()

- Remote Game Model’s Proxy
  - Communicates with
    - Skeleton
    - RemoteGameModel

**Animator / RemoteGameModel Relationship**

- Animator
  - update()

- Remote Game Model’s Proxy
  - Communicates with
    - Skeleton
    - RemoteGameModel

**Typical sequence (slightly simplified).**

- A remote client calls a synchronized mutator on the RemoteGameModel via its stub & skeleton
  - The RemoteGameModel updates its state and notifies each Animator via its stubs & skeletons.
  - The Observers (Animator objects) call the RemoteGameModel accessor “getPieceAt” via its stubs and skeleton.
  - Therefore this accessor must not be synchronized!
  - getPieceAt returns
  - The update routines return.
  - The original mutator call returns and the RemoteGameModel becomes unlocked.
Concurrent Notification

- The previous sequence is slightly simplified.
  - In fact the Animators return from update immediately (so that all can be informed essentially at the same time).
    - The Animation threads will inform the RemoteGameModel when they have completed their animation.
  - The RemoteGameModel waits until it has been informed that all animations are complete.
    - The effect is that the animations can happen concurrently, yet the RemoteGameModel does not unlock until all animations are complete and all models agree on the board state.
    - See RemotelyConcurrentlyObservable for details.

Finding the Server

- Normally an object’s address serves as a unique identifier.
  - But this only makes sense in the context of a given JVM process.
  - We would like objects to have unique identifiers that are unique in the world.
  - The miregistry allows you to give a URI to an object.
    - URIs are: rmi://host/name
  - The host must be running a miregistry process and that process should have the appropriate class files on its CLASSPATH

Finding the Server (cont.)

- The main routine for the server
  - The static method bind in Naming gives a URI to gameModel
    ```java
    public static void main(String[] args) { 
      try {
        RemoteGameModel gameModel = new RemoteGameModel();
        String name = args[0] ;
        Naming.bind(name, gameModel );
        System.err.println("Game model bound to "+name);}
      catch(java.net.MalformedURLException e) { … }
      catch(AlreadyBoundException e) { … }
      catch(RemoteException e) { … }
    }
    ```

Finding the Server (cont.)

- The clients obtain a proxy for the game model using Naming.lookup(URI)
  - From ClientMain.java
    ```java
    public static void main(String[] args) {
      try {
        String name = args[0] ;
        remoteGameModelInterface = (RemoteGameModelInterface)
        Naming.lookup(name) ;
      }
      catch(java.net.MalformedURLException e) { … }
      catch(NotBoundException e) { … }
      catch(RemoteException e) { … }
    }
    ```

Finding the Server (cont.)

- The client then can use the proxy. E.g.
  - Continuing the ClientMain main routine
    ```java
    Animator animator0 = null ;
    try {
      animator0 = new Animator( proxy ) ;
      animator0.addObserver(this) ;
    }
    ```

Hooking up the Observer.

- Each Animator calls addObserver(this) on the RemoteGameModel’s proxy.
  - Since Animator extends RemoteObject, it is passed by proxy, meaning
    - a proxy for the Animator is constructed in the JVM of the RemoteGameModel.
  - This allows the game model to call-back to the Animators to notify them of any changes to the game’s state.
The final hookup (again)

Creating the stubs and skeletons

A Few Words of Warning

A Few Words of Warning (cont.)