Comp 533 - Assignment 2: Object Replication Using RMI Synchronous Remote Procedure Calls

Date Assigned: Thu Feb 2, 2017

Completion Date: Thu Feb 16, 2017, 11:55pm

The goal of the assignment is to show you the difference between two IPC mechanisms that are at the extremes of the IPC mechanism in performance and programmability: NIO and RMI. You will learn how to use mechanisms provided by RMI for object distribution to create an application-specific mechanism for object replication.

Distributed Functionality

You will implement the functionality of the previous assignment, but will use Java RMI rather than Java NIO for communication. This means you must, as in the previous assignment:

- Implement non atomic and atomic broadcast using a server.
- Measure the performance of both kinds of broadcast, and compare it with unicast.

As in the previous assignment, you will not change the non-distributed simulation and each client will have a replicated copy of the simulation that is kept consistent by relaying messages through the server, which does not have a copy of the simulation.

In the previous assignment, you could choose one of several ways to change the broadcast method. In this assignment, each client must provide a user-interface to change the broadcast scheme. Moreover, you should process a change to the broadcast mode in one client by atomically broadcasting the new mode to all other processes (server and other clients), all of which then change the mode. RMI should make this approach easy to implement.

Something not mentioned in the recorded PPT lecture.:When a client passes an instance, I, of Remote as a parameter to a remote method, a proxy to I is passed to the server only if I has been exported using the UnicastRemoteObject.exportObject() call.

Googling for Java RMI Tutorial will give you several hits. In the GIPC package examples.rmi.counter.simple is a simple example with a server, ASimpleRMIRegistryAndCounterServer, and client, ASimpleRMICounterClient. The example is simple in that it does not involve a callback object, and unlike the class

examples, bundles the registry with the server. Run the server and client to see the result and use breakpoints to trace the remote calls.. Together, the two main classes define a distributed program in which the counter client looks up proxies to an object registered by the counter server, and invokes two methods on the registered object. You can use this program as a template, or you can use one a tutorial example you find on the Web.

Submission Instructions

As in the previous assignment, create a YouTube video demonstrating the working of your program and following the constraints discussed below. Submit your code on Sakai (the assignment will be created before the due date). See assignment 1 on how to create the video using Office Mix.

Basic Demo with Breakpoints

As in the previous assignment, your demo should involve at least three simulation processes.

Use breakpoints and the Debug window in your programming environment to demonstrate RMI calls. This means you must provide input to one simulation and use breakpoints to show:

- The inputting simulation make a remote method call in the server.
- The server executing the call.
- The server making a remote-method call in a non-inputting client.
- A non-inputting client executing the method.

Atomic and Non Atomic Broadcast

As in the previous assignment choose two non-commuting operations to show that atomic broadcast does not create an inconsistency. Next interactively change the broadcast mode to non-atomic broadcast in one client. Finally, use two non-commuting operations to show inconsistencies cause by the changed broadcast semantics.

Timings

As in the previous assignment, allow:

- input commands in a sending simulation to be provided programmatically, from the main thread.
- pure local execution of commands.

Also, as in the previous assignment, time the execution of 500 input commands at the site issuing these commands under the following conditions:

- Local execution: the simulation executes the commands locally without sending any messages to remote sites.
- Basic Distributed 3-User: Same as above except that commands are broadcast using non-atomic broadcast to two other simulations.
- Atomic Distributed 3-User: Same as above except that the broadcast is atomic.

Compare these times with those you found in assignment 1, but you do not have to submit this comparison. At some point, we will discuss the times in class.

Your video should, however, show the results of the three performance experiments.

Key Issues and Components of Basic Implementation

Read this part only after you have thought thoroughly about your design.

Your server, as in the previous assignment, allows clients to announce themselves, and relays data from clients to other clients. The clients, as in the previous assignments, announce themselves to the server, sends data to and receive data from the server. The main additional functionality is broadcasting of the broadcast mode.

RMI is supposed to be higher level than NIO, so programming this should, in theory be, easier. You do not have to create a selector thread, change ops, create, accept and connect to socket channels, or read and write bytes. Data are communicated to remote processes by simply making calls in the remote processes – instead of having a matching write and read.

However, because RMI is higher level, you need to take certain extra design and implementation steps. You must design the objects whose methods are invoked remotely. Moreover, from the point of view of this assignment, certain useful parts of the underlying byte communication layer are not automatically exposed. In particular, the server does not have a duplex communication channel to echo received data. You must therefore implement and register object proxies in both the server and client to communicate data. Moreover, the server does not know which client sent a message. This information is necessary to ensure that in local execution mode, the message is not echoed back. So this information must be explicitly communicated by the clients through remote calls. You need to also define objects at both the client and server that support remote calls to change the broadcast mode. The mode is changed and accessed by different threads, so think about races and possible deadlocks that can arise to avoid these races.

There are several valid approaches to address these problems. Again, think of the one you would like to employ before reading the remainder of this section.
Let the server export and register an object that provides a join method called by a client to associate a name with a callback client exported object. Ideally, a client should be able to export the command processor object (provided by the non-distributed simulation) directly to the server, which can remotely set the next command it it. However, RMI requires classes of exported objects to be distribution-aware – they must implement the Remote interface, Thus, in your client, you will need to create and export a distribution-aware proxy for the command processor object in Beau's non-distributed simulation, The server will use an RMI-generated proxy to call a method in this programmer-defined

proxy to call a method in the non-distributed simulation. The server will also allow clients to call a method that broadcasts input commands. Such a method will take as arguments both the string to be broadcast and the identity of the client to avoid an echo in the local mode.

The classes maintaining the broadcast mode in the server and client can be subclasses of a common class that add different side effects to the method to set the mode. Race conditions can occur in a client between a thread that reads the broadcast mode and a thread that writes to it. So these two methods should be synchronized. However, make sure that the method that sends the broadcast mode to the server is not a synchronized method– otherwise synchronous RMI calls can result in a deadlock (depending on which thread does the broadcast in the server).

If classes of remote objects do not inherit from UnicastRemoteObject, to export them, you must call UnicastRemoteObject.exportObject() as is done in the example mentioned above.