Comp 533 - Assignment 3: GIPC Asynchronous Remote Procedure Calls and Synchronized Distributed Consensus

Date Assigned: Thu Feb 16, 2017

Completion Date: Thu Mar 2, 2017, 11:55pm

In this assignment, you will use another RPC mechanism, GIPC, which demonstrates that remote procedure call can be asynchronous, and thus have the efficiency of NIO. Also, you will add an implementation of broadcast-mode broadcast that achieves "distributed consensus" – a general problem in distributed computing. This problem becomes difficult in the presence of faults. For now we will ignore faults. In addition, you will integrate the previous two assignments into this one, allowing the user of a client to choose any of the three IPC mechanisms to communicate information. The change to the IPC parameter will also be made both with and without distributed consensus. You will use thread coordination mechanisms to allow threads to wait for consensus to be achieved. The performance experiments should be performed on multiple virtual machines in the Cyverse cloud. This means, you must change server hostnames referenced in the client programs. Using the cloud will give you experience forking processes in the cloud and experience with setting up a real server – the VNC server. Get the latest version of Beau's project, ObjectEditor, and GIPC, as these have all changed since the last assignment was given.

This is a long assignment description and hence may be daunting if you do all of it in one pass. If you incrementally implement each part, it will be much easier, if not trivial.

RMI vs. GIPC

GIPC (for Generalized Inter-Process Communication) is different from RMI in many ways. The differences relevant to this assignment are that (a) the server and registry are integrated in one process, (b) proxies are created in the client process,
(c) each client must have a globally unique string name (which can be hostname concatenated with some host-relative name), and (d) calls to procedures – methods that do not return values – do not block. As in RMI, calls to functions are synchronous and block until the value is returned.

I have tried to create an API that is analogous to the one offered by RMI. The following table shows the similarities and differences (in bold).

RMI	GIPC
java.rmi.registry.LocateRegistry	inputport.rpc.GIPCLocateRegistry
java.rmi.registry.Registry	inputport.rpc.GIPCRegistry
java.rmi.server.UnicastRemoteObject	n/a
Registry LocateRegistry.createRegistry(int)	GIPCRegistry GIPCLocateRegistry.createRegistry(int)
Registry.rebind(String, Object)	GIPCRegistry.rebind(String, Object)
UnicastRemoteObject.exportObject(String, int)	n/a
LocateRegistry.getRegistry(String, int)	GIPCLocateRegistry.getRegistry(String, int, **String**)
Object Registry.lookup(String)	Object Registry.lookup(**Class**, String)
n/a	**GIPCRegistry.getInputPort().addConnectionListener(ConnectionListener)**
checked Java.rmi.RemoteException	**unchecked** inputport.rpc.duplex.GIPCRemoteException

The GIPC getRegistry() method takes not only the name of the host and port of the server (the first two arguments) but also the unique client name as a third argument. As proxies are created in the client, the lookup method takes an extra argument specifying the interface to be used to create the proxy. The last row is a facility to add a listener for connection and disconnection events. Ignore the warning message you get about using interfaces to create proxies, if you have warnings enabled.

Pull the latest version of GIPC and look at package, examples.gipc.counter.simple, for an example of a server whose methods are invoked by two different clients. The server is ASimpleGIPCRegistryAndCounterServer, and the clients are ASimpleGIPCCounterClient0 and ASimpleGIPCCounterClient1. The code in this package mirrors and, in fact uses, the RMI example, mentioned in the previous assignment, in package examples.rmi.counter.simple. A connection listener (in another package) listens for connection and disconnection events. An unchecked GIPCRemoteException is thrown when a (synchronous) call to a remote function fails.

Basic Functionality

Extend the functionality of the previous assignment. Copy the proxy creation, lookup, and registration code from the previous assignment, change the RMI calls to corresponding GIPC calls, and verify that the original functionality is still present. Do not spend more than a few minutes on this; post a Piazza message or talk to me if you face problems. I have made many changes to GIPC recently, which can cause problems.

Distributed Consensus of Broadcast Mode

Create *another* implementation of the broadcast-mode broadcast that ensures distributed consensus. This means that after the broadcast mode is changed in a client, no commands

input by the user of the client are processed until the client knows that all other clients have also changed the mode, that is, set a variable in the client that holds the mode. (What happens to these user commands is up to you. You can buffer them, at the client or server, or simply ignore them. You also do not have to worry about giving the user any feedback about these commands, though you are free to do so.)

Thus, in this implementation, each client goes into a WaitingForBroadcastModeConsenus state between the time it request a change to the broadcast mode in the server and knows that each other client also changed the mode. You must display, through a GUI or the console, when the client transitions in and out of this state. You should choose a mix of synchronous and asynchronous remote calls to efficiently achieve consensus. To make remote calls to a procedure synchronous, have it return a dummy return value.

The simulation command processor defines the registration method:
public void addPropertyChangeVetoer(PropertyChangeVetoer newVetoer)
to allow registered objects to veto changes to the command property, but you do not need it, as the "connected to simulation" feature offers the same capability.

You should (efficiently) handle the situation in which two clients simultaneously request changing of the the broadcast mode. Ignoring one of the changes at the client and/or the server is a perfectly reasonable solution.

Define an interactive parameter (one that can be changed through the user-interface) that determines of the broadcast-mode broadcast involves consensus or not.

To make the difference between the two modes clearers, this is what happens in the non-consensus mode:

1. A client receives a request from the user to change the state.
2. The client makes a call in the server to broadcast the state.
3. The server makes a callback in each client, including the sending client, to change the broadcast mode.
4. The callback changes the local mode.

In the consensus mode:

1. A client receives a request from the user to change the state.
2. The client makes a call in the server to broadcast the state, changes the local broadcast mode, and sets the waiting state, without waiting for the state to be reset.
3. The server makes a callback in each of the other clients to broadcast the new state.
4. Each of the other clients changes the broadcast mode, and goes into a waiting state.
5. The server makes a callback in each of the clients to announce that each client has changed its state.
6. The callback resets the waiting state.

During the waiting state, user commands are handled in a special – e.g. ignored or buffered.

Multiple IPC Mechanisms

Incorporate the solution to A1 and A2 into this assignment.

This means, your server must listen to messages arriving over the network in three different message queues: the NIO socket-channel created by your NIO server of A1, the socket created by RMI in A2, and the NIO socket channel created by GIPC. You must ensure that the TCP/IP port chosen for these message queues are different. (The second argument of exportObject() in RMI lets you choose the port, in case you want to change the RMI port. The A1 code you wrote and the A3 code you wrote to implement the basic functionality above required you to choose a port.)

The first step in this process is to make sure that each command goes through each mechanism to the server and each of the other clients. This is of course not what we want. Therefore, add to your client, an interactive parameter, changed through the user-interface, that specifies the current IPC mechanism. Based on its value, only one of the three IPC mechanisms is used to send information to the server.

Distributed Consensus of IPC

Allow the IPC-mechanism to be broadcast to all other sites. As in the case of the broadcast-mode broadcast, allow the atomic broadcast of this value to be done with and without consensus. Again, an interactive parameter should determine which option is taken.

Thus, in the consensus broadcast, you need to support a WaitingForIPCMechanismConsenus state, not process messages sent while a client is in this state, and resume regular processing on transitioning out of this state. Again, you should (efficiently) handle the situation in which two clients simultaneously change the IPC mechanism. As the IPC mechanism involves three alternatives (rather than two in the broadcast mode), concurrent changes to it are a bit trickier. Again, ignoring a concurrent change, in the client or server, is a good solution.

Consensus Synchronization Objects

Our algorithms have different threads changing the local state and the waiting states. For example, the main thread in a client may request change to the consensus state and set the waiting state, while an RPC thread handling server calls will reset the waiting state. Some threads do not need to wait for the waiting state to be reset. For example, an AWT thread can simply ignore commands when the client is in the waiting state. However, some threads may need to wait for consensus. For example, the main thread in your performance experiment should wait for consensus after changing the broadcast mode or IPC mechanism.

As the consensus state and the associated waiting states are shared among multiple threads, they should be in monitors. Each monitor should provide write methods for changing the state it holds and non blocking read methods for accessing this state. For example, the monitor for broadcast mode should provide a read method that returns the current local value of the state and another for getting the boolean waiting state. In addition, it should define a read method that blocks until the local state becomes the consensus state. Thus, if the waiting state is false, it immediately returns the local state. If it is true, it waits for the waiting state to be reset.

Since multiple classes in a client may need to access the monitor objects in that client, store references to them in non-public variables of some well known class(es), and define public static read methods to return values of these variables.

Use the blocking read method feature in the performance experiments. Put print statements before after the wait() calls (used to wait for consensus) to demonstrate blocking and unblocking of threads waiting for consensus.

Synchronized Broadcast

Your implementation probably has two different methods in the server for broadcasting simulation commands – one method shared by GIPC and RMI calls, and one method for NIO. This can lead to race conditions. To get more practice with thread coordination, create a global lock monitor object also available through a static method of some well-known class, that is used by the two broadcasts to ensure that they are not executed concurrently by two different threads. Define an interactive parameter to determine if the two broadcasts are synchronized.

Submission Instructions

As in the previous assignment, create a YouTube video demonstrating the working of your program and following the constraints discussed below. Use the Google form to submit the video link. Submit your code to Sakai (the assignment will be created before the due date).

GIPC Demo with Breakpoints

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

- The inputting simulation makes a remote method call in the server.
- The simulation returns before the server makes the call.
- The server executes the call.
- The server making a remote-method call in a non-inputting client.
- The server returns before the call completes,
- A non-inputting client executes the method.

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

Motivation for Broadcast Mode Consensus

Again, make the GIPC as the communication mechanism. Do not broadcast the broadcast-mode via consensus. Start with the broadcast-mode being atomic. Create a server with a single client.

- Put a breakpoint in the method in the server that broadcasts the broadcast mode.
- Change the broadcast-mode in the client to local broadcast.
- While the server is stopped, enter a user command.
- Resume the server.
- Identify the problem.

Repeat the experiment, but this time, use consensus to broadcast the broadcast-mode. Show that the problem no longer occurs.

Broadcast Mode Consensus with Multiple Clients

Ensure that the broadcast-mode is broadcast using consensus. Show the following sequence of steps:

- A client changes the broadcast mode from atomic to local and breaks before making the server call to announce the change.
- Another client does the same.
- The first client resumes execution.
- The second client resumes execution.
- Both clients output the fact that they are waiting for consensus.
- Each of the clients in the simulation breaks in the call that receives the consensus information,
- One or more commands are input by users of at least two clients, which are ignored.
- Each client resumes execution and prints the state change.
- One or more commands are input by users of at least two clients, which are executed (along with possibly ignored commands

Motivation for Synchronized Broadcast

Create two clients. Make the command-broadcast in the server not synchronized. Make the command-broadcast atomic at all sites. Make NIO the communication mechanism in one client and GIPC the mechanism in the other client. Enable NIO tracing in all processes.

- In the server, put breakpoints in the two methods that do broadcast.
- A client sends command, a, using NIO.
- Another client sends a non-commuting command, b, using GIPC.
- An NIO and a GIPC thread are now stopped in the server.
- Let the NIO server thread send command, a, to one of the clients but not the other.
- Let the GIPC server thread send command, b, to both clients.
- Resume the NIO thread to finish the broadcast.
- Show the problem.

Now make the input-command broadcast in the server synchronized, and repeat the steps above to show that the problem does not occur.

Performance Experiments

Using VNC for the clients, *for each IPC mechanism*, you should time the execution of 500 input commands at one of the Cyverse computers, 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.

Your experiments should involve no interactive input and involve one run. This means the main thread of the client should perform all the actions and use the consensus synchronization objects to wait for the changes to the broadcast mode and IPC mechanisms.

Repeat the experiments above, but this time execute the server and client on a different Cyverse computers.

Repeat the experiments on different computers, do not use VNC for the clients. Instead, run the clients "headless" directly using ssh directly.

Try to understand the difference in times.

Key Issues and Components of Basic Implementation

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

To achieve distributed consensus, a client must inform the server about the change to the some state; the server must inform other clients about the new state and receive information from them they have made the change locally, and then inform all clients that distributed consensus has been achieved.

You can do this all using only asynchronous or only synchronous procedures, but it will be most efficient to make only the calls informing other clients about the new broadcast mode as synchronous.

Both the client and server can ignore attempts to change state while they are waiting for consensus on a previous change to it. Catching it at the client is more efficient but the server has more information, in particular, it can discover and order changes to the state made concurrently by the two clients. Implementation of consensus synchronization objects and synchronized broadcast will involve use of the Java wait() and notify(0 methods. In one case they achieve synchronization and in the other they support mutual exclusion of code in different classes. When they achieve synchronization, be sure to take

into account the fact that notify is a hint rather than an absolute, and requires a different way of programming in which the wait is put in a while <not ok to proceed> loop.