Comp 734 - Assignment 5: Synchronous Receive and Remote Method Call

Date Assigned: Nov 7, 2013

Part 1 Submission Date: Nov 14, 2011

Part 1 and 2 Completion Date: Thu Nov 21, 2011

In the last assignments, you learnt how to communicate objects on GIPC ports. In this assignment you will add synchronous receive and use it to implement synchronous function and procedure calls.

Part 1: Explicit and Synchronous Receive

Most of the existing GIPC code you will use for this part is in the package: `inputport.datacomm.duplex.object`.

Create extensions of the client and server duplex object ports that provide an explicit synchronous blocking receive. It takes as an argument the name of the sender, and waits until a message from that sender arrives. Its return value should be such that the receiver can retrieve the sent object from it. Thus, the return value can either directly be the object sent or a special kind of GIPC message in which the sent object is a property.

It should be possible for multiple threads to concurrently perform receive operations. When a message arrives, it should go to (a) the first synchronous receive waiting for it, and (b) all listeners registered to receive the message.

Provide also an implicit synchronous receive, which calls the explicit receive using the sender name returned by `getSender()`. In the case of a client port, this method always returns the server.

This part essentially involves using monitors, bounded buffers (implementations of Java `BlockingQueue` such as `ArrayBlockingQueue`), and GIPC Duplex Object ports to implement your new abstraction.

The interfaces of your new server and client ports will extend DuplexClientInputPort and `DuplexServerInputPort` interfaces, respectively. The implementations of the two ports should be layered on top of the existing implementations of the duplex server and client object ports, respectively. Your new implementations can either inherit from or delegate to the existing implementations. Inheritance will be easier, delegation more flexible.

If you use inheritance, you need to know that the standard GIPC implementations of `DuplexClientInputPort<Object>` and `DuplexServerInputPort<Object>` are `ADuplexObjectClientInputPort` and `ADuplexObjectServerInputPort`, respectively. If you are using delegation, you can be more flexible and simply use the values returned by `DuplexObjectInputPortSelector.createDuplexServerInputPort()` and `DuplexObjectInputPortSelector.createDuplexClientInputPort()`.

If you use inheritance, you will have to use a separate class to be a receive listener of the object port. If you use delegation, use the Eclipse facility to generate delegate methods, which is available in the menu provided to generate getters and setters.

Define a factory class to instantiate your two ports. It should implement the interface: `DuplexInputPortFactory`. It ~~-~~can be modeled after (and extend) the existing GIPC factory: `ADuplexObjectInputPortFactory`.

As in the previous assignment, override

```
protected void initPortLaucherSupports()
```

in your testing launchers to trace and use your code for both the serializer and duplex object ports.

Submission Part 1

Explain how you handle blocking and unblocking of a synchronous receive. In particular, explain how your code handles a received message. Show test cases with traces illustrating your algorithm.

Part 2.1: Synchronous Remote Function Call

Most of the existing GIPC code you will use for parts 2.1 and 2.2 is in the package: `inputport.rpc.duplex`

Create a new implementation of a duplex RPC input port that provides an alternative implementation of the remote function call object. You do not have to create a new class for the RPC port. Instead, you have to define only a new factory class for such a port that uses your implementation of the duplex object port of part 1, and the remote call completer (executing at the local site) described below. The interface of your new factory is: `DuplexRPCInputPortFactory`. Your new factory class can be modeled after (and inherit from) the existing implementation of this interface: `ADuplexRPCInputPortFactory`.

~~The interface DuplexSentCallCompleter~~ The current implementation of a remote function call, `ADuplexSentCallCompleter` , uses monitors to wait for the returned value. It implements the interface `DuplexSentCallCompleter`. Provide an alternative implementation of the interface to use sync receive calls to do the waiting. You will have to define a factory to instantiate your implementation of this interface. The factory for this object is selected by the abstract factory, `DuplexSentCallCompleterSelector`. Your new factory can be modeled after (and extend) `ADuplexSentCallCompleterFactory`.

You can simply extend the GIPC duplex sent call completer class. If you do so, your responsibility is to override the constructor and one or more of the methods `isReturnValue`(), `processReturnValue`() and `returnValueOfRemoteFunctionCall`(). As its name indicates, `returnValueOfRemoteFunctionCall` is called by GIPC to make the caller block until the return value is received. This means that, in general, a sent call completer must have a chance to process messages received from the remote site. The method `processReturnValue`() must be overridden to do this extra processing, which can involve doing nothing. The method `isReturnValue`() determines if a message is a return value. The constructor of this class takes as an argument a `DuplexRPCInputPort`. The method `getDuplexInputPort`() of this port can be used for returning your implementation of the duplex object port of part 1.

As in the previous part, override

```
protected void initPortLaucherSupports()
```

in your launchers to trace and use your code for this and the previous parts.

Part 2: Synchronous Procedure Call

Again, you will create a new implementation of the duplex RPC input port by defining a new factory that creates a new composition of objects. Currently, a remote function call is synchronous but a remote procedure call is asynchronous. Your new implementation of a duplex RPC input port will support synchronous procedure (and function) calls.

This time you will provide an alternative implementation of both the sent call completer (executing at the local site) and the received call invoker (executing at the remote site).

As mentioned above, you will have to provide yet another implementation of `DuplexSentCallCompleter`,~~-~~ which can be an extension of the one you created above. This time, you will have to override~~,~~ the method `returnValueOfRemoteProcedureCall()` which is called by the local site to wait for the procedure call to complete.

To support synchronous procedure calls you will have to trap the invocation of the call at the remote site. You can do so by providing an alternative implementation of the interface `ReceivedCallInvoker`, ~~-~~selected by the abstract factory , `SynchronousDuplexReceivedCallInvokerSelector`. You can simply extend the ~~existing classes~~existing classes `ADuplexReceivedCallInvoker` and `ADuplexReceivedCallInvokerFactory`

You will have to override the method `handleProcedureReturn`() by taking an appropriate action. The method `handleFunctionReturn`() (which you do not have to modify) of `ADuplexReceivedCallInvoker` shows how function returns are handled.

As always, you will override

```
protected void initPortLaucherSupports()
```

in your launchers to use and test your GIPC-based code.

Execution Behavior and Performance

Use your launchers to implement the 4-user case of the previous assignment in which all of your GIPC implementations are exercised. Compare these timings with those obtained using the predefined GIPC implementations.

In your launchers, add the following code to `initPortLaucherSupports()`:

```
  DuplexReceivedCallInvokerSelector.
        setReceivedCallInvokerFactory(new ADuplexReceivedCallInvokerFactory());
```

The default factory assigned to this selector is an instance of:
`AnAsynchronousDuplexReceivedCallInvokerFactory`.

Compare the code of the two factories and explain the impact of making this change. Now observe the behavior of the 4-user case when you make this change and explain why you or do not see change.

Submission Part 2.1 and 2.2

1. In the printout you submit:
    a. Include what you submitted for part 1.
    b. Explain how your implementations of the two subparts work.
    c. Explain if your implementations rely on a received message going to both a synchronous receive and all listeners for the message?
    d. Provide the performance comparison mentioned above.
    e. Discuss the impact of replacing `AnAsynchronousDuplexReceivedCallInvokerFactory` with `ADuplexReceivedCallInvokerFactory`.
    f. Give a trace of your code of part 2 showing the major steps in it.
2. Create a Youtube video demonstrating the 4-user GIPC synchronous case of the previous assignment in which all of your GIPC implementations of part 2 are exercised. In this video, convince me through the blocking tracer and/or the debugger that the procedure calls are synchronous.

3. Submit your code on Sakai.