Comp 533 - Assignment 4: Implementing Explicit Receive and Synchronous Remote Method Invocation

Date Assigned: Mar 20, 2017

Completion Date: Tue April 4, 2017

In this assignment, you will gain get your first experience with implementing (rather than using) interprocess communication. You will extend GIPC. In the first part, you will implement an explicit synchronous receive call, which is missing in GIPC. In the second part, you will use this call to replace the GIPC implementation of synchronous remote function calls with a new implementation. This part will show you that IPC serves as an alternative to monitors. In the third part, you will use the receive() call to replace GIPC asynchronous remote procedure calls with synchronous remote procedure calls. The code in the package `examples.gipc.counter.customization` illustrates how to do the customization required by this assignment. This package, in turn, is built on top of the code in `examples.gipc.counter.layers.` The latter shows three ways of communicating with a server on the same GIPC port, using bytes, serialized objects, and RMI, respectively. Study the code in the two packages carefully before you implement this assignment. In particular, run the client and server in each package and understand what the causes of the output. The custom serializer is not used in this assignment.

Part 1: Explicit and Synchronous Receive

Most of the existing GIPC classes you will extend in this part are in the package: `inputport.datacomm.duplex.object`. Some of the relevant interfaces are in the package `inputport.datacomm.duplex.object.explicitreceive.`

The example package illustrates how to replace and use the client and server duplex object ports (which are responsible for communicating objects over the underlying byte channel) with your own implementations of these ports. For this part, replace the stub implementation of the following method in these ports with a real implementation:

```
public ReceiveReturnMessage<Object> receive(String aSource) {
      System.err.println("Receive not implemented");
      return null;
}
```

Your implementation should block the thread making this call until a message is received from the sender, `aSource`. The source of the message, together with the contents, is returned as an instance of the predefined interface `ReceiveReturnMessage.` An implementation of this interface is provided by the class `AReceiveReturnMessage. (`In Eclipse, Alt backspace completes and finds the package of a class or interface for you.)

It should be possible for multiple threads to concurrently perform receive operations. When a message arrives, it should go to (a) the thread that executed the first synchronous receive waiting for it, and (b) all listeners registered to receive the message. This means you must use a bounded buffer. Trace actions on this queue by calling the newCase() method of the classes `ReceivedMessageQueueCreated`, `ReceivedMessageQueued`, and `ReceivedMessageDequeued` when the corresponding actions occur on this queue. The first argument of the newCase() method is always the object calling it: **this**.

As in the customization example:

(a) The implementations of the two ports should be layered on top of the existing implementations of the duplex client and server object ports: `ADuplexObjectClientInputPort` and `ADuplexObjectServerInputPort`, respectively.
(b) 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`.
(c) Register these factories with `DuplexObjectInputPortSelector.`
(d) Create a custom notifier to process incoming messages.

The customization example also shows that you can test your implementation of the reply() call by having the main thread of both the server and client execute a loop that receives each message sent to the client or server. This code uses a predefined parameterless `receive()` call.

By default, the parameterless `receive()` call receives from the last sender. You should override the default implementation of parameterless `receive()` call in different ways for the client and server. The client has only one receiver, the server, so it should receive from this server, retrieving messages from the queue created for it. The semantics are trickier for the server. If there is a last sender (that is this sender is not null), the parameterless receive() call should block waiting for messages from the last sender. If there is no last sender, it should wait for a message from any sender. When a new message arrives from client C, it has the following behavior:

1. If `receive(C)` has been executes by the server, then it is consumed by that `receive();`
2. Otherwise it is consumed by the parameterless `receive().`

To create a more flexible system, allow the parameterized `receive()` to take the argument "*" to indicate receipt from any sender.

Part 2.1: Alternative Synchronous Remote Function Call

Most of the existing GIPC code you will use and extend 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 "sent call completer" object using the receive call of your custom ports implementations.

The current implementation of the "sent call completer", `ADuplexSentCallCompleter` , directly uses bounded buffers to wait for the returned value. It implements the interface `DuplexSentCallCompleter`. Provide an alternative implementation using explicit 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`.

Your implementation of the call completer can extend `ADuplexSentCallCompleter.` You will need to override in `ADuplexSentCallCompleter` the constructor and one or more of the methods `waitForReturnValue ()` and `returnValueReceived()`. The method `waitForReturnValue()` is called by GIPC to make the caller of a remote function block until the return value is received. As mentioned above you will use receive(0 to block. The method `returnValueReceived ()` receives incoming return values and does any special processing required by the call completer. In the default implementation, it puts return values in a bounded buffer.

Part 2.2: Synchronous Procedure Call

Currently, a remote function call is synchronous but a remote procedure call is asynchronous. You will now 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).

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 , `DuplexReceivedCallInvokerSelector`. You can simply extend the existing classes `ADuplexReceivedCallInvoker` and `ADuplexReceivedCallInvokerFactory,` as the example customization package illustrates.

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. Procedure returns can be handled similarly to ensure synchronous calls

Your new implementation of `DuplexSentCallCompleter` can be an extension of the one you created above. This time, you will have to override the method

`getReturnValueOfRemoteProcedureCall ()`, which is called by the local site to wait (or not) for the procedure call to complete. This method can be modelled after the super class method `getReturnValueOfRemoteFunctionCall()`.

In the atomic mode, your implementation will deadlock if you directly use your custom call invoker. Look at the sample code on how to create an asynchronous call invoker. You need to create a factory class that is a subclass of your custom factory class and passes your call invoker to a special GIPC provides asynchronous call invoker. The selector (abstract factory) for the call invoker should now be set to an instance of this class. This ensures that method calls are not invoked directly by the selector thread (thereby blocking it on callbacks, which are now synchronous through part 2), but instead by a single special thread created for these invocations.

Implementation Hints

Part 1: This part essentially involves using bounded buffers (implementations of Java `BlockingQueue` such as `ArrayBlockingQueue`) to implement yet another producer consumer problem. The consumers are the threads that execute the `receive()` calls. The producer is the thread that sends message notifications to receive listeners. In addition to sending the notifications, it will now add receives messages to the appropriate message bounded-buffers. The client is connected to a single server, so a single bounded buffer is required. The server is connected to multiple clients, so a separate bounded buffer per message source is needed. In addition, a bounded buffer for all clients is needed. When a new message arrives at a server from a client, it goes to the bounded buffer for the client if some thread is waiting for a message on the buffer; otherwise it goes to the general bounded buffer.

The code in the sample package shows that one can intercept/override calls in a client or server port by replacing these ports with custom ports using a factory class. It also shows that one can replace the standard notifier of incoming messages with a custom notifier using a factory method. The intercepted/overridden receive() calls in the two ports will execute the consumer behavior, while the intercepted/overridden `notifyPortReceive()` in the custom notifier will execute the producer behavior. Make sure the notifier calls the overridden method to sends messages to all listeners.

The port and notifier need to share a data structure. As the port creates the notifier through a factory method, this can be achieved by simply passing the data structure to the constructor of the notifier.

An alternative to overriding the notifier is to add a receive listener to every port that is instantiated. One way to do this is to make the factory that creates a port add a listener, which in turn means retrieving the shared data structure from the port and passing it to the listener. Thus, the factory class is doing more than instantiating the port – not a good idea from the point of view of separation of concerns. Another approach is to make the port, in its constructor, add a special first listener. This is cleaner and maybe more attractive than the notifier approach, which however, is more general, as it allows processing of messages sent to each listener.

The server receive semantics require a way to determine if some thread is waiting on a blocking queue. Java's implementations of blocking queues do not provide such a way. You can create a wrapper subclass that overrides the take() method to provide this information, using a global counter to do so. The overridden method can increment the counter, call the superclass take(), and decrement the counter. Now a method can be provided to determine if the counter is zero or not. (Will a Boolean work instead of a counter?)

Part2.1: Because you are building on top of Part 1, this part becomes easier to implement. The tricky part is making your new implementation fit into the GIPC framework. The GIPC default implementation of function calls uses a bounded buffer, which in, turn, uses monitors for synchronization. Your synchronous explicit receive() provides a much more convenient way to do this synchronization. The waiting for the return value is done in `waitForReturnValue ()` by repeatedly calling the synchronous receive() until an

RPC return value is received. You have to modify other methods in your call completer to make them consistent with these semantics. As `waitForReturnValue ()` no longer retrieves a value from a queue, `returnValueReceived()` should no longer put anything in the queue. Be sure that `waitForReturnValue ()` returns the ReturnValue property of the Message property of the MessageWithSource returned by receive().

Part 2.2: In the server, `handleProcedureReturn()` is now like `handleFunctionReturn()`except it sends a null return value. It can call the same method as the latter does. (In the client, get`ReturnValueOfRemoteProcedureCall(),` is like `get`ReturnValueOfRemoteFunctionCall(), except that the returned value is guaranteed to not be a remote argument so it does not have to translated to create a proxy object.)

Submission

Submit videos for the following demos:

Part 1, Multi-Layer Counter Example: Modify the example in the customization package by replacing the custom ports and notifier with your Part 1 ports and notifier. Execute port.trace.objects.ObjectTraceUtility.setTracing() to turn on tracing of your three trace calls and GIPC ones. Set a break point in the main loops of the client and server. Run the modified client and server and use the breakpoints and output of the server and client console windows to show that your implementation currently implements the required semantics.

Part 2, Multi-Layer Counter Example : Modify the example of Part 1 to now use the received call invoker and sent call completers of parts 2.1 and 2.2. Execute `port.trace.objects. ObjectTraceUtility.setTracing()` and `port.trace.rpc.RPCTraceUtility.setTracing()` to turn on both tracing of object and RPC operations. Set a breakpoint in the `increment()` and `getValue()` methods in the server. Run the modified client and server and use the breakpoint and the output of the server and client console windows to show that your implementation currently implements the required semantics for 2.1 and 2.2 – it uses the receive() call to synchronize both the function and procedure calls in the server. The tracing should should show your receive calls being executed at the right times.

Part 2, Simulation: Run the 3-client 500-command, simulation experiments on your local computer using both the default GIPC implementation and your new implementation of parts 1, 2.1 and 2.2 and compare the timings.

As usual, also submit the code on Sakai.