Design/Code Walkthrough

My first thought on interprocess communication I rejected immediately: REST web services. There is no convenient callback mechanism with REST. The web just isn't set up for it, and the purpose of REST is to separate the client and server with all kinds of proxying infrastructure to enable massive scaling. This isn't compatible with a callback architecture. Thus, the client would have to poll for shuffled decks, and this was obviously inappropriate for the specification.

My second thought on interprocess communication was to use raw sockets. I rejected this idea, and regret it. I thought it would be too "roll your own". But this application is perfect for it. Clients ping the server with only two commands with no arguments: shuffle, and get the current deck. The server always sends the same message to the clients: an array of 52 numbers. Oh how I wish I had just gone ahead and done that.

I chose RMI instead. I've never worked with RMI and thought it would be a good chance to try it. Unlike other interprocess schemes, like SOAP, it doesn't require any additional container or library software. It's been built into Java from the start. But man oh man … running it in that mode, standalone, is a giant PITA. (See the section, Issues, below.)

Interprocess communication is through RMI which proxies objects between processes. The objects to be proxied must be registered with the RMI object broker – this includes objects used for callbacks. The basic process is to define interfaces which extend java.rmi.Remote, and then to implement those interfaces appropriately.

Project card contains these interfaces, and common code. Project table contains the server. Projects PlayerCLI and Player have the CLI and GUI clients, respectively.

Project card

The client—the process that is going to receive shuffled card decks—implements interface Player, which allows the server to call back with a deck of cards.

It connects to the service "Table" and gets a Table proxy. The interface Table allows the client to register itself with a call to joinTable, passing its own Player to the service, along with a string name (for diagnostic purposes only). On joining the table it gets back a Dealer proxy. It now communicates with the service only through the Dealer (it no longer needs the Table).

The Dealer interface allows the client to request a shuffle, or to request the current deck of cards. It also has a method to disconnect from the server gracefully.

The reason for splitting the server interface into two pieces, Table and Dealer, is to allow the service to have a separate object talking to each client. This allows the service to keep track of client lifetime which makes the callbacks to the client Player proxies work better.

The deck of cards is transmitted asynchronously (from shuffle requests and getDeck requests) to clients. It is passed as a 52 element array of integers, which contains a permutation of 0..51. A helper class, Deck (which is not used in the interprocess communication and thus doesn't extend java.rmi.Remote) , implements the shuffle (in method shuffleADeck) and can return a string representation of a deck (either using Unicode "suit" characters or using an ASCII representation).

Two helper classes in project card, RmiStarter and PolicyFileLocator, are used to help register the server and client objects with the RMI object broker, and set the permissions.

Project table

The class TableImpl implements Table and handles the requests for decks and shuffles. The basic goal is that all clients show the same up-to-date deck at all times (except for a short transient period right after a shuffle is requested when the deck has not yet been transmitted to all clients). I considered it unacceptable to have a client display the wrong deck. Yet clients can attach to the server at any time and need to get the current deck without causing a shuffle. They do this by requesting a deck and getting the deck sent back to them. But the key is: there must only ever be a single deck, and the requests for shuffles and requests for decks must be serialized.

This is accomplished, in TableImpl, by processing all requests in a separate thread that pulls requests from a queue, in this case, a LinkedBlockingQueue. When there are no requests, the thread blocks. As soon as a request comes in it is dequeued and handled. The thread is implemented by the class DeckPusher, which is a Runnable.

The queue is of Object because it the requests are represented by unrelated classes. If a client requests the current deck the thread is going to send it only to that client. This is done by enqueueing the Player proxy. If a client requests a shuffle the deck will be shuffled and sent to all clients. This is done by enqueuing anything *except* a Player proxy.

The TableImpl also keeps track of the current connected clients, represented by their Player proxies, in a synchronizedSet (made of a HashSet). Additions and removals are handled in a thread-safe manner by the synchronizedSet wrapper.

Removals happen whenever

a) a client gracefully disconnects

b) a client terminates the connection and it is eventually discovered by RMI and reported, and

c) when there's any kind of exception thrown when trying to send a deck to a client.

The third option happens in TableImpl, the other two in the DealerImpl.

The DealerImpl class implements the interface Dealer and simply forwards everything to the TableImpl. The key reason to have the DealerImpl is so it can handle graceful termination when the client calls

leaveTable, and ungraceful termination when RMI discovers the client connection is dead and calls the method Unreferenced.unreferenced.

A class PlayerWrapper simply wraps the Player proxy to the client, associating the client's name with its Player proxy. This is simply for diagnostic purposes.

Finally, there is a class TableStarter which contains the service entry point and creates and registers the service.

Project PlayerCLI

The command line client is very simple. The class PlayerImpl implements the Player callback interface, and talks to the server to connect and disconnect. It runs a input loop to send shuffle requests, and to exit. And whenever it receives the deck callback it prints the deck to the console.

The class PlayerStarter contains the client's entry point, and creates the PlayerImpl, registers it as a callback, and sets it running.

Issues

RMI is reasonably easy to design and code against, but is impossible to configure for standalone operation. I'm sure it can be configured reasonably when it is run out of standard Java containers, e.g., JBoss or Tomcat. But for just processes talking to each other it is very very difficult to get things going. I'm pretty sure I don't have it running the 'right' way—but it does run.

One of the major problems is that the rmiregistry tool, the RMI object broker, issues no diagnostics whatsoever. It has no logging mode for failures to provide proxys. Your various RMI calls will fail with ClassNotFoundException for any one of a couple of dozen reasons and there is no way to figure out which reason applies. What a nightmare!