Callbacks

There are two types of callbacks supported within the JBoss Remoting package; pull and push. Pull callbacks (sometimes referred to as synchronous) are made up of a callback listener registering to receive callbacks from the server and then programmatically getting its callbacks from the server. The server will merely hold these callbacks until the client requests them. Push callbacks (sometimes referred to as asynchronous) are made up of a callback listener registering to receive callbacks from the server and the server sending these callbacks to the registered listener upon receipt of a callback.

Regardless of using push or pull callbacks, the server sub-system will be responsible for registering a callback handler and generating the callbacks to be fired to the callback handlers. The actual callback handler to be registered with the server sub-system, the `ServerInvokerCallbackHandler`, will be a proxy representation of the client callback handler.

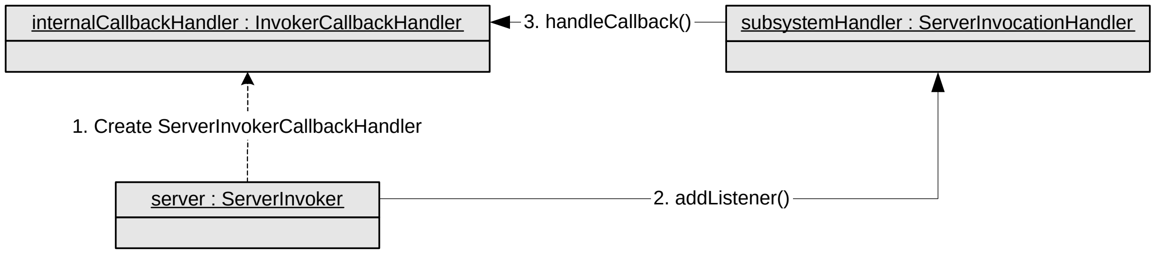
internalCallbackHandler : InvokerCallbackHandler
3. handleCallback()
subsystemHandler : ServerInvocationHandler
1. Create ServerInvokerCallbackHandler
server : ServerInvoker
2. addListener()

Figure 1 - Server Sub-System

This callback handler proxy will then determine if it should hold the callbacks until requested by the client, in the case of pull callbacks, or send the callback to the client, in the case of push callbacks. The server sub-system will not be aware of if the client is using pull or push callbacks.

Pull Callbacks

Pull callbacks within JBoss remoting allow the user to control when callbacks are received by having the client call on the server for its callbacks. The request to get the callbacks from the server is a synchronous, blocking call and will return a collection of the callbacks that exist for the calling client (which can contain zero or more callbacks).

Pull Callback Process

This section will walk through the process of registering and retrieving pull callbacks.

Register for callbacks

The first step is for the user to create an implementation of the `InvokerCallbackHandler`. The only method within the `InvokerCallbackHandler` is the `handleCallback()` method, which does not require any implementation for pull callbacks. However, a callback handler is required so can be registered and unregistered with server so that the server knows when to start and stop collecting callbacks for the client.

Next the user will need to call `addListener()` on its Client object (see general JBoss Remoting documentation for more information on the Client class). There are two `addListener()` methods, one with just `InvokerCallbackHandler` as a parameter and the other with both a `InvokerCallbackHandler` and a `InvokerLocator`. In order to use pull callbacks, the user must call the method with the one parameter. The parameter, `InvokerCallbackHandler`, is just the user's empty shell implementation

Upon calling `addListener()`, the *Client* will then register the callback handler with the target server by calling `addListener()`. At this point, the callback handler is not actually sent to the target server. Instead, the target server will create an internal proxy callback handler, using the session id from the *Client*'s invocation as an identifier. The target server will then register this proxy callback handler with the server's subsystem handler by calling `addListener()`. The client is now setup to retrieve callbacks.

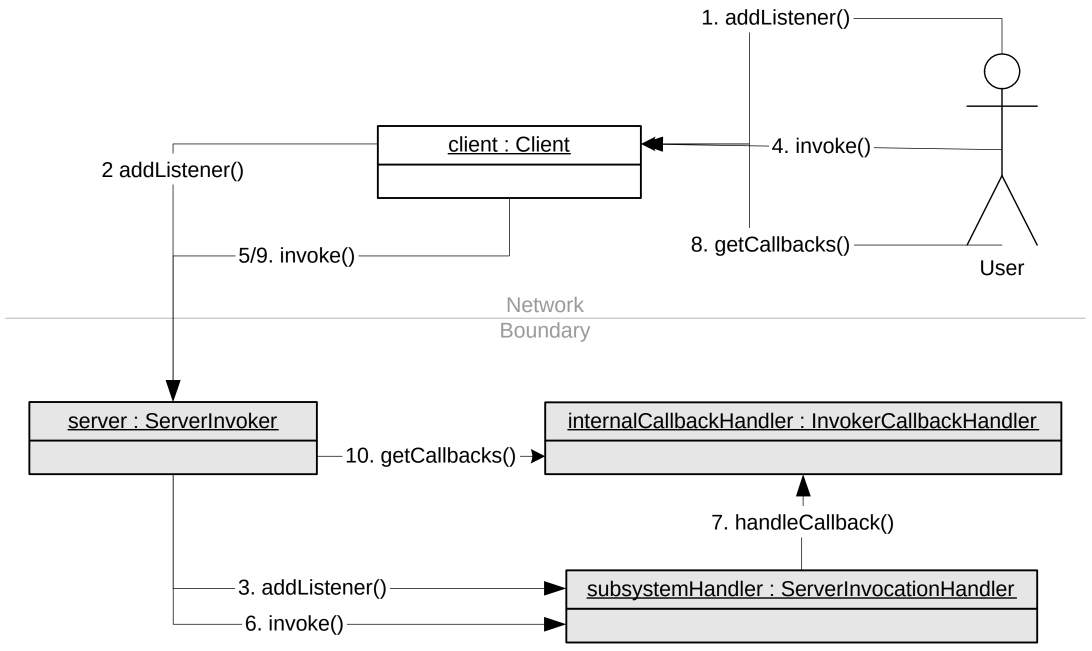

Figure 2 - Pull Callback

Retrieving callbacks

Callbacks are originally initiated by the target subsystem's handler. Often a callback will be initiated in response to an invocation made by the client (as illustrated in Figure 2 - Pull Callback). When this occurs, the subsystem handler will call `handleCallback()` on the proxy callback handler that was previously registered with it. The internal proxy callback handler will then store the callback within its internal collection.

The user can then call on the Client to retrieve its callbacks by calling `getCallbacks()`, which will return a *List* of callback objects. The server's proxy callback handler will continue to store the callbacks for the client until the client is unregistered by calling `removeListener()` on the Client.

Push Callbacks

Push callbacks within JBoss remoting should be used when the user wants receive callbacks from the server as soon as the server receives them from the server's subsystem handler without blocking. Using push callbacks can be more powerful than pull callbacks because it allows to user to specify a different transport protocol to call back on, but can also be more complex to setup.

Push Callback Process

This section will walk through the process of registering and receiving push callbacks, as well as covering some of the more discrete details of push callbacks.

Register for callbacks

The first step is for the user to create an implementation of the `InvokerCallbackHandler` (how to implement this interface will be discussed in detail later). The only method within the `InvokerCallbackHandler` is the `handleCallback()` method, which will receive the callback object.

Next the user will need to call `addListener()` on its Client object (see general JBoss Remoting documentation for more information on the Client class). There are two `addListener()` methods, one with just `InvokerCallbackHandler` as a parameter and the other with both a `InvokerCallbackHandler` and a `InvokerLocator`. In order to use push callbacks, the user must call the method with the two parameters. The first parameter, `InvokerCallbackHandler`, is just the user's implementation on which to receive callbacks. The second, `InvokerLocator`, is the locator to the server the user wishes to initially receive the callbacks from the server, which will be referred to as the callback server. The callback server can be any JBoss Remoting server, using any transport protocol or subsystem as its handler (see general JBoss Remoting documentation for more information about JBoss Remoting servers). However, the most common configuration will be to use a remoting server that is within the same VM as the client and using the same subsystem as client.

Once the user calls the `addListener()` method on its *Client*, the *Client* will first call the callback server to register the provided callback handler to receive callbacks. The *Client* will be unaware of if the callback server is local to the same VM or a remote server (Figure 4 - Push Callback (Remote Callback Server) indicates that the callback server is remote and Figure 3 - Push Callback (Local Callback Server) indicates that the callback server is local). Since in most cases the callback server will be local, this scenario will be covered in this section. The scenario where the callback server is remote will be covered in detail in the `InvokerCallbackHandler` section.

Once the callback handler is registered with the callback server, the *Client* will then register the callback handler with the target server by calling `addListener()`. At this point, the callback handler is not actually sent to the target server. Instead, the target server will create an internal proxy callback handler, using the session id from the *Client*'s invocation as an identifier as well as the locator to the callback server. The target server will then register this proxy callback handler with the server's subsystem handler by calling `addListener()`. The client's callback handler is now setup to receive callbacks.

Receiving callbacks

Callbacks are originally initiated by the target subsystem's handler. Often a callback will be initiated in response to an invocation made by the client (as illustrated in Figure 3 - Push Callback (Local Callback Server)). When this occurs, the subsystem handler will call `handleCallback()` on the proxy callback handler that was previously registered with it. The internal proxy callback handler will then use its *Client* object, which refers to the callback server based on the locator that was originally used to register the client callback handler with the target server, to callback on the callback server. This callback is really just a remoting invocation and uses the exact same mechanism that is used by the user's *Client* to call the target server. Once again, the proxy callback handler's *Client* is unaware of the physical location of the callback server (so could be within local VM or remote).

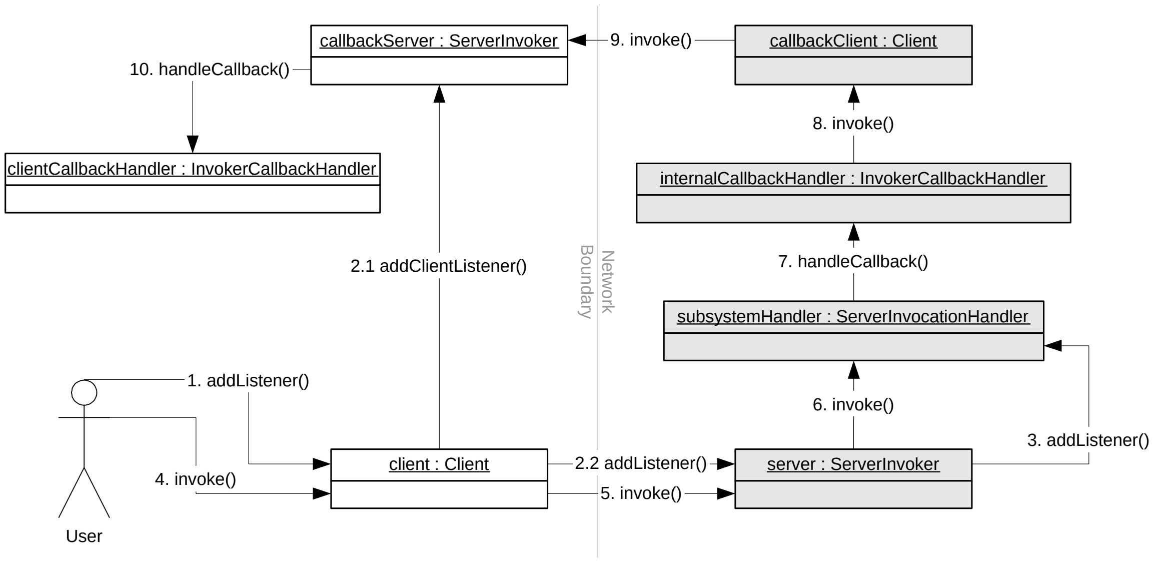

Figure 3 - Push Callback (Local Callback Server)

The callback server will then call `handleCallback()` on the callback handler the user originally registered. It is important to note that sub-system associated with the callback server is not directly involved in the dispatching of callbacks to the client callback handler; this is all handled within the JBoss Remoting layer.

Once the user no longer wishes to receive callbacks, the user will need to remove the callback handler by calling `removeListener()` on the Client.

InvokerCallbackHandler

When implementing the `InvokerCallbackHandler` the biggest consideration will be if the callback server to be used will be local to the Client to be used (within the same VM) or will be a remote callback server. In most cases, using a callback server that is local will be sufficient (and also the most efficient). In this case, all that is needed is to implement the `InvokerCallbackHandler` interface.

There may be some cases when the use of a remote callback server is desired. Although this requires a little more effort, it is possible. The first step in accomplishing this is to create a new interface that extends the `InvokerCallbackHandler` interface as well as the `Serializable` interface and then implement this newly created interface. This is required since the default pull callback implementation, which uses the un-serializable `InvokerCallbackHandler` interface, assumes that no remoting is required to register the client callback handler with the callback server.

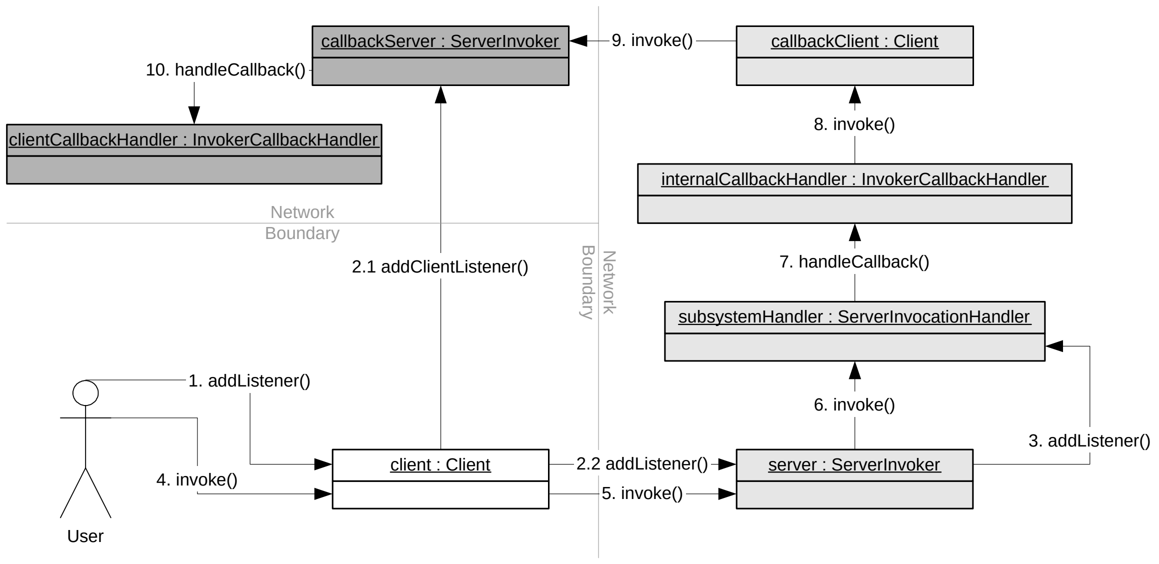

Figure 4 - Push Callback (Remote Callback Server)

The second step is to consider how this new serializable callback handler will process its callbacks. Therefore, it is important to understand exactly what the process for receiving callbacks will be in this remote callback server scenario (see Figure 4 - Push Callback (Remote Callback Server). Since the new callback handler will actually be serialized and sent to the remote callback server (step 2.1), there will be two instances of this new callback handler; one instance within the client VM and another in the remote callback server VM. The two will not have direct knowledge of one another (unless explicitly added to the callback handler implementation). This means that when the `handleCallback()` method is called (step 10 in Figure 4 - Push Callback (Remote Callback Server)), the callback handler will not have any implied connection back to the original client. At this point, it will be completely up to the user's implementation of the callback handler to figure out what further processing is needed once the callback is delivered.