AltRMI - Overview

by Paul Hammant, Vinay Chandran

1. Introduction

AltRMI is a from-scratch replacement for RMI. It has a number of different features that make it easier to use. It tries as far as possible to be transparent in use. It is also inspired by the remoting facility in .NET. This does not mean that it has SOAP capabilty (yet), as it is more like the proprietary RPC transport for the .NET framework.

The name AltRMI is inspired by the 'alt' usenet newsgroup hierarchy. We have thought about renaming it especially as alt means old in German. DRMI (Distributed RMI), PMI (Proxied Method Invocation), ARC (Apache Remote Control) and JRemoting were all considered as replacements, but for now we stick with AltRMI.

The mail list for this project is 'projects at incubator.apache.org'. Subscribe here. Please mark your emails with a subject of [altrmi] + your normal subject choice.

2. Feature Differences

Some good, some bad:

- It transports normal Java interfaces (no need to extend java.rmi.Remote)
- None of the remote capable methods have to throw java.rmi.RemoteException.
- Compared to RMI in use for EJB, it does not transport over CORBA (yet).

3. Connection Robustness

Given that there are eight fallacies of distributed computing, we feel it important to show that AltRMI is not ignoring these issues.
The principal benefit for a developer making beans or an application server is that RemoteException is missing. That does not mean that communications failure is ignored. AltRMI still illustrates communication failure via InvocationException which a subclass of RuntimeException. This basically allows the exception to be thrown, but not specified on each method (like RemoteException does). Many feel that allowing the bean developer to ignore the robustness issues is a mistake. We think not given the following.

1. The container could be programmed to know about InvocationException.
2. AltRMI has configurable policies that can help re-establish connection whilst in use.
3. Standard handling of RemoteException sucks.
4. It is difficult in EJB, in terms of coverage, to test your huge amounts of RemoteException handling code.
5. Most web-app uses of beans have a single "handler" place where pertinent exceptions are already caught.

3.1. 1. The container could be programmed to know about InvocationException

A lot of beans coding is 'bean invokes method in bean which invokes method in bean'. In this case there are several places in the invocation stack where the container's logic is delegating between beans. Container could easily handle failing connections and take multiple actions: re-establish report, redirect, abend services or server. If there is a configurable policy for such events that may include the invoking of methods in, say, 'contingency' beans.

3.2. 2. AltRMI has configurable policies that can help reestablish connection whilst in use.

AltRMI has a pluggable architecture for re-establishing connections (and reporting timings etc). Whilst in the middle of an invocation, if the connection is lost, AltRMI can try to re-establish the connection and complete the method invocation normally. A delay would of course be encountered, but if administrators are watching the logs, then they can determine where failures are happening and what to do long term about it. Programmed policies (configured in the container) could be "try perpetually to reconnect", "try five times only, one a second", "fail immediately".

3.3. 3. Standard handling of RemoteException sucks.

Referring to the various ways EJB teams handle RemoteException, in the thousands of places in a typical J2EE solution where it is thrown, different solutions are...

3.3.1. 3.1. Declare throws RemoteException on every applicable method.

That means that it can often arrive back at the container. The container always reports it verbosely.

3.3.2. 3.2. Have a standard catch block and pass the RemoteException to a standard handle method that does something with it.

That something can often by turn it into a custom derivative of RuntimeException as well as reporting it. This strategy makes you wonder why it was not a derivative of RuntimeException in the first place.

3.3.3. 3.3. Try the failing method call again, or n times.

Clutters your code with reams of retry logic. What if it still fails? Combine this with (1) or (2) as well?

3.4. 4. It is difficult in EJB, in terms of coverage, to test your huge amounts of RemoteException handling code

Your EJB team has developed a huge amount of code for the business logic, and consequentually loads of code concerning RemoteException. Question how do they test the "connection failing" logic? Do they rip out cables while the machine is in use? No that does not yield good coverage. Do they have test cases and mock beans that throw RemoteException? Yes probably, but that is an artificial connection outage. However most teams do not test more than a single case, and are happy for the same RemoteException handler block to be used all over the place.

3.5. 5. Most web-app uses of beans have a single 'handler' place where pertinent exceptions are already caught.

Webapps that use multiple beans (assuming a decent MVC separation or a framework like Velocity) already have a place where central exception handling is going on. With AltRMI, you can catch InvocationException where you feel is fit. EJB teams that choose to have throws RemoteException on all methods (percolating it up the stack) probably also choose to finally handle it centrally. Like so ...

```
public Template handleRequest(HttpServletRequest req, HttpServletResponse resp, Context
    Template template = null;
    String templateName = null;
    HttpSession sess = req.getSession();
    sess.setAttribute(ERR_MSG_TAG, "all ok");
    try {
        try {
            // Process the command
            templateName = processRequest(req, resp, ctx);
            // Get the template
            template = getTemplate(templateName);
        } catch (InvocationException aie) {
            template = getTemplate("commfailure.vm");
        }
    } catch (ResourceNotFoundException rnfe) {
```

```
        // blah blah
    } catch (ParseErrorException pee) {
        // blah blah
    } catch (Exception e) {
        // blah blah
    }
    return template;
}
```

4. Things yet to do

There is an ongoing plan for features to be added to AltRMI. On the transports page, there are some related future requirements listed. Below are the big features yet to do.

4.1. Callback

We have implemented an experimental callback structure. We have had to make the communication asynchronous to do this.

4.2. True dynamic creation of Proxies

We curently use javac to compile stubs from source. It feels natuaral to use this technique as we think in terms of the Java the language. We know that the main interface to Javac is deprecated in JDK1.4 and feel we should move to some less static and more beanlike tool. An obvious choice would be BCEL and we are working on an implementation.

4.3. Secure Transports

Some variations on the current transports to allow SSL connections.

5. External uses of AltRMI

5.1. Instrument project - Avalon

The Excalibur Instrument package as part of Avalon.

5.2. Enterprise Object Broker

An EJB replacement, current hosted at SourceForge

5.3. Marathon Man

A functional test runner for Swing applications at SourceForge