Pluggable Transports

by Paul Hammant

1. Introduction

AltRMI has pluggable and reimplementable transports. They differ in terms of speed and layers of transport. Some are in VM, others between VMs using sockets and various Java concepts.

2. Supplied Request/Response Transports

The supplied transports fall into two categories - Intra-JVM and Inter-JVM. The Inter-JVM types are for bridging a network divide over TCP/IP. This can also mean two JVMs in the same physical machine, using local-loop TCP/IP. The Intra-JVM types are for situations where normal dynamic proxy will not work. For example when the client and the server both have a definition of the same interface in different classloaders. Most Java projects do not involve trees of classloaders, but writing frameworks like Avalon-Phoenix or or an implementation of the EJB specification will.

All of these transports are synchronous too. That means that an invocation across their connection will wait until it is completed server side before the next invocation is allowed through.

2.1. Plain Sockets / ObjectStream & CustomStream varients

This transport is a streaming type that uses serialization of objects over a TCP/IP connection. There are two variations. The first uses java.io.ObjectInputStream & java.io.ObjectOutputStream (AKA 'ObjectStream', the second uses what we call 'CustomStream'. CustomStream came into being because of this bug http://developer.java.sun.com/developer/bugParade/bugs/4499841.html which seriously restricts the usefulness of ObjectStream as a transport (and the same java.io classes for other uses). Custom Stream is slower by 20%, but we recommend it's use over ObjectStream. At least until the bug at Sun is fixed (please vote for it).

2.2. Over RMI

This is another transport that bridges two different JVMs using TCP/IP. It is actually the fastest of all the TCP/IP using transports. and takes advantage of RMI as it's transport while hiding RMI from the AltRMI client and server.

2.3. Piped with same VM / ObjectStream & CustomStream varients

In a similar way to the ObjectStream and CustomStream implementations of the plain sockets transport, these offer transport using a pipe inside the JVM. Not needed for most users of AltRMI these prove useful for developers making complex trees of classloaders with high separation from each other. As a Pipe is being used there is is some opportunity for buffering of invocations. This might slow the throughput down but this may relieve other parts of a particular design.

2.4. Direct within same VM

There are 'Direct' and 'DirectMarshalled' transports. These are use useful in the same scenarios as the Piped one, but with some small differences. Principally, there is no pipe - the invocation is immediately handled on the server side. With Direct there is also the fact that all mutually visible classes and interfaces would have to be in a commonly visible classloader. With DirectMarshalled, there can be duplicate interfaces and class definitions as in the streamed types of transport.

2.5. JNDI

Many of the basic transport types are accessible client side through JNDI. This makes the client side usage more standards compliant, but these is no need to choose it over the bespoke AltRMI client usage at all.

3. Supplied Callback capable Transports

All of these transports are asynchronous. Thais means that an invocation across the connection does not wait until the reply is ready before it allows another request though. This allows two things - excpetionally lengthy requests (that might ordinarily affect timeouts) to be performed and callbacks (server invoking requests on the client). There is a small (15%) cost to using this transport for simple cases, but its benefits outweigh its deficiencies.

Whilst the Callback enabled transports are better from the point of view of asynchronous behaviour

3.1. xxx

4. Future Transports

- SOAP - Might require additional undynamic "toWSDL()" step.
- CORBA - Might require additional undynamic "toIDL()" step.
- JMS
- UDP
- Over RMI over IIOP
- Over JMS
- Over RMI over HTTP
- Over HTTP (custom impl)
- TLS enabled versions of many of the above.

5. Speed

Counting the number of 'void testSpeed()' invocations in 10 seconds, we can guage the differences (Paul's Athlon900/Win2K machine)

5.1. AltRMI types over TCP/IP

For remote publication
Speed Test type Count Relative
------------------------------ ------- --------
a) ObjectStream over sockets #2 2702 1.00
b) Over RMI 4359 1.61
c) CustomStream over sockets 6069 2.25
d) ObjectStream over sockets #1 10088 3.73

5.2. AltRMI types in the same VM

These are useful for complete classloader separation of interface & implementation using different classloaders. The implementation and 'remote' proxy do not need to see the same interfaces etc..
Speed Test type Count Relative
------------------------------ ------- --------
e) ObjectStream over Pipe #2 12095 4.48
f) Direct Marshalled #3 20759 7.68
g) ObjectStream over Pipe #1 61166 22.64

h) Direct Unmarshalled #4 2391498 885.08
#1 Without calling reset() as workaround to the ObjectStream bug #2 With calling reset() as workaround to the ObjectStream bug #3 Completely separates classloaders of client and server. Requires a thread for each though. #4 Good as DynamicProxy for separation. Does not separate classloaders of client and server.

5.3. Non AltRMI types

- In VM, without using AltRMI - for comparison. - The inteface, implementation and proxy cannot be separated in terms of branches of classloader for these three. The same interfaces etc must be visible to both implementation and proxy.
Speed Test type Count Relative
------------------------------ ------- --------
i) DyanmicProxy #5
(copied from Excalibur) 20282070 7506.32
j) Hand-coded proxy #5 41214422 15253.30
k) No Proxy #5 42384804 15686.46
#4 - For all of these three, the actual timing may slow down the test.

6. Secrets of classloading

There is a feature of classloading that affects the way that the an AltRMI using client and server interoperte when it comes to resolving classes and interfaces for a given object. As is widely known, the JVM resolves depended on classes for a being-instantiated object at runtime. The issue concerns a class definition existing twice in a tree of classloaders and whther the JVM considers an instance of each to be of the same type.

Consider a tree of three classloaders - A, B and C. Consider that A is the parent classloader of B & C. This means that B can access all the classes mounted by itself and by A. Similarly C can access all the classes mounted by itself and A. Now if A had a singleton that stored a single object via **void setObject(object o);** and **Object getObject();**, and clases in B amp; C could invoke those methods freely, the you might consider that B has a way of taking to C. if B called (essentially) **A.setObject("Hello")**, then C could indeed call **String s = A.getObject()** without any problem. Say a class being passed were called 'Thing' and was in the classloader of B and duplicated in the classloader of C, but not in A at all, then it would not be passable by the setter/getter mechanism outlined above. Why? The JVM considers then different classes because they are mounted in different classloaders (even though from the same source). That is a secret of classloading (at least as it pertains to RPC in one VM).

The issue is relevent to AltRMI mostly if it is being used to connect two nodes of a single classloader tree. If the transport chosen is 'Direct' then you will get ClassCastExceptions

thrown by the JVM if you had been passed an Object you wanted to cast up to something, and that something were represented by a class definition in both the server and client nodes of the classloader tree. If the something class were in a mutualy visible parent class loader then no issue would be apparent. IF the client and server were in separate VMs, then no issue would be apparent, principally because the marchalling to serialized form neatly hides the two class definitions from the JVM. This is the clue to the solving of the issue for a particular client/server (in one JVM) configuration you may be cooking up. If you choose Piped or DirectMarshalled as trasnports, then you can have the same class definition in multiple classloader nodes. Of couse, both Piped and DirectMarshalled are slower than Direct as transports. Configuration choices for the developer/deployer.

Copyright (c) @year@ The Apache Incubator Project. All rights reserved. $Revision: 1.2 $ $Date: 2003/02/16 21:41:35 $