JBoss Remoting Guide

JBoss Remoting version 2.2.3.SP1

September 15, 2009

Copyright © 2009 JBoss, a division of Red Hat .

Table of Contents

1. Overview 1
    1.1. What is JBoss Remoting 1
    1.2. Features 1
    1.3. How to get JBoss Remoting 2
    1.4. What's new? 3
        1.4.1. In release 2.2.3.SP1 3
        1.4.2. In release 2.2.3 3
        1.4.3. In release 2.2.2.SP11 3
        1.4.4. In release 2.2.2.SP10 3
        1.4.5. In release 2.2.2.SP9 3
        1.4.6. In release 2.2.2.SP8 3
        1.4.7. In release 2.2.2.SP7 4
        1.4.8. In release 2.2.2.SP4 4
        1.4.9. In release 2.2.2.SP2 4
        1.4.10. In release 2.2.2.GA 4
2. Architecture 5
3. JBoss Remoting Components 9
    3.1. Discovery 11
    3.2. Transports 11
4. Remoting libraries and thirdparty dependancies 13
    4.1. Third party libraries 14
5. Configuration 19
    5.1. General transport configuration 19
        5.1.1. Server side configuration 19
            5.1.1.1. Programmatic configuration. 19
            5.1.1.2. Declarative configuration 21
            5.1.1.3. Callback client configuration 23
        5.1.2. Client side configuration 23
    5.2. Handlers 24
    5.3. Discovery (Detectors) 28
    5.4. Transports (Invokers) 30
        5.4.1. Features introduced in Remoting version 2.2 30
            5.4.1.1. Binding to 0.0.0.0 30
            5.4.1.2. Support for IPv6 addresses 31
        5.4.2. Server Invokers 31
        5.4.3. Configurations affecting the invoker client 31
        5.4.4. How the server bind address and port is determined 32
        5.4.5. Socket Invoker 33
            5.4.5.1. How the Socket Invoker works 34
        5.4.6. SSL Socket Invoker 36
        5.4.7. RMI Invoker 36
        5.4.8. SSL RMI Invoker 37
        5.4.9. HTTP Invoker 37
        5.4.10. HTTPS Invoker 38

5.4.11. HTTP(S) Client Invoker - proxy and basic authentication 38
5.4.12. Servlet Invoker 40
5.4.12.1. Configuration 42
5.4.13. SSL Servlet Invoker 42
5.4.14. Exception handling for web based clients 43
5.4.15. Multiplex Invoker 44
5.4.15.1. Setting up the server 44
5.4.15.2. Setting up the client 45
5.4.15.3. Shutting down invoker groups. 48
5.4.15.4. Examples 49
5.4.15.5. Configuration properties 51
5.4.16. SSL Multiplex Invoker 52
5.4.17. Bisocket invoker 52
5.4.17.1. Overview 53
5.4.17.2. Details 54
5.4.18. SSL Bisocket invoker 57
5.5. Marshalling 57
5.6. Callbacks 60
5.6.1. Callback overview 60
5.6.1.1. Callback connections 60
5.6.1.2. Transmitting callbacks 61
5.6.1.3. Callback stores. 62
5.6.1.4. Callback acknowledgements 63
5.6.2. Registering callback handlers. 64
5.6.2.1. Pull callbacks. 64
5.6.2.2. Push callbacks. 65
5.6.3. Unregistering callback handlers 68
5.6.4. Callback store configuration. 69
5.6.5. Callback Exception Handling 71
5.7. Socket factories and server socket factories 71
5.7.1. Server side programmatic configuration 72
5.7.1.1. Server socket factories. 72
5.7.1.2. Socket factories 73
5.7.2. Client side programmatic configuration 74
5.7.2.1. Server socket factories. 75
5.7.2.2. Socket factories. 76
5.7.3. Server side configuration in the JBoss Application Server 77
5.7.4. Client side configuration from the JBoss Application Server 78
5.7.5. Socket creation listeners 78
5.7.6. SSL transports 79
5.7.7. SSLSocketBuilder 81
5.7.8. SSLServerSocketFactoryService 87
5.7.9. General Security How To 87
5.7.10. Troubleshooting Tips 88
5.8. Timeouts 89
5.8.1. General timeout configuration 89
5.8.2. Per invocation timeouts 89
5.8.3. Transport specific timeout handling 90
5.8.3.1. Socket and bisocket transports 90

5.8.3.2. HTTP transport 90
5.8.3.3. Quick client disconnect 91
5.9. Configuration by properties 91
6. Sending streams 103
6.1. Configuration 104
6.2. Issues 104
7. Remote classloading facility 105
7.1. Classloading in client invokers 105
7.2. Server side support for remote classloading 106
8. Serialization 108
9. Network Connection Monitoring 109
9.1. Client side monitoring 109
9.2. Server side monitoring 110
9.3. Interactions between client side and server side connection monitoring 112
10. Transporters - beaming POJOs 114
11. How to use it - sample code 115
11.1. Simple invocation 115
11.2. HTTP invocation 116
11.3. Oneway invocation 119
11.4. Discovery and invocation 120
11.5. Callbacks 121
11.6. Streaming 123
11.7. JBoss Serialization 124
11.8. Transporters 125
11.8.1. Transporters - beaming POJOs 125
11.8.2. Transporters sample - simple 126
11.8.3. Transporter sample - basic 128
11.8.4. Transporter sample - JBoss serialization 133
11.8.5. Transporter sample - clustered 138
11.8.6. Transporters sample - multiple 143
11.8.7. Transporters sample - proxy 146
11.8.8. Transporter sample -complex 150
11.9. Multiplex invokers 152
12. Client programming model 154
13. Compatibility and versioning 155
14. Getting the JBossRemoting source and building 156
15. Known issues 158
16. Future plans 159
17. Release Notes 160

1

Overview

1.1. What is JBoss Remoting

The purpose of JBoss Remoting is to provide a single API for most network based invocations and related service that uses pluggable transports and data marshallers. The JBossRemoting API provides the ability for making synchronous and asynchronous remote calls, push and pull callbacks, and automatic discovery of remoting servers. The intention is to allow for the use of different transports to fit different needs, yet still maintain the same API for making the remote invocations and only requiring configuration changes, not code changes.

JBossRemoting is a standalone project, separate from the JBoss Application Server project, but will be the framework used for many of the JBoss projects and components when making remote calls. JBossRemoting is included in the recent releases of the JBoss Application Server and can be run as a service within the container as well. Service configurations are included in the configuration section below.

1.2. Features

The features available with JBoss Remoting are:

- **Server identification** – a simple url based identifier which allows for remoting servers to be identified and called upon.
- **Pluggable transports** – can use different protocol transports the same remoting API.

  Provided transports:

  - Socket (SSL Socket)
  - RMI (SSL RMI)
  - HTTP(S)
  - Multiplex (SSL Multiplex)
  - Servlet (SSL Servlet)
  - BiSocket (SSL BiSocket)
- **Pluggable data marshallers** – can use different data marshallers and unmarshallers to convert the invocation payloads into desired data format for wire transfer.

- **Pluggable serialization** - can use different serialization implementations for data streams.

  Provided serialization implementations:

  - Java serialization
  - JBoss serialization

- **Automatic discovery** – can detect remoting servers as they come on and off line.

  Provided detection implementations:

  - Multicast
  - JNDI

- **Server grouping** – ability to group servers by logical domains, so only communicate with servers within specified domains.
- **Callbacks** – can receive server callbacks via push and pull models. Pull model allows for persistent stores and memory management.
- **Asynchronous calls** – can make asynchronous, or one way, calls to server.
- **Local invocation** – if making an invocation on a remoting server that is within the same process space, remoting will automatically make this call by reference, to improve performance.
- **Remote classloading** – allows for classes, such as custom marshallers, that do not exist within client to be loaded from server.
- **Sending of streams** – allows for clients to send input streams to server, which can be read on demand on the server.
- **Clustering** - seamless client failover for remote invocations.
- **Connection failure notification** - notification if client or server has failed
- **Data Compression** - can use compression marshaller and unmarshaller for compresssion of large payloads.

All the features within JBoss Remoting were created with ease of use and extensibility in mind. If you have a suggestion for a new feature or an improvement to a current feature, please log in our issue tracking system at http://jira.jboss.com

1.3. How to get JBoss Remoting

The JBossRemoting distribution can be downloaded from http://labs.jboss.com/portal/jbossremoting [http://labs.jboss.com/portal/jbossremoting] . This distribution contains everything needed to run JBossRemoting stand alone. The distribution includes binaries, source, documentation, javadoc, and sample code.

1.4. What's new?

1.4.1. In release 2.2.3.SP1

1. Introduction of write timeout facility;
2. improved reliability for callbacks in bisocket transport;
3. improved treatment of invocation retries in socket and bisocket transports;
4. miscellaneous bug fixes.

1.4.2. In release 2.2.3

1. Introduction of "connection identity" concept;
2. upgrading of servlet transport;
3. improved configuration flexibility;
4. miscellaneous bug fixes.

1.4.3. In release 2.2.2.SP11

1. More flexible classloading by client side unmarshallers;
2. miscellaneous bug fixes.

1.4.4. In release 2.2.2.SP10

1. `CoyoteInvoker` stores `InvokerLocator` query in `InvocationRequest`;
2. miscellaneous bug and documentation fixes.

1.4.5. In release 2.2.2.SP9

1. Fixed potential build up of cancelled `TimerTasks` in bisocket transport;
2. it's possible to turn on server side connection monitoring (leasing) declaratively;
3. miscellaneous bug fixes.

1.4.6. In release 2.2.2.SP8

1. Remoting remote classloading facility now works with isolated EARs;
2. `InvokerLocator` can process zone indices;
3. improved client side connection monitoring;
4. miscellaneous bug fixes.

1.4.7. In release 2.2.2.SP7

1. Server side and client side connection listeners can be tied together.

1.4.8. In release 2.2.2.SP4

1. IPv6 addresses are supported;
2. `org.jboss.remoting.callback.ServerInvokerCallbackHandler` can register itself as a lease connection listener.

1.4.9. In release 2.2.2.SP2

1. The servlet transport can throw an exception generated on the server side;
2. servers can bind to 0.0.0.0.

1.4.10. In release 2.2.2.GA

Release 2.2.2.GA includes a number of bug fixes, greater configurability, and a couple of new features, including

1. an improved callback polling method;
2. the ability for the client to discover its IP address as seen by the server side of the connection;

The following changes affect configurability:

1. The address and port of the bisocket transport secondary server socket are configurable;
2. `org.jboss.remoting.ConnectorValidator` parameters are configurable;
3. the maximum number of errors before a `org.jboss.remoting.callback.CallbackPoller` shuts down can be specified;
4. there is a separate timeout parameter for callbacks.

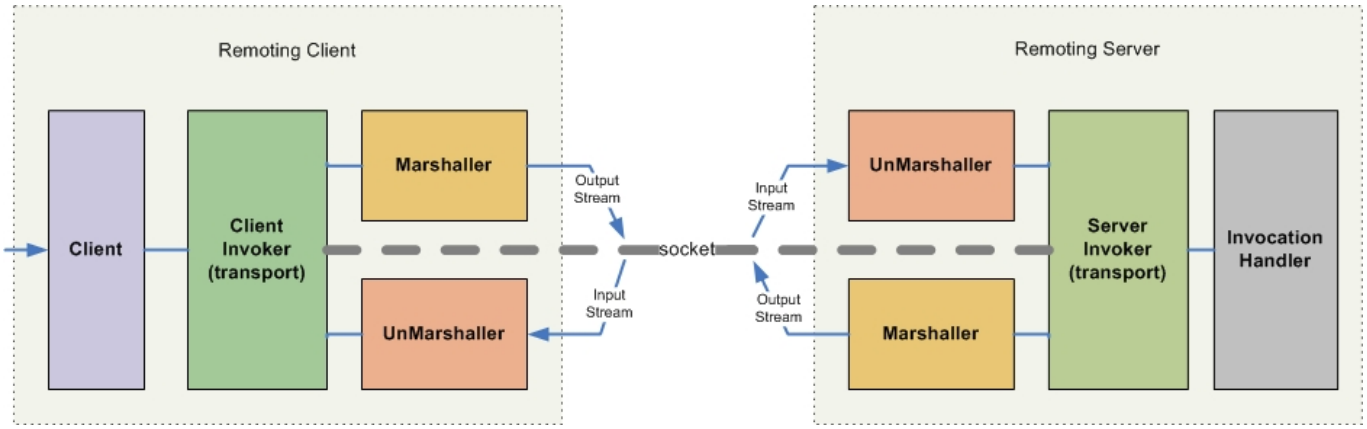
For the JIRA items related to release 2.2.2.GA, see Release Notes.

2

Architecture

The most critical component of the JBoss Remoting architecture is how servers are identified. This is done via an InvokerLocator, which can be represented by a simple String with a URL based format (e.g., socket://myhost:5400). This is all that is required to either create a remoting server or to make a call on a remoting server. The remoting framework will then take the information embedded within the InvokerLocator and construct the underlying remoting components needed and build the full stack required for either making or receiving remote invocations.

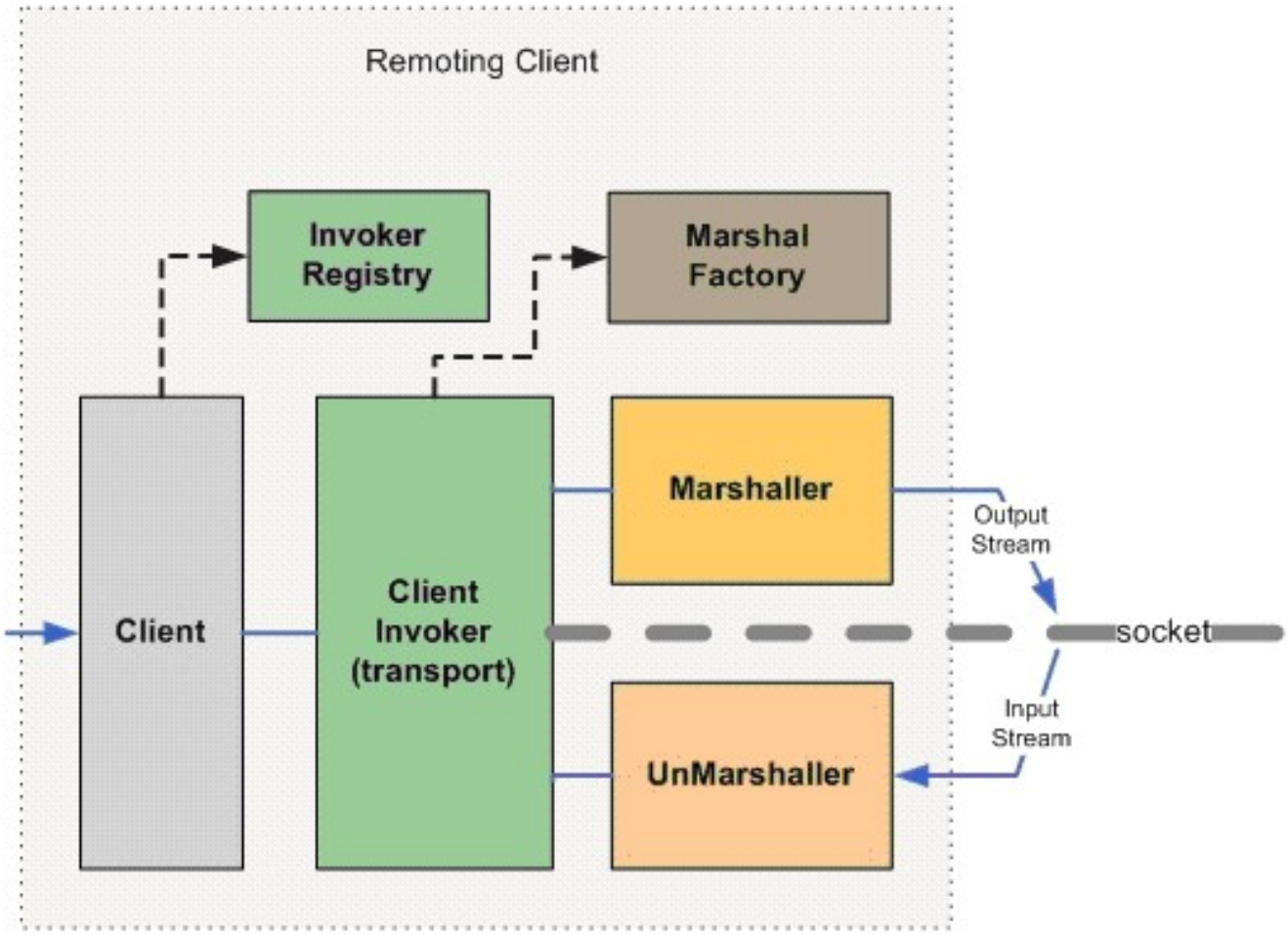
There are several layers to this framework that mirror each other on the client and server side. The outermost layer is the one which the user interacts with. On the client side, this is the Client class upon which the user will make its calls. On the server side, this is the InvocationHandler, which is implemented by the user and is the ultimate receiver of invocation requests. Next is the transport, which is controlled by the invoker layer. Finally, at the lowest layer is the marshalling, which converts data type to wire format.



When a user calls on the Client to make an invocation, it will pass this invocation request to the appropriate client invoker, based on the transport specified by the locator url. The client invoker will then use the marshaller to convert the invocation request object to the proper data format to send over the network. On the server side, an unmarshaller will receive this data from the network and convert it back into a standard invocation request object and send it on to the server invoker. The server invoker will then pass this invocation request on to the user's implementation of the invocation handler. The response from the invocation handler will pass back through the server invoker and on to the marshaller, which will then convert the invocation response object to the proper data format and send back to the client. The unmarshaller on the client will convert the invocation response from wire data format into standard invocation response object, which will be passed back up through the client invoker and Client to the original caller.

Client

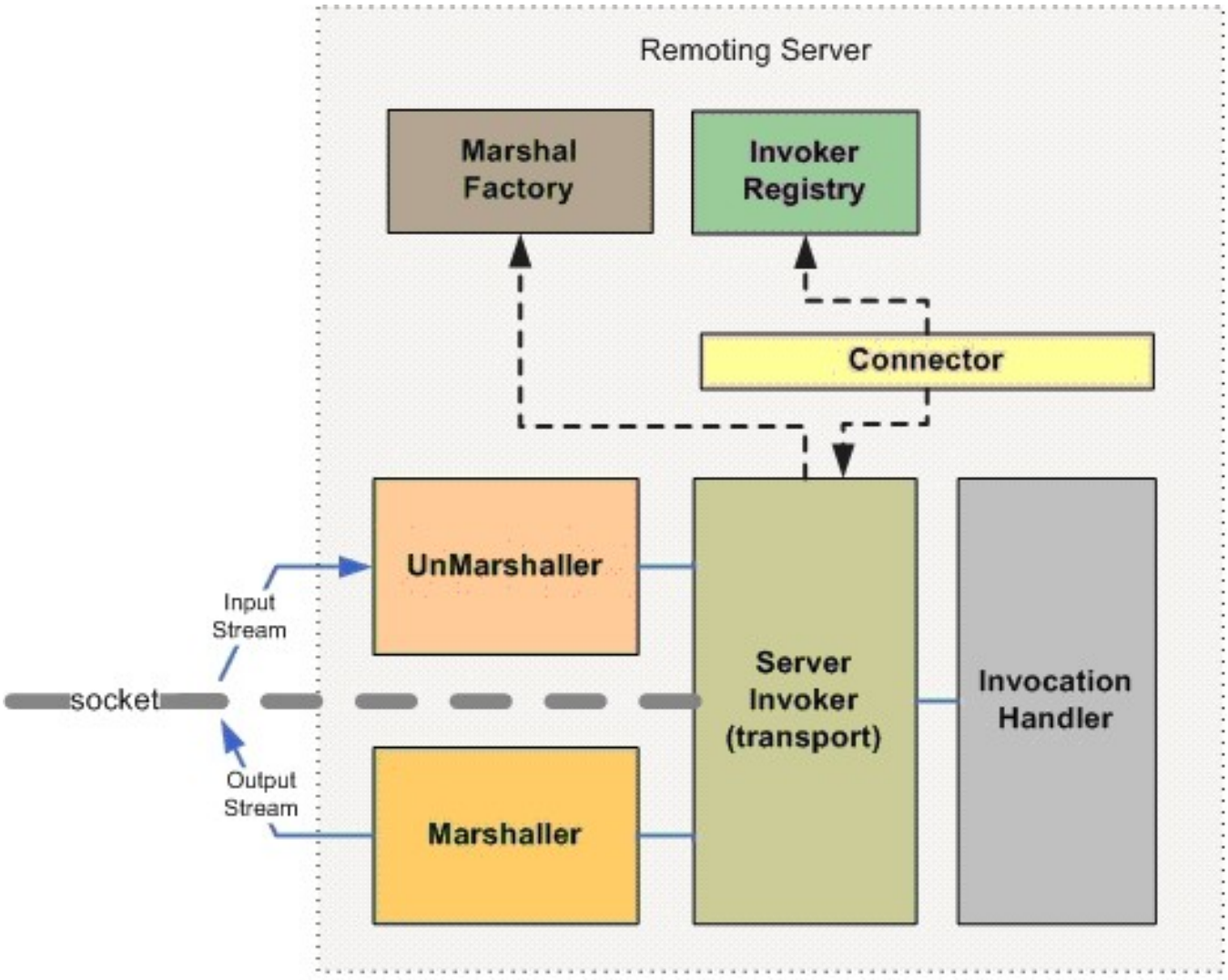
On the client side, there are a few utility class that help in figuring out which client invoker and marshal instances should be used.



For determining which client invoker to use, the Client will pass the InvokerRegistry the locator for the target server it wishes to make invocations on. The InvokerRegistry will return the appropriate client invoker instance based on information contained within the locator, such as transport type. The client invoker will then call upon the MarshalFactory to get the appropriate Marshaller and UnMarshaller for converting the invocation objects to the proper data format for wire transfer. All invokers have a default data type that can be used to get the proper marshal instances, but can be overridden within the locator specified.

Server

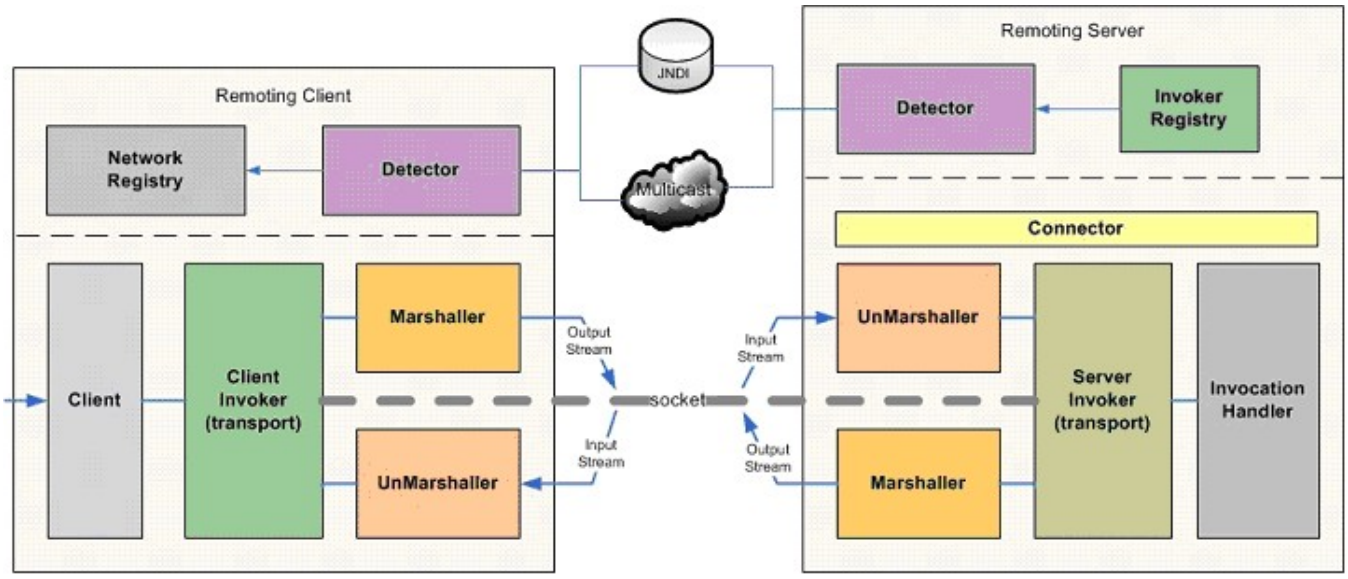
On the server side, there are also a few utility classes for determining the appropriate server invoker and marshal instances that should be used. There is also a server specific class for tying the invocation handler to the server invoker.



On the server side, it is the Connector class that is used as the external point for configuration and control of the remoting server. The Connector class will call on the InvokerRegistry with its locator to create a server invoker. Once the server invoker is returned, the Connector will then register the invocation handlers on it. The server invoker will use the MarshalFactory to obtain the proper marshal instances as is done on the client side.

Detection

To add automatic detection, a remoting Detector will need to be added on both the client and the server side as well as a NetworkRegistry to the client side.



When a Detector on the server side is created and started, it will periodically pull from the InvokerRegistry all the server invokers that it has created. The detector will then use the information to publish a detection message containing the locator and subsystems supported by each server invoker. The publishing of this detection message will be either via a multicast broadcast or a binding into a JNDI server. On the client side, the Detector will either receive the multicast broadcast message or poll the JNDI server for detection messages. If the Detector determines a detection message is for a remoting server that just came online it will register it in the NetworkRegistry. The NetworkRegistry houses the detection information for all the discovered remoting servers. The NetworkRegistry will also emit a JMX notification upon any change to this registry of remoting servers. The change to the NetworkRegistry can also be for when a Detector has discovered that a remoting server is no longer available and removes it from the registry.

3

JBoss Remoting Components

This section covers a few of the main components exposed within the Remoting API with a brief overview.

org.jboss.remoting.Client – is the class the user will create and call on from the client side. This is the main entry point for making all invocations and adding a callback listener. The Client class requires only the InvokerLocator for the server you wish to call upon and that you call connect before use and disconnect after use (which is technically only required for stateful transports and when client leasing is turned on, but good to call in either case).

org.jboss.remoting.InvokerLocator – is a class, which can be described as a string URI, for identifying a particular JBossRemoting server JVM and transport protocol. For example, the InvokerLocator string socket://192.168.10.1:8080 describes a TCP/IP Socket-based transport, which is listening on port 8080 of the IP address, 192.168.10.1. Using the string URI, or the InvokerLocator object, JBossRemoting can make a client connection to the remote server. The format of the locator string is the same as the URI type: `[transport]://[host]:<port>/path/?<parameter=value>&<parameter=value>`

A few important points to note about the InvokerLocator. The string representation used to construct the InvokerLocator may be modified after creation. This can occur if the host supplied is 0.0.0.0, in which case the InvokerLocator will attempt to replace it with the value of the local host name. This can also occur if the port specified is less than zero or not specified at all (in which case remoting will select a random port to use).

The InvokerLocator will accept host name as is and will not automatically convert to IP address (since 2.0.0 release). There are two comparison operators for InvocatorLocators, `equals()` and `isSameEndpoint()`, and neither resolve a hostname to IP address or vice versa. `equals()` compares all components of the InvokerLocator, character by character, while `isSameEndpoint()` uses only protocol, host, and port. The following examples are just some of the comparisons that can be seen in `org.jboss.test.remoting.locator.InvokerLocatorTestCase`:

`new InvokerLocator("http://localhost:1234/services/uri:Test").equals(new InvokerLocator("http://localhost:1234"))` returns false

`new InvokerLocator("http://localhost:1234/services/uri:Test").equals(new InvokerLocator("http://127.0.0.1:1234"))` returns false

`new InvokerLocator("http://localhost:1234/services/uri:Test").isSameEndpoint(new InvokerLocator("http://localhost:1234"))` returns true

`new InvokerLocator("http://localhost:1234/services/uri:Test").isSameEndpoint(new InvokerLocator("http://127.0.0.1:1234"))` returns false

org.jboss.remoting.transport.Connector - is an MBean that loads a particular ServerInvoker implementation for a given transport subsystem and one or more ServerInvocationHandler implementations that handle Subsystem invocations on the remote server JVM. The Connector is the main user touch point for configuring and managing a remoting server.

org.jboss.remoting.ServerInvocationHandler – is the interface that the remote server will call on with an invocation received from the client. This interface must be implemented by the user. This implementation will also be required to keep track of callback listeners that have been registered by the client as well.

org.jboss.remoting.InvocationRequest – is the actual remoting payload of an invocation. This class wraps the caller's request and provides extra information about the invocation, such as the caller's session id and its callback locator (if one exists). This will be object passed to the ServerInvocationHandler.

org.jboss.remoting.stream.StreamInvocationHandler – extends the ServerInvocationHandler interface and should be implemented if expecting to receive invocations containing an input stream.

org.jboss.remoting.callback.InvokerCallbackHandler – the interface for any callback listener to implement. Upon receiving callbacks, the remoting client will call on this interface if registered as a listener.

org.jboss.remoting.callback.Callback – the callback object passed to the InvokerCallbackHandler. It contains the callback payload supplied by the invocation handler, any handle object specified when callback listener was registered, and the locator from which the callback came.

org.jboss.remoting.network.NetworkRegistry – this is a singleton class that will keep track of remoting servers as new ones are detected and dead ones are detected. Upon a change in the registry, the NetworkRegistry fires a NetworkNotification.

org.jboss.remoting.network.NetworkNotification – a JMX Notification containing information about a remoting server change on the network. The notification contains information in regards to the server's identity and all its locators.

org.jboss.remoting.detection.Detection – is the detection message fired by the Detectors. Contains the locator and subsystems for the server invokers of a remoting server as well as the remoting server's identity.

org.jboss.remoting.ident.Identity – is one of the main components remoting uses during discovery to identify remoting server instances (is actually the way it guarantees uniqueness). If have two remoting servers running on the same server, they can be uniquely identified. The reason the identity is persisted (currently only able to do this to the file system) is so if a server crashes and then restarts, can identify it when it restarts as the one that crashed (instead of being a completely new instance that is being started). This may be important from a monitoring point as would want to know that the crashed server is back online.

When creating the identity to be presisted, remoting will first look to see if a system property for 'jboss.identity' has been set already. If it has, will use that one. If not, will get the value for the 'ServerDataDir' attribute of the 'jboss.system:type=ServerConfig' mbean. If can retrieve this value, will use this as the directory to write out the 'jboss.identity' file. If not, will look to see if a system property has been set for 'jboss.identity.dir'. If it has, will use this as the directory to write the 'jboss.identity' file to, otherwise, will default to '.'. If for some reason the file can not be written to, will throw a RuntimeException, which will cause the detector to error during startup. For more details on how and where the identity is persisted, can refer to org.jboss.remoting.ident.Identity.createId().

org.jboss.remoting.detection.multicast.MulticastDetector – is the detector implementation that broadcasts its Detection message to other detectors using multicast.

org.jboss.remoting.detection.jndi.JNDIDetector – is the detector implementation that registers its Detection message to other detectors in a specified JNDI server.

There are a few other components that are not represented as a class, but important to understand.

Subsystem – a sub-system is an identifier for what higher level system an invocation handler is associated with. The sub-system is declared as any String value. The reason for identifying sub-systems is that a remoting Connector's server invoker may handle invocations for multiple invocation handlers, which need to be routed based on sub-system. For example, a particular socket based server invoker may handle invocations for both customer processing and order processing. The client making the invocation would then need to identify the intended sub-system to handle the invocation based on this identifier. If only one handler is added to a Connector, the client does not need to specify a sub-system when making an invocation.

Domain – a logical name for a group to which a remoting server can belong. The detectors can discriminate as to which detection messages they are interested based on their specified domain. The domain to which a remoting server belongs is stored within the Identity of that remoting server, which is included within the detection messages. Detectors can be configured to accept detection messages from one, many or all domains.

3.1. Discovery

One of the features of JBoss Remoting is to be able to dynamically discover remoting servers. This is done through the use of what remoting calls detectors. These detectors run in same instance as the servers and the clients. The detectors that run within the server instance automatically gets list of remoting servers running locally and emits a detection message contain information about those servers, such as their locator url and subsystems supported. The detector running within the client instance will receive these detection messages and update a local registry, called the network registry, with this information. The client detector will also monitor the remoting servers it has discovered in case one were to fail, in which case, will notify the network registry of the failure The network registry will then fire events to registered listeners (via JMX notifications), to include events such as new server added or server failure.

There are currently two types of detector implementations; multicast and JNDI. The multicast detectors use multicast channel to send and receive detection messages. The JNDI detectors use a well known JNDI server to bind and lookup detection messages.

The standard approach for detecting remoting servers happens in a passive manner, in that as detection messages are received by the client detector, they will cause an event to fire. In some cases, will need ability to synchronously discover the remoting servers that exist upon startup. This can be done by calling the forceDetection() method on the detector. This will return an array of NetworkInstances which contains the server information. Note, this method can take a few seconds to return (at least in multicast implementation).

3.2. Transports

Service provider interface

The transport implementations within remoting, called invokers, are responsible for handling the wire protocol to be used by remoting clients and servers. Remoting will load client and server invoker (transport) implementations (within the InvokerRegistry) using factories. The factory class to be loaded will always be either TransportClientFactory (for loading client invoker) or TransportServerFactory (for loading server invoker). These classes must implement `org.jboss.remoting.transport.ClientFactory` and `org.jboss.remoting.transport.ServerFactory` interfaces respectively. The package under which the TransportClientFactory and TransportServerFactory will always start with `org.jboss.test.remoting.transport`, then the transport protocol type. For example, the 'socket'

transport factories are `org.jboss.remoting.transport.socket.TransportClientFactory` and `org.jboss.remoting.transport.socket.TransportServerFactory`. The API for org.jboss.remoting.transport.ClientFactory is:

```
   public ClientInvoker createClientInvoker(InvokerLocator locator, Map config) throws IOException;
   public boolean supportsSSL();
```

The API for org.jboss.remoting.transport.ServerFactory is:

```
   public ServerInvoker createServerInvoker(InvokerLocator locator, Map config) throws IOException;
   public boolean supportsSSL();
```

An example of a transport client factory for the socket transport (`org.jboss.remoting.transport.socket.TransportClientFactory`) is:

```
public class TransportClientFactory implements ClientFactory
{
   public ClientInvoker createClientInvoker(InvokerLocator locator, Map config)
         throws IOException
   {
      return new SocketClientInvoker(locator, config);
   }

   public boolean supportsSSL()
   {
      return false;
   }
}
```

The packages used within the factory does not matter as long as they are on the classpath. Note that the transport factories are only loaded upon request for that protocol. Also, the client and server factories have been separated so that only the one requested is loaded (and thus the corresponding classes needed for that invoker implementation). So if only ask for a particular client transport invoker, only those classes are loaded and the ones needed for the server are not required to be on the classpath.

The biggest reason for taking this approach is allows users ability to plugin custom transport implementation with zero config. Remoting comes with the following transports: socket, sslsocket, http, https, multiplex, sslmultiplex, servlet, sslservlet, rmi, sslrmi.

4

Remoting libraries and thirdparty dependancies

Remoting partitions its functionality into several different libraries to allow the size of the footprint to be controlled according to the features that will be used. Remoting distribution will include the following remoting binaries (found in the lib directory of the distribution).

jboss-remoting.jar - this binary contains all the remoting classes. This is the only remoting jar that is needed to perform any remoting function within JBoss Remoting. Of course, third party jars will be required.

Since some may want to better control size of the binary footprint needed to use remoting, the remoting classes have been broken out into multiple remoting binaries based on their function. There are four categories of these binaries; core, detection, transport, and other.

core

jboss-remoting-core.jar - contains all the core remoting classes needed for remoting to function. If not using jboss-remoting.jar, then jboss-remoting.core.jar will be required.

detection

jboss-remoting-detection - contains all the remoting classes needed to perform automatic discovery of remoting servers. It includes both the jndi and multicast detector classes as well as the network registry classes.

transport

jboss-remoting-socket.jar - contains all the classes needed for the socket and sslsocket transports to function as both a client and a server.

jboss-remoting-socket-client.jar - contains all the classes needed for the socket and sslsocket transports to function as a client only. This means will not be able to perform any push callbacks or sending of streams using this jar.

jboss-remoting-http.jar - contains all the classes needed for the http and https transports to function as a client and a server.

jboss-remoting-http-client.jar - contains all the classes needed for the http, https, servlet, and sslservlet transports to function as a client only. This means will not be able to perform any push callbacks or sending of streams using this jar.

jboss-remoting-servlet.jar - contains all the classes needed for the servlet or sslservlet transports to function as a server only (also requires servlet-invoker.war be deployed within web container as well).

jboss-remoting-rmi.jar - contains all the classes needed for the rmi and sslrmi transports to function as a client

and a server.

jboss-remoting-multiplex.jar - contains all the classes needed for the multiplex and sslmultiplex transports to function as a client and a server.

jboss-remoting-bisocket.jar - contains all the classes needed for the bisocket and sslbisocket transports to function as both a client and a server.

jboss-remoting-bisocket-client.jar - contains all the classes needed for the bisocket and sslbisocket transports to function as a client only. This means will not be able to perform any push callbacks or sending of streams using this jar.

other

jboss-remoting-serialization.jar - contains just the remoting serialization classes (and serialization manager implementations for java and jboss).

jboss-remoting-samples.jar - all the remoting samples showing example code for different remotng functions.

4.1. Third party libraries

This section covers which thirdparty jars are required based on the feature or transport to be used. Remember, any jboss-remoting-XXX.jar can be replaced with just the jboss-remoting.jar.

Table 1 gives the direct dependencies of the Remoting jars on third party jars (and jboss-remoting-core.jar, in most cases). Table 2 gives the transitive closure of Table 1.

The information in Tables 1 and 2 was derived with the use of the extremely useful **Tattletale** tool, available on the jboss.org website: http://www.jboss.org/tattletale [http://www.jboss.org/tattletale]

Table 4.1. Jar dependencies.

jar	depends on
jboss-remoting-bisocket-client.jar:	concurrent.jar, jboss-common.jar, jboss-remoting-core.jar, jboss-serialization.jar
jboss-remoting-bisocket.jar:	concurrent.jar, jboss-common.jar, jboss-remoting-core.jar, jboss-serialization.jar
jboss-remoting-core.jar:	concurrent.jar, jboss-common.jar, jboss-jmx.jar, jboss-serialization.jar, log4j.jar, saxon.jar
jboss-remoting-detection.jar:	jboss-common.jar, jboss-jmx.jar, jboss-remoting-core.jar, jnpserver.jar, log4j.jar, saxon.jar

jar	depends on
jboss-remoting-http-client.jar:	jboss-common.jar, jboss-remoting-core.jar
jboss-remoting-http.jar:	commons-httpclient.jar, jboss-common.jar, jboss-jmx.jar, jboss-remoting-core.jar, tomcat-coyote.jar, tomcat-util.jar
jboss-remoting-multiplex.jar:	concurrent.jar, jboss-common.jar, jboss-jmx.jar, jboss-remoting-core.jar, jboss-serialization.jar, log4j.jar
jboss-remoting-rmi.jar:	jboss-common.jar, jboss-remoting-core.jar
jboss-remoting-samples.jar:	dom4j.jar, jboss-common.jar, jboss-jmx.jar, jboss-remoting-core.jar, jboss-remoting-detection.jar, jboss-remoting-multiplex.jar, jnpserver.jar, junit.jar, saxon.jar, servlet-api.jar
jboss-remoting-serialization.jar:	jboss-common.jar, jboss-remoting-core.jar, jboss-serialization.jar
jboss-remoting-servlet.jar:	jboss-common.jar, jboss-remoting-core.jar, servlet-api.jar
jboss-remoting-socket-client.jar:	concurrent.jar, jboss-common.jar, jboss-remoting-core.jar
jboss-remoting-socket.jar:	concurrent.jar, jboss-common.jar, jboss-remoting-core.jar, jboss-serialization.jar
jboss-remoting.jar:	commons-httpclient.jar, concurrent.jar, dom4j.jar, jboss-common.jar, jboss-jmx.jar, jboss-serialization.jar, jnpserver.jar, junit.jar, log4j.jar, saxon.jar, servlet-api.jar, tomcat-coyote.jar, tomcat-util.jar

Table 4.2. Transitive closure of jar dependencies.

jar	depends on
jboss-remoting-bisocket-client.jar:	commons-httpclient.jar, commons-logging-api.jar, concurrent.jar, dom4j.jar, jboss-common.jar, jboss-

jar	depends on
	jmx.jar, jboss-remoting-core.jar, [jboss-remoting-detection.jar], jboss-serialization.jar, [jnpserver.jar], log4j.jar, trove.jar
jboss-remoting-bisocket.jar:	commons-httpclient.jar, commons-logging-api.jar, concurrent.jar, dom4j.jar, jboss-common.jar, jboss-jmx.jar, jboss-remoting-core.jar, [jboss-remoting-detection.jar], jboss-serialization.jar, [jnpserver.jar], log4j.jar, trove.jar
jboss-remoting-core.jar:	commons-httpclient.jar, commons-logging-api.jar, concurrent.jar, dom4j.jar, jboss-common.jar, jboss-jmx.jar, [jboss-remoting-detection.jar], jboss-serialization.jar, [jnpserver.jar], log4j.jar, trove.jar
jboss-remoting-detection.jar:	commons-httpclient.jar, commons-logging-api.jar, concurrent.jar, dom4j.jar, jboss-common.jar, jboss-jmx.jar, jboss-remoting-core.jar, jboss-serialization.jar, jnpserver.jar, log4j.jar, trove.jar
jboss-remoting-http-client.jar:	commons-httpclient.jar, commons-logging-api.jar, concurrent.jar, dom4j.jar, jboss-common.jar, jboss-jmx.jar, jboss-remoting-core.jar, [jboss-remoting-detection.jar], jboss-serialization.jar, [jnpserver.jar], log4j.jar
jboss-remoting-http.jar:	commons-httpclient.jar, commons-logging-api.jar, concurrent.jar, dom4j.jar, jboss-common.jar, jboss-jmx.jar, jboss-remoting-core.jar, [jboss-remoting-detection.jar], jboss-serialization.jar, [jnpserver.jar], log4j.jar, trove.jar
jboss-remoting-multiplex.jar:	commons-httpclient.jar, commons-logging-api.jar, concurrent.jar, dom4j.jar, jboss-common.jar, jboss-jmx.jar, jboss-remoting-core.jar, [jboss-remoting-detection.jar], jboss-serialization.jar, [jnpserver.jar], log4j.jar, trove.jar
jboss-remoting-rmi.jar:	commons-httpclient.jar, commons-logging-api.jar, concurrent.jar, dom4j.jar, jboss-common.jar, jboss-jmx.jar, jboss-remoting-core.jar, [jboss-remoting-detection.jar], jboss-serialization.jar, [jnpserver.jar], log4j.jar, trove.jar

jar	depends on
jboss-remoting-samples.jar:	commons-httpclient.jar, commons-logging-api.jar, concurrent.jar, dom4j.jar, jboss-common.jar, jboss-jmx.jar, jboss-remoting-core.jar, jboss-remoting-detection.jar, jboss-remoting-multiplex.jar, jboss-serialization.jar, jnpserver.jar, junit.jar, log4j.jar, servlet-api.jar, trove.jar
jboss-remoting-serialization.jar:	commons-httpclient.jar, commons-logging-api.jar, concurrent.jar, dom4j.jar, jboss-common.jar, jboss-jmx.jar, jboss-remoting-core.jar, [jboss-remoting-detection.jar], jboss-serialization.jar, [jnpserver.jar], log4j.jar, trove.jar
jboss-remoting-servlet.jar:	commons-httpclient.jar, commons-logging-api.jar, concurrent.jar, dom4j.jar, jboss-common.jar, jboss-jmx.jar, jboss-remoting-core.jar, [jboss-remoting-detection.jar], jboss-serialization.jar, [jnpserver.jar], log4j.jar, servlet-api.jar, trove.jar
jboss-remoting-socket-client.jar:	commons-httpclient.jar, commons-logging-api.jar, concurrent.jar, dom4j.jar, jboss-common.jar, jboss-jmx.jar, jboss-remoting-core.jar, [jboss-remoting-detection.jar], jboss-serialization.jar, [jnpserver.jar], log4j.jar, trove.jar
jboss-remoting-socket.jar:	commons-httpclient.jar, commons-logging-api.jar, concurrent.jar, dom4j.jar, jboss-common.jar, jboss-jmx.jar, jboss-remoting-core.jar, [jboss-remoting-detection.jar], jboss-serialization.jar, [jnpserver.jar], log4j.jar, trove.jar
jboss-remoting.jar:	commons-httpclient.jar, commons-logging-api.jar, concurrent.jar, dom4j.jar, jboss-common.jar, jboss-jmx.jar, jboss-remoting-detection.jar, jboss-serialization.jar, jnpserver.jar, junit.jar, log4j.jar, servlet-api.jar, tomcat-apr.jar, tomcat-coyote.jar, tomcat-util.jar, trove.jar

Notes.

1. jboss-remoting-core contains the transporter classes, and some of those need jboss-remoting-detection.jar, which explains the proliferation of jboss-remoting-detection.jar in Table 2. If transporters are not used, then jboss-remoting-detection.jar can be omitted. Moreover, JNDI detection requires jnpserver.jar, so that, if transporters are not

used, jnpserver.jar can be omitted.

2. jboss-common.jar needs commons-httpclient.jar, which explains the transitive dependency of all Remoting jars on commons-httpclient.jar. Also, commons-httpclient.jar needs commons-logging-api.jar.

3. Remoting requires the use of JMX classes. It does not require the JBoss implementation (jboss-jmx.jar) of JMX in order to function correctly, so can replace jboss-jmx.jar with another JMX implementation library (or exclude it if using jdk 1.5 or higher, which has JMX implementation built in).

4. Multicast detection requires jboss-remoting-detection.jar, concurrent.jar, dom4j.jar. JNDI detection: requires jboss-remoting-detection.jar, concurrent.jar, dom4j.jar, jnpserver.jar (for jndi api classes). dom4j.jar for use of detection is required by jboss-jmx.jar.

5

Configuration

This covers the configuration for JBoss Remoting discovery, connectors, marshallers, and transports. All the configuration properties specified can be set either via calls to the object itself, including via JMX (so can be done via the JMX or Web console), or via a JBoss AS service xml file. Examples of service xml configurations can be seen with each of the sections below. There is also an example-service.xml file included in the remoting distribution that shows full examples of all the remoting configurations.

5.1. General transport configuration

Remoting offers a variety of ways of configuring transports on the server side and client side. This section presents an overview, and the rest of the chapter elaborates the material presented here. For easy reference the configuration parameters discussed throughout the chapter are gathered together at the end of the chapter in section Configuration by properties

5.1.1. Server side configuration

The heart of the server side is the `Connector`, and it is through the `Connector` that the server side of a transport is configured. The central goals of configuration on the server side are to establish a server invoker and supply it with a set of invocation handlers. Only one invoker can be declared per `Connector`. Although declaring an invocation handler is not required, it should only be omitted in the case of declaring a callback server that will not receive direct invocations, but only callback messages. Otherwise client invocations can not be processed. The invocation handler is the only interface that is required by the remoting framework for a user to implement and will be what the remoting framework calls upon when receiving invocations.

There are two general approaches to server side configuration: programmatic and declarative. A variety of programmatic techniques work in any environment, including the JBoss Application Server (JBossAS). Moreover, JBossAS adds the option of declarative configuration. In particular, the SARDeployer (see The JBoss 4 Application Server Guide on the labs.jboss.org web site) can read information from a *-service.xml file and use it to configure MBeans such as `Connectors`.

5.1.1.1. Programmatic configuration.

The simplest way to configure a `Connector` is to pass an `InvokerLocator` to a `Connector` constructor. For example, the code fragment

```
String locatorURI = "socket://test.somedomain.com:8084";
String params = "/?clientLeasePeriod=10000&timeout=120000";
locatorURI += params;
InvokerLocator locator = new InvokerLocator(locatorURI);
Connector connector = new Connector(locator);
connector.create();
```

```
SampleInvocationHandler invocationHandler = new SampleInvocationHandler();
connector.addInvocationHandler("sample", invocationHandler);
connector.start();
```

creates a server invoker based on the socket transport, directs it to listen for invocations on port 8084 of host test.somedomain.com, and passes two configuration parameters, "clientLeasePeriod" and "timeout". It also supplies the server invoker with an invocation handler.

One limitation of the `InvokerLocator` is that it can only represent string values. An alternative that overcomes this limitation is to pass some or all of the parameters to the `Connector` by way of a configuration map. The following code fragment accomplishes all that the previous fragment does, but it passes one parameter by way of the `InvokerLocator` and passes the other by way of a configuration map. It also passes in a non-string object, a `ServerSocketFactory`:

```
String locatorURI = "socket://test.somedomain.com:8084";
String params = "/?clientLeasePeriod=10000";
locatorURI += params;
InvokerLocator locator = new InvokerLocator(locatorURI);
HashMap config = new HashMap();
config.put(ServerInvoker.TIMEOUT, 120000);
config.put(ServerInvoker.SERVER_SOCKET_FACTORY, new MyServerSocketFactory());
Connector connector = new Connector(locator, config);
connector.create();
SampleInvocationHandler invocationHandler = new SampleInvocationHandler();
connector.addInvocationHandler("sample", invocationHandler);
connector.start();
```

Note that the value of `ServerInvoker.TIMEOUT` is "timeout", and the value of `ServerInvoker.SERVER_SOCKET_FACTORY` is "serverSocketFactory". These configuration map keys are discussed throughout the chapter and accumulated in section Configuration by properties. Also, server socket factory configuration is covered in Socket factories and server socket factories.

A third programmatic option is available for those configuration properties which happen to be server invoker MBean properties. In the following fragment, the server invoker is obtained from the `Connector` and a `ServerSocketFactory` is passed to it by way of a setter method:

```
String locatorURI = "socket://test.somedomain.com:8084";
String params = "/?clientLeasePeriod=10000";
locatorURI += params;
InvokerLocator locator = new InvokerLocator(locatorURI);
HashMap config = new HashMap();
config.put(ServerInvoker.TIMEOUT, 120000);
Connector connector = new Connector(locator, config);
connector.create();
ServerInvoker serverInvoker = connector.getServerInvoker();
ServerSocketFactory ssf = new MyServerSocketFactory();
serverInvoker.setServerSocketFactory(ssf);
SampleInvocationHandler invocationHandler = new SampleInvocationHandler();
connector.addInvocationHandler("sample", invocationHandler);
connector.start();
```

Note. The `Connector` creates the server invoker during the call to `Connector.create()`, so this option only works

after that method has been called. Also, depending on the parameter and the transport, this option may or may not be effective after the call to `Connector.start()`, which calls `start()` on the server invoker.

A fourth option, which exists primarily to support the declarative mode of configuration presented below, is to pass an XML document to the `Connector`. The following fragment duplicates the behavior of the first and second examples above.

```
            HashMap config = new HashMap();
            config.put(ServerInvoker.TIMEOUT, 120000);
            Connector connector = new Connector(config);

            // Set xml configuration element.
            StringBuffer buf = new StringBuffer();
            buf.append("<?xml version=\"1.0\"?>\n");
            buf.append("<config>");
            buf.append("   <invoker transport=\"socket\">");
            buf.append("      <attribute name=\"serverBindAddress\">test.somedomain.com</attribute>");
            buf.append("      <attribute name=\"serverBindPort\">8084</attribute>");
            buf.append("      <attribute name=\"clientLeasePeriod\">10000</attribute>");
            buf.append("   </invoker>");
            buf.append("   <handlers>");
            buf.append("      <handler subsystem=\"mock\">");
            buf.append("         org.jboss.remoting.transport.mock.SampleInvocationHandler");
            buf.append("      </handler>");
            buf.append("   </handlers>");
            buf.append("</config>");
            ByteArrayInputStream bais = new ByteArrayInputStream(buf.toString().getBytes());
            Document xml = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(bais);
            connector.setConfiguration(xml.getDocumentElement());

            connector.create();
            connector.start();
```

Note that there is no `InvokerLocator` in this example. If the `Connector` gets an `InvokerLocator`, it ignores the presence of the xml document. Note also that this method only supports the use of string values, so it is necessary to include the fully qualified name of the invocation handler, from which the handler is created by calling the default constructor.

An example of this option in use can be found in `org.jboss.test.remoting.configuration.SocketClientConfigurationTestCase`.

5.1.1.2. Declarative configuration

The configuration option discussed at the end of the previous section, passing an XML document to the `Connector`, works in conjunction with the service archive deployer (SARDeployer) inside the JBoss Application Server to allow declarative configuration on the server side. In particular, the SARDeployer reads XMl documents containing MBean descriptors from files whose name has the form "*-service.xml". When it sees a descriptor for a `Connector` MBean, it passes the descriptor's `<config>` element to a newly created `Connector`.

There are two ways in which to specify the server invoker configuration via a service xml file. The first is to specify just the InvokerLocator attribute as a sub-element of the Connector MBean. For example, a possible configuration for a Connector using a socket invoker that is listening on port 8084 on the test.somedomain.com address would be:

```
            <mbean code="org.jboss.remoting.transport.Connector"
```

```
            name="jboss.remoting:service=Connector,transport=Socket"
            display-name="Socket transport Connector">
         <attribute name="InvokerLocator">
            <![CDATA[socket://test.somedomain.com:8084]]>
         </attribute>
         <attribute name="Configuration">
            <config>
               <handlers>
                  <handler subsystem="mock">
                     org.jboss.remoting.transport.mock.MockServerInvocationHandler
                  </handler>
               </handlers>
            </config>
         </attribute>
      </mbean>
```

Note that all the server side socket invoker configurations will be set to their default values in this case. Also, it is important to add CDATA to any locator uri that contains more than one parameter.

The other way to configure the Connector and its server invoker in greater detail is to provide an `invoker` sub-element within the config element of the Configuration attribute. The only attribute of invoker element is transport, which will specify which transport type to use (e.g.. socket, rmi, http, or multiplex). All the sub-elements of the invoker element will be attribute elements with a name attribute specifying the configuration property name and then the value. An `isParam` attribute can also be added to indicate that the attribute should be added to the locator uri, in the case the attribute needs to be used by the client. An example using this form of configuration is as follows:

```
      <mbean code="org.jboss.remoting.transport.Connector"
            name="jboss.remoting:service=Connector,transport=Socket"
            display-name="Socket transport Connector">

         <attribute name="Configuration">
            <config>

               <invoker transport="socket">
                  <attribute name="numAcceptThreads">1</attribute>
                  <attribute name="maxPoolSize">303</attribute>
                  <attribute name="clientMaxPoolSize" isParam="true">304</attribute>
                  <attribute name="socketTimeout">60000</attribute>
                  <attribute name="serverBindAddress">192.168.0.82</attribute>
                  <attribute name="serverBindPort">6666</attribute>
                  <attribute name="clientConnectAddress">216.23.33.2</attribute>
                  <attribute name="clientConnectPort">7777</attribute>
                  <attribute name="enableTcpNoDelay" isParam="true">false</attribute>
                  <attribute name="backlog">200</attribute>
               </invoker>

               <handlers>
                  <handler subsystem="mock">
                     org.jboss.remoting.transport.mock.MockServerInvocationHandler
                  </handler>
               </handlers>
            </config>
         </attribute>

      </mbean>
```

Also note that `${jboss.bind.address}` can be used for any of the bind address properties, which will be replaced with the bind address specified to JBoss when starting (i.e. via the -b option).

All the attributes set in this configuration could be set directly in the locator uri of the InvokerLocator attribute value, but would be much more difficult to decipher visually and is more prone to editing mistakes.

One of the components of a locator uri that can be expressed within the InvokerLocator attribute is the path. For example, can express a locator uri path of 'foo/bar' via the InvokerLocator attribute as:

```
<attribute name="InvokerLocator"><![CDATA[socket://test.somedomain.com:8084/foo/bar]]></attri
```

To include the path using the Configuration attribute, can include a specific 'path' attribute. So the same InvokerLocator can be expressed as follows with the Configuration attribute:

```
<attribute name="Configuration">
   <config>
     <invoker transport="socket">
       <attribute name="serverBindAddress">test.somedomain.com</attribute>
       <attribute name="serverBindPort">8084</attribute>
       <attribute name="path">foo/bar</attribute>
     </invoker>
     ...
```

Note: The value for the 'path' attribute should NOT start or end with a / (slash).

5.1.1.3. Callback client configuration

Remoting supports asynchronous computation and delivery of results through a callback mechanism, as described in Section Callbacks. Callbacks are sent from the server side to the client side on a callback connection which is the reverse of the usual client to server connection. That is, a client invoker on the server side communicates with a server invoker on the client side (in the case of push callbacks - again, see Section Callbacks). When a callback connection is created, all of the configuration information passed to the server side `Connector` is passed on to the server side callback client invoker. It follows that callback client invokers are configured by way of the server side `Connector`.

5.1.2. Client side configuration

Invoker configuration on the client side parallels configuration on the server side, with the exception that (1) it operates in a simpler environment (in particular, it does not assume the presence of an MBeanServer) and (2) it does not support a declarative option. However, it does support versions of the first three server side programmatic options, with the `Client` class playing the central role played by the `Connector` class on the server side.

Again, the most straightforward form of configuration is to put the configuration parameters on the `InvokerLocator`. For example, the fragment

```
String locatorURI = "socket://test.somedomain.com:8084";
String params = "/?clientMaxPoolSize=10&timeout=360000";
locatorURI += params;
InvokerLocator locator = new InvokerLocator(locatorURI);
Client client = new Client(locator);
client.connect();
```

creates a `Client` using the socket transport to connect to a server on host test.somedomain.com, listening on port 8084. It also passes in two parameters, "clientMaxPoolSize" and "timeout", that will be used by the client invoker.

It is also possible to use configuration maps on the client side. The following code fragment accomplishes all that the previous fragment does, but it passes one parameter by way of the `InvokerLocator` and passes the other by way of a configuration map. It also passes in a non-string object, a `SocketFactory`:

```
        String locatorURI = "socket://test.somedomain.com:8084";
        String params = "/?clientMaxPoolSize=10";
        locatorURI += params;
        InvokerLocator locator = new InvokerLocator(locatorURI);
        HashMap config = new HashMap();
        config.put(ServerInvoker.TIMEOUT, 360000);
        config.put(Remoting.CUSTOM_SOCKET_FACTORY, new MySocketFactory());
        Client client = new Client(locator, config);
        client.connect();
```

Note that the value of `ServerInvoker.TIMEOUT` is "timeout", and the value of `Remoting.CUSTOM_SOCKET_FACTORY` is "customSocketFactory". These configuration map keys are discussed throughout the chapter and accumulated in section Configuration by properties. Also, socket factory configuration is covered in Socket factories and server socket factories.

Finally, a third programmatic option is available for those configuration properties which happen to be client invoker MBean properties. In the following fragment, the client invoker is obtained from the `Client` and a `SocketFactory` is passed to it by way of a setter method:

```
        String locatorURI = "socket://test.somedomain.com:8084";
        String params = "/?clientMaxPoolSize=10";
        locatorURI += params;
        InvokerLocator locator = new InvokerLocator(locatorURI);
        HashMap config = new HashMap();
        config.put(ServerInvoker.TIMEOUT, 360000);
        Client client = new Client(locator, config);
        client.connect();
        SocketFactory sf = new MySocketFactory();
        ClientInvoker clientInvoker = client.getInvoker();
        clientInvoker.setSocketFactory(sf);
```

Note. The `Client` creates the client invoker during the call to `Client.connect()`, so this option only works after that method has been called.

5.2. Handlers

Handlers are classes that the invocation is given to on the server side (the final target for remoting invocations). To implement a handler, all that is needed is to implement the `org.jboss.remoting.ServerInvocationHandler` interface. There are a two ways in which to register a handler with a Connector. The first is to do it programmatically. The second is via service configuration. For registering programmatically, can either pass the ServerInvocationHandler reference itself or an ObjectName for the ServerInvocationHandler (in the case that it is an MBean). To pass the handler reference directly, call `Connector::addInvocationHandler(String subsystem, ServerInvocationHandler handler)`. For example (from `org.jboss.remoting.samples.simple.SimpleServer`):

```
InvokerLocator locator = new InvokerLocator(locatorURI);
Connector connector = new Connector();
connector.setInvokerLocator(locator.getLocatorURI());
connector.create();

SampleInvocationHandler invocationHandler = new SampleInvocationHandler();
// first parameter is sub-system name. can be any String value.
connector.addInvocationHandler("sample", invocationHandler);

connector.start();
```

To pass the handler by ObjectName, call `Connector::addInvocationHandler(String subsystem, ObjectName handlerObjectName)` . For example (from `org.jboss.test.remoting.handler.mbean.ServerTest`):

```
MBeanServer server = MBeanServerFactory.createMBeanServer();
InvokerLocator locator = new InvokerLocator(locatorURI);
Connector connector = new Connector();
connector.setInvokerLocator(locator.getLocatorURI());
connector.start();

server.registerMBean(connector, new ObjectName("test:type=connector,transport=socket"));

// now create Mbean handler and register with mbean server
MBeanHandler handler = new MBeanHandler();
ObjectName objName = new ObjectName("test:type=handler");
server.registerMBean(handler, objName);

connector.addInvocationHandler("test", objName);
```

Is important to note that if not starting the Connector via the service configuration, will need to explicitly register it with the MBeanServer (will throw exception otherwise).

If using a service configuration for starting the Connector and registering handlers, can either specify the fully qualified class name for the handler, which will instantiate the handler instance upon startup (which requires there be a void parameter constructor), such as:

```
<handlers>
  <handler subsystem="mock">
    org.jboss.remoting.transport.mock.MockServerInvocationHandler
  </handler>
</handlers>
```

where MockServerInvocationHandler will be constructed upon startup and registered with the Connector as a handler.

Can also use an ObjectName to specify the handler. The configuration is the same, but instead of specifying a fully qualified class name, you specify the ObjectName for the handler, such as (can see `mbeanhandler-service.xml` under remoting tests for full example):

```
<handlers>
  <handler subsystem="mock">test:type=handler</handler>
</handlers>
```

The only requirement for this configuration is that the handler MBean must already be created and registered with the MBeanServer at the point the Connector is started.

Handler implementations

The Connectors will maintain a reference to the handler instances provided (either indirectly via the MBean proxy or directly via the instance object reference). For each request to the server invoker, the handler will be called upon. Since the server invokers can be multi-threaded (and in most cases would be), this means that the handler may receive concurrent calls to handle invocations. Therefore, handler implementations should take care to be thread safe in their implementations.

Stream handler

There is also an invocation handler interface that extends the ServerInvocationHandler interface specifically for handling of input streams as well as normal invocations. See the section on sending streams for further details. As for Connector configuration, it is the same.

HTTP handlers

Since there is extra information needed when dealing with the http transport, such as headers and response codes, special consideration is needed by handlers. The handlers receiving http invocations can get and set this extra information via the InvocationRequest that is passed to the handler.

Server invoker for the http transport will add the following to the InvocationRequest's request payload map:

MethodType - the http request type (i.e., GET, POST, PUT, HEADER, OPTIONS). Can use the contant value HTTPMetadataConstants.METHODTYPE, if don't want to use the actual string 'MethodType' as the key to the request payload map.

Path - the url path. Can use the contant value HTTPMetadataConstants.PATH, if don't want to use the actual string 'Path' as the key to the request payload map.

HttpVersion - the client's http version. Can use the contant value HTTPMetadataConstants.HTTPVERSION, if don't want to use the actual string 'HttpVersion' as the key to the request payload map.

Other properties from the original http request will also be included in the request payload map, such as request headers. Can reference org.jboss.test.remoting.transport.http.method.MethodInvocationHandler as an example for pulling request properties from the InvocationRequest.

The only time this will not be added is a POST request where an InvocationRequest is passed and is not binary content type (application/octet-stream).

The handlers receiving http invocations can also set the response code, response message, response headers, and content-type. To do this, will need to get the return payload map from the InvocationRequest passed (via its getReturnPayload() method). Then populate this map with whatever properties needed. For response code and message, will need to use the following keys for the map:

ResponseCode - Can use the constant value HTTPMetaDataConstants.RESPONSE_CODE, if don't want to use the actual string 'ResponseCode' as they key. **IMPORTANT** - The value put into map for this key MUST be of type java.lang.Integer.

ResponseCodeMessage - Can use the constant value HTTPMetadataConstants.RESPONSE_CODE_MESSAGE, if don't want to use the actual string 'ResponseCodeMessage' as the key. The value put into map for this key should be of type java.lang.String.

Is also important to note that ALL http requests will be passed to the handler. So even OPTIONS, HEAD, and PUT method requests will need to be handled. So, for example, if want to accept OPTIONS method requests, would need to populate response map with key of 'Allow' and value of 'OPTIONS, POST, GET, HEAD, PUT', in order to tell calling client that all these method types are allowed. Can see an example of how to do this within org.jboss.test.remoting.transport.http.method.MethodInvocationHandler.

The PUT request will be handled the same as a POST method request and the PUT request payload will be included within the InvocationRequest passed to the server handler. It is up to the server handler to set the proper resonse code (or throw proper exception) for the processing of the PUT request. See http://www.ietf.org/rfc/rfc2616.txt?number=2616 [http://www.ietf.org/rfc/rfc2616.txt?number=2616], section 9.6 for details on response codes and error responses).

HTTP Client

The HttpClientInvoker will now put the return from HttpURLConnection getHeaderFields() method into the metadata map passed to the Client's invoke() method (if not null). This means that if the caller passes a non-null Map, it can then get the response headers. It is important to note that each response header field key in the metadata map is associated with a list of response header values, so to get a value, would need code similar to:

```
Object response = remotingClient.invoke((Object) null, metadata);
String allowValue = (String) ((List) metadata.get("Allow").get(0);
```

Can reference org.jboss.test.remoting.transport.http.method.HTTPInvokerTestClient for an example of this.

Note that when making a http request using the OPTIONS method type, the return from the Client's invoke() method will ALWAYS be null.

Also, if the response code is 400, the response returned will be that of the error stream and not the standard input stream. So is important to check for the response code.

Two values that will always be set within the metadata map passed to the Client's invoke() method (when not null), is the response code and response message from the server. These can be found using the keys:

ResponseCode - Can use the constant value HTTPMetaDataConstants.RESPONSE_CODE, if don't want to use the actual string 'ResponseCode' as the key. **IMPORTANT** - The value returned for this key will be of type java.lang.Integer.

ResponseCodeMessage - Can use the constant value from HTTPMetadataConstants.RESPONSE_CODE_MESSAGE, if don't want to use the actual string 'ResponseCodeMessage' as the key. The value returned for this key will be of type java.lang.String.

An example of getting the response code can be found within

org.jboss.test.remoting.transport.http.method.HTTPInvokerTestClient.

5.3. Discovery (Detectors)

Domains

Detectors have the ability to accept multiple domains. What domains that the detector will accept as viewable can either be set programmatically via the method:

```
public void setConfiguration(org.w3c.dom.Element xml)
```

or by adding to jboss-service.xml configuration for the detector. The domains that the detector is currently accepting can be retrieved from the method:

```
public org.w3c.dom.Element getConfiguration()
```

The configuration xml is a MBean attribute of the detector, so can be set or retrieved via JMX.

There are three possible options for setting up the domains that a detector will accept. The first is to not call the `setConfiguration()` method (or just not add the configuration attribute to the service xml). This will cause the detector to use only its domain and is the default behavior. This enables it to be backwards compatible with earlier versions of JBoss Remoting (JBoss 4, DR2 and before).

The second is to call the `setConfiguration()` method (or add the configuration attribute to the service xml) with the following xml element:

```
<domains>
  <domain>domain1</domain>
  <domain>domain2</domain>
</domains>
```

where `domain1` and `domain2` are the two domains you would like the detector to accept. This will cause the detector to accept detections only from the domains specified, and no others.

The third and final option is to call the setConfiguration() method (or add the configuration attribute to the service xml) with the following xml element:

```
<domains>
</domains>
```

This will cause the detector to accept all detections from any domain.

By default, remoting detection will ignore any detection message the it receives from a server invoker running within its own jvm. To disable this, add an element called 'local' to the detector configuration (alongside the domain element) to indicate should accept detection messages from local server invokers. This will be false by default, so maintains the same behavior as previous releases. For example:

```
        <domains>
            <domain>domain1</domain>
            <domain>domain2</domain>
        </domains>
        <local/>
```

An example entry of a Multicast detector in the jboss-service.xml that accepts detections only from the roxanne and sparky domains using port 5555, including servers in the same jvm, is as follows:

```
        <mbean code="org.jboss.remoting.detection.multicast.MulticastDetector"
              name="jboss.remoting:service=Detector,transport=multicast">
           <attribute name="Port">5555</attribute>
           <attribute name="Configuration">
             <domains>
               <domain>roxanne</domain>
               <domain>sparky</domain>
             </domains>
             <local/>
           </attribute>
        </mbean>
```

Global Detector Configuration

The following are configuration attributes for all the remoting detectors.

DefaultTimeDelay - amount of time, in milliseconds, which can elapse without receiving a detection event before suspecting that a server is dead and performing an explicit invocation on it to verify it is alive. If this invocation, or ping, fails, the server will be removed from the network registry. The default is 5000 milliseconds.

HeartbeatTimeDelay - amount of time to wait between sending (and sometimes receiving) detection messages. The default is 1000 milliseconds.

JNDIDetector

Port - port to which detector will connect for the JNDI server.

Host - host to which the detector will connect for the JNDI server.

ContextFactory - context factory string used when connecting to the JNDI server. The default is `org.jnp.interfaces.NamingContextFactory`.

URLPackage - url package string to use when connecting to the JNDI server. The default is `org.jboss.naming:org.jnp.interfaces`.

CleanDetectionNumber - Sets the number of detection iterations before manually pinging remote server to make sure still alive. This is needed since remote server could crash and yet still have an entry in the JNDI server, thus making it appear that it is still there. The default value is 5.

Can either set these programmatically using setter method or as attribute within the remoting-service.xml (or anywhere else the service is defined). For example:

```
```

```
        <mbean code="org.jboss.remoting.detection.jndi.JNDIDetector"
               name="jboss.remoting:service=Detector,transport=jndi">
           <attribute name="Host">localhost</attribute>
           <attribute name="Port">5555</attribute>
        </mbean>
```

If the JNDIDetector is started without the Host attribute being set, it will try to start a local JNP instance (the JBoss JNDI server implementation) on port 1088.

MulticastDetector

DefaultIP - The IP that is used to broadcast detection messages on via multicast. To be more specific, will be the ip of the multicast group the detector will join. This attribute is ignored if the Address has already been set when started. Default is 224.1.9.1.

Port - The port that is used to broadcast detection messages on via multicast. Default is 2410.

BindAddress - The address to bind to for the network interface.

Address - The IP of the multicast group that the detector will join. The default will be that of the DefaultIP if not explicitly set.

BufferSize - The size of the buffer used by the MulticastSocket. The default is 10000.

If any of these are set programmatically, need to be done before the detector is started (otherwise will use default values).

5.4. Transports (Invokers)

This section covers configuration issues for each of the transports, beginning with a set of properties that apply to all transports. The material in a later section in this chapter, Socket factories and server socket factories, also applies to all transports.

5.4.1. Features introduced in Remoting version 2.2

Subsequent to the release of Remoting 2.2.0.GA, some transport independent features have been introduced.

5.4.1.1. Binding to 0.0.0.0

Before release 2.2.2.SP2, a Remoting server could bind to only one specific IP address. In particular, the address 0.0.0.0 was translated to the host returned by `java.net.InetAddress.getLocalHost()` (or its equivalent IP address). As of release 2.2.2.SP2, a server started with the address 0.0.0.0 binds to all available interfaces.

Note. If 0.0.0.0 appears in the `InvokerLocator`, it needs to be translated to an address that is usable on the client side. If the system property `InvokerLocator.BIND_BY_HOST` (actual value "remoting.bind_by_host") is set to "true", the `InvokerLocator` host will be transformed to the value returned by `InetAddress.getLocalHost().getHostName()`. Otherwise, it will be transformed to the value returned by `InetAddress.getLocalHost().getHostAddress()`.

5.4.1.2. Support for IPv6 addresses

As of release 2.2.2.SP4, `org.jboss.remoting.InvokerLocator` will accept IPv6 IP addresses. For example,

```
socket://[::1]:3333/?timeout=10000
socket://[::]:4444/?timeout=10000
socket://[::ffff:127.0.0.1]:5555/?timeout=10000
socket://[fe80::205:9aff:fe3c:7800%7]:6666/?timeout=10000
```

5.4.2. Server Invokers

The following configuration properties are common to all the current server invokers.

serverBindAddress - The address on which the server binds to listen for requests. The default is an empty value which indicates the server should be bound to the host provided by the locator url, or if this value is null, the local host as provided by `InetAddress.getLocalHost()`.

serverBindPort - The port to listen for requests on. A value of 0 or less indicates that a free anonymous port should be chosen.

maxNumThreadsOneway - specifies the maximum number of threads to be used within the thread pool for accepting one way invocations on the server side. This property will only be used in the case that the default thread pool is used. If a custom thread pool is set, this property will have no meaning. This property can also be retrieved or set programmatically via the `MaxNumberOfOnewayThreads` property.

onewayThreadPool - specifies either the fully qualified class name for a class that implements the `org.jboss.util.threadpool.ThreadPool` interface or the JMX ObjectName for an MBean that implements the `org.jboss.util.threadpool.ThreadPool` interface. This will replace the default `org.jboss.util.threadpool.BasicThreadPool` used by the server invoker.

Note that this value will NOT be retrieved until the first one-way (server side) invocation is made. So if the configuration is invalid, will not be detected until this first call is made. The thread pool can also be accessed or set via the `OnewayThreadPool` property programmatically.

Important to note that the default thread pool used for the one-way invocations on the server side will block the calling thread if all the threads in the pool are in use until one is released.

5.4.3. Configurations affecting the invoker client

There are some configurations which will impact the invoker client. These will be communicated to the client invoker via parameters in the Locator URI. These configurations can not be changed during runtime, so can only be set up upon initial configuration of the server invoker on the server side. The following is a list of these and their effects.

clientConnectPort - the port the client will use to connect to the remoting server. This would be needed in the case that the client will be going through a router that forwards requests made externally to a different port internally.

clientConnectAddress - the ip or hostname the client will use to connect to the remoting server. This would be

needed in the case that the client will be going through a router that forwards requests made externally to a different ip or host internally.

If no client connect address or server bind address specified, will use the local host's address (via `InetAddress.getLocalHost().getHostAddress()`).

Note The role played by "clientConnectAddress" and "clientConnectPort" deserves some further elaboration. When a server is set up, it is either given an `InvokerLocator` explicitly, or it is given enough information in an MBean XML file from which to construct an `InvokerLocator`, and a client uses the host field and port field in the `InvokerLocator` to determine how to connect to the server. It follows that if an explicit `InvokerLocator` is passed to the server, then the host and port fields are either given explicitly or are generated, so there is no need for "clientConnectAddress" or "clientConnectPort" fields. However, if the server is configured by way of an MBean XML file, and no explicit `InvokerLocator` is specified, then the "clientConnectAddress" and "clientConnectPort" parameters can be used to specify the host and port fields in the `InvokerLocator`. If they are omitted, then the host and port fields will be derived from the values of the "serverBindAddress" and "serverBindPort" parameters (or generated, if those fields are omitted). Therefore, there is a role for the "clientConnectAddress" and "clientConnectPort" parameters **only if** clients are meant to connect to a host and port different than the bind host and bind port. Such a situation might occur in the presence of a translating firewall between the client and the server.

5.4.4. How the server bind address and port is determined

If the serverBindAddress property is set, the server invoker will bind to that address. Otherwise, it will, with one exception, use the address in the InvokerLocator (if there is one). The exception is the case in which the clientConnectAddress property is set, which indicates that the adddess in the InvokerLocator is not the real address of the server's host. In that case, and in the case that there is no address in the InvokerLocator, the server will bind to the address of the local host, as determined by the call

```
        InetAddress.getLocalHost().getHostAddress();
```

In other words, the logic is

```
        if (serverBindAddress is set)
           use it
        else if (the host is present in the InvokerLocator and clientConnectAddress is not set)
           use host from InvokerLocator
        else
           use local host address
```

If the serverBindPort property is set, it will be used. If this value is 0 or a negative number, then the next available port will be found and used. If the serverBindPort property is not set, but the clientConnectPort property is set, then the next available port will be found and used. If neither the serverBindPort nor the clientConnectPort is set, then the port specified in the original InvokerLocator will be used. If this is 0 or a negative number, then the next available port will be found and used. In the case that the next available port is used because either the serverBindPort or the original InvokerLocator port value was either 0 or negative, the InvokerLocator will be updated to reflect the new port value.

Note. In the case that a bind port isn't specified, the utility class `org.jboss.remoting.transport.PortUtil` is used to supply an available port. By default, it will look for a port in the range 1024 to 65535, inclusively. As of re-

lease 2.2.3.SP1, PortUtil can be configured to search a smaller range by setting the values PortUtil.MIN_PORT (actual value "minPort") and / or PortUtil.MAX_PORT (actual value "maxPort") in the InvokerLocator, a configuration map, an MBean XML file. The range is static; that is, whenever "minPort" or "maxPort" are set, they affect all subsequent calls in the JVM. Note that `PortUtil` will apply a new "minPort" value only if it is greater than the current value, and it will apply a new "maxPort" value only if it is less than the current value. And it will never apply a new value when the result would be such that the value of "maxPort" would be less than the value of "minPort".

5.4.5. Socket Invoker

The following configuration properties can be set at any time, but will not take effect until the socket invoker, on the server side, is stopped and restarted.

timeout - The socket timeout value passed to the Socket.setSoTimeout() method. The default on the server side is 60000 (one minute). If the timeout parameter is set, its value will also be passed to the client side (see below).

backlog - The preferred number of unaccepted incoming connections allowed at a given time. The actual number may be greater than the specified backlog. When the queue is full, further connection requests are rejected. Must be a positive value greater than 0. If the value passed if equal or less than 0, then the default value will be assumed. The default value is 200.

numAcceptThreads - The number of threads that exist for accepting client connections. The default is 1.

maxPoolSize - The number of server threads for processing client. The default is 300.

serverSocketClass - specifies the fully qualified class name for the custom SocketWrapper implementation to use on the server.

socket.check_connection - indicates if the invoker should try to check the connection before re-using it by sending a single byte ping from the client to the server and then back from the server. This config needs to be set on both client and server to work. This if false by default.

idleTimeout - indicates the number of seconds a pooled server thread can be idle (meaning time since last invocations request processed) before it should be cleaned up and removed from the thread pool. The value for this property must be greater than zero in order to enable idle timeouts on pooled server threads (otherwise they will not be checked). Setting to value less than zero will disable idle timeout checks on pooled server threads, in the case was previously enabled. The default value for this property is -1.

continueAfterTimeout - indicates what a server thread should do after experiencing a `java.net.SocketTimeoutException`. If set to "true", or if JBossSerialization is being used, the server thread will continue to wait for an invocation; otherwise, it will return itself to the thread pool.

Configurations affecting the Socket invoker client

There are some configurations which will impact the socket invoker client. These will be communicated to the client invoker via parameters in the Locator URI. These configurations can not be changed during runtime, so can only be set up upon initial configuration of the socket invoker on the server side. The following is a list of these and their effects.

enableTcpNoDelay - can be either true or false and will indicate if client socket should have TCP_NODELAY turned on or off. TCP_NODELAY is for a specific purpose; to disable the Nagle buffering algorithm. It should

only be set for applications that send frequent small bursts of information without getting an immediate response; where timely delivery of data is required (the canonical example is mouse movements). The default is false.

timeout - The socket timeout value passed to the Socket.setSoTimeout() method. The default on the client side is 1800000 (or 30 minutes).

clientMaxPoolSize - the client side maximum number of active socket connections. This basically equates to the maximum number of concurrent client calls that can be made from the socket client invoker. The default is 50.

numberOfCallRetries - number of retries for making invocation. It is possible that an existing socket connection timed out while waiting within the pool. Since not doing a connection check by default, will throw away the connection and try to get a new one. Will do this for whatever the numberOfCallRetries (which defaults to 3) is. However, when reaches numberOfCallsRetries - 2, will flush the entire connection pool under the assumption that all connections in the pool have timed out and are invalid and will start over by creating a new connection. If still fails, will throw MarshalException with the cause being the original SocketException.

clientSocketClass - specifies the fully qualified class name for the custom SocketWrapper implementation to use on the client. Note, will need to make sure this is marked as a client parameter (using the 'isParam' attribute). Making this change will not affect the marshaller/unmarshaller that is used, which may also be a requirement.

socket.check_connection - indicates if the invoker should try to check the connection before re-using it by sending a single byte ping from the client to the server and then back from the server. This config needs to be set on both client and server to work. This if false by default.

An example of locator uri for a socket invoker that has TCP_NODELAY set to false and the client's max pool size of 30 would be:

```
socket://test.somedomain.com:8084/?enableTcpNoDelay=false&maxPoolSize=30
```

To reiterate, these client configurations can only be set within the server side configuration and will not change during runtime.

5.4.5.1. How the Socket Invoker works

The Socket Invoker is one of the more complicated invokers mainly because allows the highest degree of configuration. To better understand how changes to configuration properties for the Socket invoker (both client and server) will impact performance and scalability, will discuss the implementation and how it works in detail.

server

When the socket server invoker is started, it will create one, and only one, instance of java.net.ServerSocket. Upon being started, it will also create and start a number of threads to be used for accepting incoming requests from the ServerSocket. These threads are called the accept threads and the number of them created is controlled by the 'numAcceptThreads' property. When these accept threads are started, they will call accept() on the ServerSocket and block until the ServerSocket receives a request from a client, where it will return a Socket back to the accept thread who called the accept() method. As soon as this happens, the accept thread will try to pass off the Socket to another thread for processing.

The threads that actually process the incoming request, referred to as server threads, are stored in a pool. The accept thread will try to retreive the first available server thread from the pool and hand off the Socket for processing. If the pool does not contain any available server threads and the max pool size has not been reached, a new server

thread will be created for processing. Otherwise, if the max pool size has been reached, the accept thread will wait for one to become available (will wait until socket timeout has been reached). The size of the server thread pool is defined by the 'maxPoolSize' property. As soon as the accept thread has been able to hand off the Socket to a server thread for processing, it will loop back to ServerSocket and call accept() on it again. This will continue until the socket server invoker is stopped.

The server thread processing the request will be the thread of execution through the unmarshalling of the data, calling on the server invocation handler, and marshalling of response back to the client. After the response has been sent, the server thread will then hold the socket connection and wait for another request to come from this client. It will wait until the socket is closed by the client, a socket timeout occurs, or receives another request from the client in which to process. When the client socket connection session is closed, meaning timeout or client closed socket connection, then the thread will return itself to the pool.

If all the server threads from the pool are in use, meaning have a client connection established, and the pool has reached its maximum value, the accept threads (no matter how many there are) will have to wait until one of the server threads is available for processing. This why having a large number of accept threads does not provide any real benefit. If all the accept threads are blocked waiting for server thread, new client requests will then be queued until it can be accepted. The number of requests that can be queued is controlled by the backlog and can be useful in managing sudden bursts in requests.

If take an example with a socket server invoker that has max pool set to 300, accept threads is 2, and backlog is 200, will be able to make 502 concurrent client calls. The 503rd client request will get an exception immediately. However, this does not mean all 502 requests will be guaranteed to be processed, only the first 300 (as they have server threads available to do the processing). If 202 of the server threads finish processing their requests from their initial client connections and the connection is released before the timeout for the other 202 that are waiting (200 for backlog and 2 for accept thread), then they will be processed (of course this is a request by request determination).

As of JBossRemoting 2.2.0 release, can also add configuration for cleaning up idle server threads using the 'idleTimeout' configuration property. Setting this property to a value of greater than zero will activate idle timeout checking, which is disabled by default. When enabled, the idle timeout checker will periodically iterate through the server threads that are active and inactive and if have not processed a request within the designated idle timeout period, the server thread will be shutdown and removed from corresponding pool. Active server threads are ones that have a socket connection associated with it and are in a blocked read waiting for data from the client. Inactive server threads are ones that have finished processing on a particular socket connection and have been returned to the thread pool for later reuse.

Note. Prior to release 2.2.3.SP1, a bug allowed the termination of server threads in the middle of a long invocation. If the idle timeout facility is used, upgrading is strongly recommended. Note that even with the bug fix, a server thread that is engaged in a long invocation when the idle timeout checker is activated can be removed from its thread pool and marked for later destruction once the invocation is complete. The "idleTimeout" value should be configured accordingly.

Note. Prior to release 2.2.2.SP7, if a server thread experienced a `java.net.SocketTimeoutException`, it would return itself to the thread pool and could not be reused until a new socket connection was created for it to use. In principle, it would be more efficient for the server thread simply to try again to read the next invocation, and, in release 2.2.2.SP7, that is what it does. Unfortunately, `java.io.ObjectInputStream` ceases to function once it experiences a `SocketTimeoutException`. The good news is that `org.jboss.serial.io.JBossObjectInputStream`, made available by the JBossSerialization project, does not suffer from that problem. Therefore, as of release 2.2.2.SP8, when

it experiences a `SocketTimeoutException`, a server thread will check whether it is using a `JBossObjectInputStream` or not and act accordingly. Just to allow for the possibility that an application is using yet another version of `ObjectInputStream`, the parameter `ServerThread.CONTINUE_AFTER_TIMEOUT` (actual value "continueAfterTimeout") allows the behavior following a `SocketTimeoutException` to be configured explicitly.

client

When the socket client invoker makes its first invocation, it will check to see if there is an available socket connection in its pool. Since is the first invocation, there will not be and will create a new socket connection and use it for making the invocation. Then when finished making invocation, will return the still active socket connection to the pool. As more client invocations are made, is possible for the number of socket connections to reach the maximum allowed (which is controlled by 'clientMaxPoolSize' property). At this point, when the next client invocation is made, it will wait up to 30 seconds for an existing connection to be returned to the pool. If it doesn't get a connection within 30 seconds, it will throw an IllegalStateException.

Once the socket client invoker goes and get an available socket connection from the pool, are not out of the woods yet. There is still a possibility that the socket connection returned, while still appearing to be valid, has timed out while sitting in the pool. So if discover this while trying to make invocation, will throw it away and retry the whole process again. Will do this up to the number set by the numberOfCallRetries before throwing an exception. The trick here is that when get to numberOfCallRetries -2, will assume that any socket connection gotten from the pool will have timed out and will flush the pool all together so that the next retry will cause a new socket connection to be recreated. A typical scenario when this might occur is when have had a burst of client invocations and then a long period of inactivity.

Note. As of release 2.2.2.GA, the server side of the socket transport can capture the IP address of the client side of a TCP connection from client to server and make it available to application code on the client side. The address can be retrieved as follows:

```
Client client = new Client(locator);
...
InvocationResponse response = (InvocationResponse) client.invoke("$GET_CLIENT_LOCAL_ADDRESS$");
InetAddress address = (InetAddress) response.getResult();
```

5.4.6. SSL Socket Invoker

Supports all the configuration attributes as the Socket Invoker. The main difference is that the SSL Socket Invoker uses an `SSLServerSocket` by default, created by an `SSLServerSocketFactory`. See section Socket factories and server socket factories for more information.

5.4.7. RMI Invoker

registryPort - the port on which to create the RMI registry. The default is 3455. This also needs to have the isParam attribute set to true.

Note. The RMI server invoker creates a socket factory and passes it to a client invoker along with the RMI stub, so the socket factory must be serializable. Therefore, if a socket factory is passed in to the server invoker by one of the methods discussed in section Socket factories and server socket factories, then the user is responsible for supplying a serializable socket factory.

5.4.8. SSL RMI Invoker

This is essentially identical to the RMI invoker, except that it creates SSL socket and server socket factories by default.

Note. The SSL RMI server invoker creates a socket factory and passes it to a client invoker along with the RMI stub, so the socket factory must be serializable. If the SSL RMI server invoker is allowed to create an `SSLSocketFactory` from SSL parameters, as discussed in section Socket factories and server socket factories, it will take care to create a serializable socket factory. However, if a socket factory is passed in to the server invoker (also discussed in section Socket factories and server socket factories), then the user is responsible for supplying a serializable socket factory. See sslrmi below for more information.

5.4.9. HTTP Invoker

The HTTP server invoker implementation is based on the Apache Tomcat connector components which support GET, POST, HEAD, OPTIONS, and HEAD method types and keep-alive. Therefore, most any configuration allowed for Tomcat can be configured for the remoting HTTP server invoker. For more information on the configuration attributes available for the Tomcat connectors, please refer to http://tomcat.apache.org/tomcat-5.5-doc/config/http.html. http://tomcat.apache.org/tomcat-5.5-doc/config/http.html So for example, if wanted to set the maximum number of threads to be used to accept incoming http requests, would use the 'maxThreads' attribute. The only exception when should use remoting configuration over the Tomcat configuration is for attribute 'address' (use serverBindAddress instead) and attribute 'port' (use serverBindPort instead).

Note: The http invoker no longer has the configuration attributes 'maxNumThreadsHTTP' or 'HTTPThreadPool' as thread pooling is now handled within the Tomcat connectors, which does not expose external API for setting these.

Since the remoting HTTP server invoker implementation is using Tomcat connectors, is possible to swap out the Tomcat protocol implementations being used. By default, the protocol being used is `org.apache.coyote.http11.Http11Protocol`. However, it is possible to switch to use the `org.apache.coyote.http11.Http11AprProtocol` protocol, which is based on the Apache Portable Runtime (see http://tomcat.apache.org/tomcat-5.5-doc/apr.html and http://apr.apache.org/ for more details). If want to use the APR implementation, simply put the tcnative-1.dll (or tcnative-1.so) on the system path so can be loaded. The APR native binaries can be found at http://tomcat.heanet.ie.

Note: There is a bug with release 1.1.1 of APR where get an error upon shutting down (see JBREM-277 for more information). This does not impact anything while running, but is still an issue when shutting down (as upon starting up again, can get major problems). This should be fixed in a later release of APR so can just replace the 1.1.1 version of tcnative-1.dll with the new one.

Client request headers

The HTTP Invoker allows for some of the properties to be passed as request headers from client caller. The following are possible http headers and what they mean:

sessionId - is the remoting session id to identify the client caller. If this is not passed, the HTTP server invoker will try to create a session id based on information that is passed. Note, this means if the sessionId is not passed as part of the header, there is no guarantee that the sessionId supplied to the invocation handler will always indicate the request from the same client.

subsystem - the subsystem to call upon (which invoker handler to call upon). If there is more than one handler per Connector, this will need to be set (otherwise will just use the only one available).

These request headers are set automatically when using a remoting client, but if using another client to send request to the HTTP server invoker, may want to add these headers.

Response headers

If a request on the HTTP transport is made with the `org.jboss.remoting.Client` method

```
public Object invoke(Object param, Map metadata) throws Throwable
```

then `org.jboss.remoting.transport.http.HTTPClientInvoker` returns the HTTP response headers in a map in metadata, associated with the key `org.jboss.remoting.transport.http.HTTPMetadataConstants.RESPONSE_HEADERS` (actual value "Response-Headers"). For example, the response header "Date" can be retrieved as follows:

```
Object payload = ... ;
HashMap metadata = new HashMap();
client.invoke(payload, metadata);
Map responseHeaders = (Map) metadata.get(HTTPMetadataConstants.RESPONSE_HEADERS);
String date = (String) responseHeaders.get("Date");
```

CR/LF in HTTP transport

By default, the HTTP transport uses `org.jboss.remoting.marshal.http.HTTPMarshaller` and `org.jboss.remoting.marshal.http.HTTPUnMarshaller` to marshal and unmarshal invocations and responses. Prior to Remoting version 2.2.3, `HTTPUnMarshaller` stripped CR/LF characters. As of version 2.2.3, the default behavior remains the same, but it is possible to change the behavior, on the client and the server, by setting the parameter `HTTPUnMarshaller.PRESERVE_LINES` (actual value "preserveLines") to "true".

5.4.10. HTTPS Invoker

Supports all the configuration attributes as the HTTP Invoker, plus the following:

SSLImplementation - Sets the Tomcat SSLImplementation to use. This should always be `org.jboss.remoting.transport.coyote.ssl.RemotingSSLImplementation`.

The main difference with the HTTP invoker is that the HTTPS Invoker uses an `SSLServerSocket` by default, created by an `SSLServerSocketFactory`. See section Socket factories and server socket factories for more information.

5.4.11. HTTP(S) Client Invoker - proxy and basic authentication

This section covers configuration specific to the HTTP Client Invoker only and is NOT related to HTTP(S) invoker configuration on the server side (via service xml).

proxy

There are a few ways in which to enable http proxy using the HTTP client invoker. The first is simply to add the following properties to the metadata Map passed on the Client's invoke() method: `http.proxyHost` and `http.proxyPort`.

An example would be:

```
Map metadata = new HashMap();
...

// proxy info
metadata.put("http.proxyHost", "ginger");
metadata.put("http.proxyPort", "80");

...

response = client.invoke(payload, metadata);
```

The http.proxyPort property is not required and if not present, will use default of 80. Note: setting the proxy config in this way can ONLY be done if using JDK 1.5 or higher.

The other way to enable use of an http proxy server from the HTTP client invoker is to set the following system properties (either via `System.setProperty()` method call or via JVM arguments): `http.proxyHost`, `http.proxyPort`, and `proxySet`.

An example would be setting the following JVM arguments:

```
-Dhttp.proxyHost=ginger -Dhttp.proxyPort=80 -DproxySet=true
```

Note: when testing with Apache 2.0.48 (mod_proxy and mod_proxy_http), all of the properties above were required.

Setting the system properties can be used for JDK 1.4 and higher. However, will not be able to specify proxy server per remoting client if use system properties..

Basic authentication - direct and via proxy

The HTTP client invoker also has support for BASIC authentication for both proxied and non-proxied invocations. For proxied invocations, the following properties need to be set: `http.proxy.username` and `http.proxy.password`.

For non-proxied invocations, the following properties need to be set: `http.basic.username` and `http.basic.password`.

For setting either proxied or non-proxied properties, can be done via the metadata map or system properties (see setting proxy properties above for how to). However, for authentication properties, values set in the metadata Map will take precedence over those set within the system properties.

Note: Only the proxy authentication has been tested using Apache 2.0.48; non-proxied authentication has not.

Since there are many different ways to do proxies and authentication in this great world of web, not all possible configurations have been tested (or even supported). If you find a particular problem or see that a particular implementation is not supported, please enter an issue in Jira (http://jira.jboss.com) under the JBossRemoting project, as

this is where bugs and feature requests belong. If after reading the documentation have unanswered questions about how to use these features, please post them to the remoting forum (http://www.jboss.org/index.html?module=bb&op=viewforum&f=222 [http://www.jboss.org/index.html?module=bb&op=viewforum&f=222]).

Host name verification

During the SSL handshake when making client calls using https transport, if the URL's hostname and the server's identification hostname mismatch, a javax.net.ssl.HostnameVerifier implementation will be called to determine if this connection should be allowed. The default implementation will not allow this, but it is possible to override the default behavior

One option is to use the key `HTTPSClientInvoker.HOSTNAME_VERIFIER` (actual value "hostnameVerifier") to supply the name of a class that implements the `javax.net.ssl.HostnameVerifier` interface, passing it either in the metadata map supplied with an invocation or in the configuration map supplied when the `HTTPSClientInvoker` was created. If the key appears in both maps, the value in the metadata map takes precedence.

In the absence of an explicitly declared `HostnameVerifier`, another way to configure the hostname verification behavior is to declare that all host names are acceptable, which can be accomplished by setting the `HTTPSClientInvoker.IGNORE_HTTPS_HOST` property (actual value "org.jboss.security.ignoreHttpsHost") to true. In order of increasing precedence, the property may be set (1) as a system property, (2) in the configuration map supplied when the `HTTPSClientInvoker` was created, or in the metadata map supplied with an invocation.

Finally, in the absence of both an explicitly declared `HostnameVerifier` and an explicit directive to ignore host names, an `HTTPSClientInvoker` will check to see if its `SocketFactory` is an instance of `org.jboss.remoting.security.CustomSSLSocketFactory` and, if so, if authentication has been turned off. If that is the case, host names will be ignored. See Section Socket factories and server socket factories for more information about `SocketFactory` configuration.

5.4.12. Servlet Invoker

The servlet invoker is a server invoker implementation that uses a servlet running within a web container to accept initial client invocation requests. The servlet request is then passed on to the servlet invoker for processing.

The deployment for this particular server invoker is a little different than the other server invokers since a web deployment is also required. To start, the servlet invoker will need to be configured and deployed. This can be done by adding the Connector MBean service to an existing service xml or creating a new one. The following is an example of how to declare a Connector that uses the servlet invoker:

```
<mbean code="org.jboss.remoting.transport.Connector"
       name="jboss.remoting:service=Connector,transport=Servlet"
       display-name="Servlet transport Connector">

   <attribute name="InvokerLocator">
      servlet://localhost:8080/servlet-invoker/ServerInvokerServlet
   </attribute>

   <attribute name="Configuration">
      <config>
         <handlers>
            <handler subsystem="test">
               org.jboss.test.remoting.transport.web.WebInvocationHandler
            </handler>
```

```
                  </handlers>
               </config>
            </attribute>
         </mbean>
```

An important point of configuration to note is that the value for the InvokerLocator attribute is the exact url used to access the servlet for the servlet invoker (more on how to define this below), with the exception of the protocol being servlet instead of http. This is important if using automatic discovery, as this is the locator url that will be discovered and used by clients to connect to this server invoker.

The next step is to configure and deploy the servlet that fronts the servlet invoker. The pre-built deployment file for this servlet is the servlet-invoker.war file (which can be found in lib directory of the release distribution or under the output/lib/ directory if doing a source build). By default, it is actually an exploded war, so the servlet-invoker.war is actually a directory so that can be more easily configured (feel free to zip up into an actual war file if prefer). In the WEB-INF directory is located the web.xml file. This is a standard web configuration file and should look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
    <servlet>
        <servlet-name>ServerInvokerServlet</servlet-name>
        <description>The ServerInvokerServlet receives requests via HTTP
           protocol from within a web container and passes it onto the
           ServletServerInvoker for processing.
        </description>
        <servlet-class>org.jboss.remoting.transport.servlet.web.ServerInvokerServlet</servlet-class>
        <init-param>
            <param-name>invokerName</param-name>
            <param-value>jboss.remoting:service=invoker,transport=servlet</param-value>
            <description>The servlet server invoker</description>
           <!--
            <param-name>locatorUrl</param-name>
            <param-value>servlet://localhost:8080/servlet-invoker/ServerInvokerServlet</param-value>
            <description>The servlet server invoker locator url</description>
           -->
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>ServerInvokerServlet</servlet-name>
        <url-pattern>/ServerInvokerServlet/*</url-pattern>
    </servlet-mapping>
</web-app>
```

There are two ways in which the servlet can obtain a reference to the servlet server invoker it needs to pass its request onto. The first is by using the param 'invokerName', as is shown above. The value for this should be the JMX ObjectName for the servlet server invoker that was deployed as a service mbean (see service xml above). The other way is to provide a param 'locatorUrl' with a value that matches the locator url of the servlet server invoker to use. In this case, will use the InvokerRegistry to find the server invoker instead of using JMX, which is useful if not deploying server invoker as a mbean service or if want to run in web container other than the JBoss application server. Note, one or the other param is required. If both are provided, the 'locatorUrl' param take precedence.

This file can be changed to meet any web requirements you might have, such as adding security (see sslservlet) or

changing the actual url context that the servlet maps to. If the url that the servlet maps to is changed, will need to change the value for the InvokerLocator in the Connector configuration mentioned above.

Note. Prior to Remoting version 2.2.3, `org.jboss.remoting.transport.servlet.ServletServerInvoker` generated a single MBean ObjectName for representing `ServletServerInvokers` as MBeans, which meant that an MBeanServer could be aware of only a single `ServletServerInvoker` MBean. As of version 2.2.3, that restriction can be eliminated by setting the parameter `org.jboss.remoting.transport.servlet.ServletServerInvoker.CREATE_UNIQUE_OBJECT_NAME` (actual value "createUniqueObjectName") to "true".

5.4.12.1. Configuration

unwrapSingletonArrays - If the map returned by `javax.servlet.http.HttpServletRequest.getParameterMap()` maps a String key to an array of length one, the value in the array will be extracted and associated with the key.

createUniqueObjectName - If set to "true", each `InvokerLocator` will be transformed into a unique ObjectName.

Issues

One of the issues of using Servlet invoker is that the invocation handlers (those that implement ServerInvocationHandler) can not return very much detail in regards to a web context. For example, the content type used for the response is the same as that of the request.

5.4.13. SSL Servlet Invoker

The SSL Servlet Invoker is exactly the same as its parent, Servlet Invoker, with the exception that it uses the protocol of 'sslservlet'. On the server side it is deployed exactly the same as a servlet invoker would be but requires setting up ssl within the web container (i.e. enabling the ssl connector within Tomcat's server.xml). This will usually require specifing a different port as well.

An example of the mbean service xml for deploying the ssl servlet server invoker would be:

```
<?xml version="1.0" encoding="UTF-8"?>

<server>
  <mbean code="org.jboss.remoting.transport.Connector"
         name="jboss.remoting:service=Connector,transport=SSLServlet"
         display-name="SSL Servlet transport Connector">

             <attribute name="InvokerLocator">
                sslservlet://localhost:8443/servlet-invoker/ServerInvokerServlet
             </attribute>
             <attribute name="Configuration">
                <config>
                   <handlers>
                      <handler subsystem="test">org.jboss.test.remoting.transport.web.WebInvocationHandl
                   </handlers>
                </config>
             </attribute>
  </mbean>
</server>
```

An example of servlet-invoker.war/WEB-INF/web.xml for the ssl server invoker servlet would be:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
    <servlet>
        <servlet-name>ServerInvokerServlet</servlet-name>
        <description>The ServerInvokerServlet receives requests via HTTP
           protocol from within a web container and passes it onto the
           ServletServerInvoker for processing.
        </description>
        <servlet-class>org.jboss.remoting.transport.servlet.web.ServerInvokerServlet</servlet-class>
        <init-param>
            <param-name>locatorUrl</param-name>
            <param-value>sslservlet://localhost:8443/servlet-invoker/ServerInvokerServlet</param-value>
            <description>The servlet server invoker locator url</description>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
   <servlet-mapping>
        <servlet-name>ServerInvokerServlet</servlet-name>
        <url-pattern>/ServerInvokerServlet/*</url-pattern>
    </servlet-mapping>
</web-app>
```

5.4.14. Exception handling for web based clients

Web based clients, meaning remoting clients that call on web based remoting servers (i.e. http, https, servlet, and sslservlet) have special needs when it comes to handling exceptions that come from the servers they are calling on. The main reason for this is that depending on what type of server they are calling on, they might receive the error in different formats.

By default, web based clients will throw an exception when the response code from the server is greater than 400. The exact exception type thrown will depend on the type of web server the client is interacting with. If it is a JBoss Remoting server (http or https server invoker), the exception thrown will be the one originally generated on the server side. If the server is not a JBoss Remoting server (e.g. JBossAS, Tomcat, Apache Web Server, etc.), the exception throw will be `org.jboss.test.remoting.transport.http.WebServerError`. The WebServerError's message will be the error html returned by the web server. To turn off the throwing of an exception when the web server responds with an error, can add config to the configuration map passed to the `Client.invoke()` method where they key is `HTTPMetadataConstants.NO_THROW_ON_ERROR` (actual text value 'NoThrowOnError') and a value of of type java.lang.String set to 'true'. This will cause the http client invoker to not throw an exception, but instead return the data from the web server error stream. In the case that the data returned from this error stream is of type java.lang.String (i.e. is error html), it will be wrapped in a WebServerError and returned as this type. The raw data from the web server can the be retrieved by getting the WebServerError's message.

Note. Prior to Remoting release 2.2.2.SP2, the servlet transport returned a simple error message in the event of an error on the server side. As of release 2.2.2.SP2, the exception handling behavior described above can be requested for the the servlet and sslservlet transports as well by configuring the server with the parameter `org.jboss.remoting.transport.http.HTTPMetadataConstants.RETURN_EXCEPTION` (actual value "return-exception") set to "true".

5.4.15. Multiplex Invoker

The multiplex invoker is intended to replicate the functionality of the socket invoker with the added feature that it supports multiple streams of communication over a single pair of sockets. Multiplexing may be motivated by, for example, a desire to conserve socket resources or by firewall restrictions on port availability. This additional service is made possible by the Multiplex subproject, which provides "virtual" sockets and "virtual" server sockets. Please refer to the Multiplex documentation at

http://labs.jboss.com/portal/jbossremoting/docs/index.html
[http://labs.jboss.com/portal/jbossremoting/docs/index.html]

for further details.

In a typical multiplexed scenario a `Client` on a client host, through a `MultiplexClientInvoker` *C*, could make synchronous method invocations to a `MultiplexServerInvoker` on a server host, and at the same time (and over the same TCP connection) asynchronous push callbacks could be made to a `MultiplexServerInvoker` *S* on the client host. In this, the **Prime Scenario**, which motivated the creation of the multiplex invoker, *C* and *S* use two different virtual sockets but share the same port and same actual socket. We say that *C* and *S* belong to the same **invoker group**.

One of the primary design goals of the Multiplex subsystem is for virtual sockets and virtual server sockets to demonstrate behavior as close as possible to their real counterparts, and, indeed, they implement complete socket and server socket APIs. However, they are necessarily different in some respects, and it follows that the multiplex invoker is somewhat different than the socket invoker. In particular, there are three areas specific to the multiplex invoker that must be understood in order to use it effectively:

1. Establishing on the server an environment prerequisite for creating multiplex connections
2. Configuring the client for multiplexed method invocations and callbacks
3. Shutting down invoker groups.

5.4.15.1. Setting up the server

There are two kinds of `MultiplexServerInvokers`, **master** and **virtual**, corresponding to the two kinds of virtual server sockets: `MasterServerSocket` and `VirtualServerSocket`. Briefly, the difference between the two virtual server socket classes is that a `MasterServerSocket` is derived from `java.net.ServerSocket` and its `accept()` method is implemented by way of the inherited method `super.accept()`. A `MasterServerSocket` can accept connect requests from multiple machines. A `VirtualServerSocket`, on the other hand, is based on an actual socket connected to another actual socket on some host *H*, and consequently a `VirtualServerSocket` can accept connect requests only from *H*.

Each multiplex connection depends on a pair of connected real sockets, one on the client host and one on the server host, and this connection is created when an actual socket contacts an actual server socket. It follows that a multiplex connection begins with a connection request to a `MasterServerSocket`. Once the connection is established, it is possible to build up **virtual socket groups**, consisting of virtual sockets (and at most one `VirtualServerSocket`) revolving around the actual socket at each end of the connection. Each virtual socket in a socket group at one end is connected to a virtual socket in the socket group at the other end.

Master and virtual MultiplexServerInvokers assume the characteristics of their server sockets: MasterServerSocket and VirtualServerSocket, respectively. That is, a master MultiplexServerInvoker can accept requests from any host, while a virtual MultiplexServerInvoker can accept requests only from the particular host to which it has a multiplex connection. Since a multiplex connection begins with a connection request to a MasterServerSocket, it follows that the use of the multiplex invoker must begin with a connection request from the client (made by either a MultiplexClientInvoker or a virtual MultiplexServerInvoker: see below) to a master MultiplexServerInvoker on the server. The master MultiplexServerInvoker responds by "cloning" itself (metaphorically, not necessarily through the use of clone()) into a virtual MultiplexServerInvoker with the same parameters and same set of invocation handlers but with a VirtualServerSocket belonging to a new socket group. In so doing the master MultiplexServerInvoker builds up a **server invoker farm** of virtual MultiplexServerInvokers, each in contact with a different MultiplexClientInvoker over a distinct multiplex connection. The virtual MultiplexServerInvokers do the actual work of responding to method invocation requests, sent by their corresponding MultiplexClientInvokers through virtual sockets in a socket group at the client end of a multiplex connection to virtual sockets created by the VirtualServerSocket in the socket group at the server end of the connection. Note that virtual MultiplexServerInvokers share data structures with the master, so that registering invocation handlers with the master makes them available to the members of the farm. The members of a master MultiplexServerInvoker's invoker farm are accessible by way of the methods

1. MultiplexServerInvoker.getServerInvokers() and
2. MultiplexServerInvoker.getServerInvoker(InetSocketAddress)

the latter of which returns a virtual MultiplexServerInvoker keyed on the address to which its VirtualServerSocket is connected. When the master MultiplexServerInvoker shuts down, its farm of virtual invokers shuts down as well

There are two ways of constructing a virtual MultiplexServerInvoker, one being the cloning method just discussed. It is also possible to construct one directly. Once a multiplex connection is established, a virtual MultiplexServerInvoker can be created with a VirtualServerSocket belonging to a socket group at one end of the connection. The MultiplexServerInvoker constructor determines whether to create a virtual or master invoker according to the presence or absence of certain parameters, discussed below, that may be added to its InvokerLocator. Server rules 1 through 3 described below result in the construction of a virtual MultiplexServerInvoker, and server rule 4 (the absence of these parameters) results in the construction of a master MultiplexServerInvoker.

Setting up the server, then, is simply a matter of starting a master MultiplexServerInvoker with a simple InvokerLocator, unadorned with any parameters specific to the multiplex invoker. As always, the server invoker is not created directly but by way of a Connector, as in the following:

```
Connector connector = new Connector();
Connector.setInvokerLocator("multiplex://demo.jboss.com:8080");
Connector.create()
Connector.start()
```

5.4.15.2. Setting up the client

Before multiplex connections can be established, a master `MultiplexServerInvoker` must be created as described in the previous section. For example, the Prime Scenario would begin with starting a master `MultiplexServerInvoker` on the server host, followed by starting, on the client host, a `MultiplexClientInvoker` *C* and a virtual `MultiplexServerInvoker` *S* (in either order). The first to start initiates a multiplex connection to the master `MultiplexServerInvoker` and requests the creation of a virtual `MultiplexServerInvoker`. Note that it is crucial for *C* and *S* to know that they are meant to share a multiplex connection, i.e., that they are meant to belong to the same invoker group. Consider the following attempt to set up a shared connection between hosts bluemonkey.acme.com and demo.jboss.com. First, *C* is initialized on host bluemonkey.acme.com with the `InvokerLocator` multiplex://demo.jboss.com:8080, and, assuming the absence of an existing multiplex connection to demo.jboss.com:8080, a new virtual socket group based on a real socket bound to an arbitrary port, say 32000, is created. Then *S* is initialized with `InvokerLocator` multiplex://bluemonkey.acme.com:4444, but since it needs to bind to port 4444, it is unable to share the existing connection. [Actually, the example is slightly deceptive, since multiplex://bluemonkey.acme.com:4040 would result in the creation of a master `MultiplexServerInvoker`. But if it were suitably extended with the parameters discussed below so that a virtual `MultiplexServerInvoker` were created, the virtual invoker would be unable to share the existing connection.]

So *C* and *S* need to agree on the address and port of the real socket underlying the virtual socket group they are intended to share on the client host and the address and port of the real socket underlying the peer virtual socket group on the server host. Or, more succintly, they must know that they are meant to belong to the same invoker group. Note the relationship between an invoker group and the virtual socket group which supports it: a `MultiplexClientInvoker` uses virtual sockets in its underlying virtual socket group, and a `MultiplexServerInvoker` in an invoker group has a `VirtualServerSocket` that creates virtual sockets in the underlying virtual socket group.

C and *S* each get half of the information necessary to identify their invoker group directly from their respective `InvokerLocators`. In particular, *C* gets the remote address and port, and *S* gets the binding address and port. The additional information may be provided through the use of **invoker group parameters**, which may be communicated to *C* and *S* in one of two ways:

1. they may be appended to the `InvokerLocator` passed to the `Client` which creates *C* and/or to the `InvokerLocator` passed to the `Connector` which creates *S*
2. they may be stored in a configuration `Map` which is passed to the `Client` and/or `Connector`.

In either case, there are two ways in which the missing information can be supplied to *C* and *S*:

1. The information can be provided explicitly by way of invoker group parameters:
    a. *multiplexBindHost* and *multiplexBindPort* parameters can be passed to *C*, and
    b. *multiplexConnectHost* and *multiplexConnectPort* parameters can be passed to *S*.
2. *C* and *S* can be tied together by giving them the same **multiplexId**, supplied by invoker group parameters:
    a. *clientMultiplexId*, for the `MultiplexClientInvoker`, and
    b. *serverMultiplexId*, for the `MultiplexServerInvoker`.

    Giving them matching multiplexIds tells them that they are meant to belong to the same invoker group and

that they should provide the missing information to each other.

The behavior of a starting `MultiplexClientInvoker` *C* is governed by the following four **client rules**:

1. If *C* has a *clientMultiplexId* parameter, it will use it to attempt to find a `MultiplexServerInvoker` *S* with a *serverMultiplexId* parameter with the same value. If it succeeds, it will retrieve binding host and port values, create or reuse a suitable multiplex connection to the server, and start. Moreover, if *S* was unable to start because of insufficient information (server rule 3), then *C* will supply the missing information and *S* will start. Note that in this situation *C* will ignore any *multiplexBindHost* and *multiplexBindPort* parameters passed to it.

2. If *C* does not find a `MultiplexServerInvoker` through a multiplexId (either because it did not get a *clientMultiplexId* parameter or because there is no `MultiplexServerInvoker` with a matching multiplexId), but it does have *multiplexBindHost* and *multiplexBindPort* parameters, then it will create or reuse a suitable multiplex connection to the server, and start. Also, if it has a multiplexId, it will advertise itself for the benefit of a `MultiplexServerInvoker` that may come along later (see server rule 1).

3. If *C* has a multiplexId and neither finds a `MultiplexServerInvoker` with a matching multiplexId nor has *multiplexBindHost* and *multiplexBindPort* parameters, then it will not start, but it will advertise itself so that it may be found later by a `MultiplexServerInvoker` (see server rule 1).

4. If *C* has neither *clientMultiplexId* nor *multiplexBindHost* and *multiplexBindPort* parameters, it will create or reuse a multiplex connection from an arbitrary local port to the host and port given in its `InvokerLocator`, and start.

Similarly, the behavior of a starting `MultiplexServerInvoker` *S* is governed by the following four **server rules**:

1. If *S* has a *serverMultiplexId* parameter, it will use it to attempt to find a `MultiplexClientInvoker` *C* with a matching *clientMultiplexId*. If it succeeds, it will retrieve server host and port values, create a `VirtualServerSocket`, create or reuse a suitable multiplex connection to the server, and start. Moreover, if *C* was unable to start due to insufficient information (client rule 3), then *S* will supply the missing information and *C* will start. Note that in this situation *S* will ignore *multiplexConnectHost* and *multiplexConnectPort* parameters, if any, in its `InvokerLocator`.

2. If *S* does not find a `MultiplexClientInvoker` through a multiplexId (either because it did not get a *serverMultiplexId* parameter or because there is no `MultiplexClientInvoker` with a matching multiplexId), but it does have *multiplexConnectHost* and *multiplexConnectPort* parameters, then it will create a `VirtualServerSocket`, create or reuse a suitable multiplex connection to the server, and start. Also, if it has a multiplexId, it will advertise itself for the benefit of a `MultiplexClientInvoker` that may come along later (see client rule 1).

3. If *S* has a multiplexId and neither finds a `MultiplexClientInvoker` with a matching multiplexId nor has *multiplexConnectHost* and *multiplexConnectPort* parameters, then it will not start, but it will advertise itself so that it may be found later by a `MultiplexClientInvoker` (see client rule 1).

4. If *S* has neither *serverMultiplexId* nor *multiplexConnectHost* and *multiplexConnectPort* parameters, it will create a `MasterServerSocket` bound to the host and port in its `InvokerLocator` and start.

5.4.15.2.1. Notes

1. Like server invokers, client invokers are not started directly but are started indirectly through calls to `Client(InvokerLocator locator)`, such as:

```
Client client = new Client("multiplex://demo.jboss.com:8080/?clientMultiplexId=id0");
client.connect();
```

   N.B. For the multiplex invoker, it is important to call `Client.connect()`. Otherwise, the last `MultiplexClientInvoker` that leaves an invoker group will not get a chance to shut the group down.

2. It should not be inferred that `MultiplexClientInvokers` and `MultiplexServerInvokers` belong to the same invoker group *only if* they are required to do so by invoker group parameters. In fact, if two `Clients` are created with the `InvokerLocator` multiplex://demo.jboss.com, the second one, lacking any constraints on its binding address and port, is certainly not prevented from sharing a connection with the first. Rather, the function of the invoker group parameters is to *force* `MultiplexClientInvokers` and `MultiplexServerInvokers` to share a connection.

3. There are situations in which the method of passing parameters by way of the configuration map is preferable to appending them to an `InvokerLocator`. One of the functions of an `InvokerLocator` is to identify a server, and modifying the content of its `InvokerLocator` may interfere with the ability to locate the server. For example, one of the features of JBoss Remoting is the substitution of method calls for remote invocations when it discovers that a server runs in the same JVM as the client. However, appending multiplex parameters to the `InvokerLocator` by which the server is identified will prevent the Remoting runtime from recognizing the local presence of the server, and the optimization will not occur.

4. It is possible, and convenient, to set up a multiplexing scenario using no parameters other than *clientMultiplexId* and *serverMultiplexId*. Note, however, that in this case neither the `Clients` nor the `Connector` will be fully initialized until after both have been started. If the `Clients` and the `Connector` are to be started independently, then the other parameters must be used. **N.B.** If a `Client` depends on `Connector` in the same invoker group to supply binding information, it is an error to call methods such as `Client.connect()` and `Client.invoke()` until the `Connector` has been started.

5. `Clients` and the optional `Connector` may be created (and the `Connector` started) in any order.

5.4.15.3. Shutting down invoker groups.

A virtual socket group will shut down, releasing a real socket and a number of threads, when (1) its last member has closed and (2) the socket group at the remote end of the multiplex connection agrees to the proposed shut down. The second condition prevents a situation in which a new virtual socket tries to join what it thinks is a viable socket group at the same time that the peer socket group is shutting down. So for a virtual socket group to shut down, all members at both ends of the connection must be closed.

The implication of this negotiated shutdown mechanism is that as long as the `VirtualServerSocket` used by a virtual `MultiplexServerInvoker` remains open, resources at the client end of the connection cannot be freed, and for this reason it is important to understand how to close virtual `MultiplexServerInvokers`.

There are three ways in which a virtual `MultiplexServerInvoker` that belongs to a master `MultiplexServerInvoker`'s invoker farm can shut down.

- When a master `MultiplexServerInvoker` is closed, it closes all of the virtual `MultiplexServerInvokers` it created.
- A virtual `MultiplexServerInvoker` can be retrieved by calling either `MultiplexServerInvoker.getServerInvokers()` or `MultiplexServerInvoker.getServerInvoker(InetSocketAddress)` on its master `MultiplexServerInvoker` and then closed directly.
- When the `accept()` method of its `VirtualServerSocket` times out, and when it detects that all multiplex invokers in the invoker group at the client end of the connection have shut down, a virtual `MultiplexServerInvoker` will shut itself down. Note that when all members leave an invoker group, it is guaranteed not to be revived, i.e., no new members may join.

The third method insures that without any explicit intervention, closing all multiplex invokers on the client (by way of calling `Client.disconnect()` and `Connector.stop()`) is guaranteed to result in the eventual release of resources. The timeout period may be adjusted by setting the *timeout* parameter (see below). Alternatively, the second method, in conjunction with the use of `MultiplexServerInvoker.isSafeToShutdown()`, which returns `true` on `MultiplexServerInvoker M` if and only if (1) `M` is not virtual, or (2) all of the multiplex invokers in the invoker group at the client end of `M`'s connection have shut down. For example, a thread could be dedicated to looking for useless `MultiplexServerInvokers` and terminating them before their natural expiration through timing out.

5.4.15.4. Examples

The following are examples of setting up a client for multiplexed synchronous and asynchronous communication. They each assume the existence of a master `MultiplexServerInvoker` running on demo.jboss.com:8080.

For complete examples see the section Multiplex Invoker.

1. A `MultiplexClientInvoker` *C* starts first:

```
String parameters = "multiplexBindHost=localhost&multiplexBindPort=7070&clientMultiplexId=dem
String locatorURI = "multiplex://demo.jboss.com:8080/?" + parameters;
InvokerLocator locator = new InvokerLocator(locatorURI);
Client client = new Client(locator);
client.connect();
```

   and then it is found by a `MultiplexServerInvoker` with a matching multiplexId, which joins *C*'s invoker group and starts:

```
Connector connector = new Connector();
String parameters = "serverMultiplexId=demoId1";
String locatorURI = "multiplex://localhost:7070/?" + parameters;
InvokerLocator locator = new InvokerLocator(locatorURI);
connector.setInvokerLocator(locator.getLocatorURI());
connector.create();
connector.start();
```

2. A `MultiplexClientInvoker` *C* starts:

```
String parameters = "multiplexBindHost=localhost&multiplexBindPort=7070";
String locatorURI = "multiplex://demo.jboss.com:8080/?" + parameters;
```

```
InvokerLocator locator = new InvokerLocator(locatorURI);
Client client = new Client(locator);
client.connect();
```

and a `MultiplexServerInvoker` *S* starts independently, joining *C*'s invoker group by virtue of having matching local and remote addresses and ports:

```
Connector connector = new Connector();
String parameters = "multiplexConnectHost=demo.jboss.com&multiplexConnectPort=8080";
String locatorURI = "multiplex://localhost:7070/?" + parameters;
InvokerLocator locator = new InvokerLocator(locatorURI);
connector.setInvokerLocator(locator.getLocatorURI());
connector.create();
connector.start();
```

3. A `MultiplexClientInvoker` *C* is created but does not start:

```
String parameters = "clientMultiplexId=demoId2";
String locatorURI = "multiplex://demo.jboss.com:8080/?" + parameters;
InvokerLocator locator = new InvokerLocator(locatorURI);
Client client = new Client(locator);
```

and then a `MultiplexServerInvoker` *S* is created with a matching multiplexId, allowing both *C* and *S* to start:

```
Connector connector = new Connector();
String parameters = "serverMultiplexId=demoId2";
String locatorURI = "multiplex://localhost:7070/?" + parameters;
InvokerLocator locator = new InvokerLocator(locatorURI);
connector.setInvokerLocator(locator.getLocatorURI());
connector.create();
connector.start();
client.connect();
```

Note the call to `Client.connect()` after the call to `Connector.start()`.

4. A `MultiplexClientInvoker` *C* starts in an invoker group based on a real socket bound to an arbitrary local port:

```
String locatorURI = "multiplex://demo.jboss.com:8080";
InvokerLocator locator = new InvokerLocator(locatorURI);
Client client = new Client(locator);
client.connect();
```

and then a `MultiplexServerInvoker` *S* starts independently:

```
Connector connector = new Connector();
String locatorURI = "multiplex://localhost:7070";
InvokerLocator locator = new InvokerLocator(locatorURI);
```

```
connector.setInvokerLocator(locator.getLocatorURI());
connector.create();
connector.start();
```

Note that *S* creates a `MasterServerSocket` rather than a `VirtualServerSocket` in this case and so does not share a multiplex connection and does not belong to an invoker group.

5. This is example 1, rewritten so that the invoker group parameters are passed by way of a configuration `Map` instead of `InvokerLocators`. A `MultiplexClientInvoker` *C* starts first:

```
String locatorURI = "multiplex://demo.jboss.com:8080";
InvokerLocator locator = new InvokerLocator(locatorURI);
Map configuration = new HashMap();
configuration.put(MultiplexInvokerConstants.MULTIPLEX_BIND_HOST_KEY, "localhost");
configuration.put(MultiplexInvokerConstants.MULTIPLEX_BIND_PORT_KEY, "7070");
configuration.put(MultiplexInvokerConstants.CLIENT_MULTIPLEX_ID_KEY, "demoId1");
Client client = new Client(locator, configuration);
client.connect();
```

and then it is found by a `MultiplexServerInvoker` with a matching multiplexId, which joins *C*'s invoker group and starts:

```
String locatorURI = "multiplex://localhost:7070";
InvokerLocator locator = new InvokerLocator(locatorURI);
Map configuration = new HashMap();
configuration.put(MultiplexInvokerConstants.SERVER_MULTIPLEX_ID_KEY, "demoId1");
Connector connector = new Connector(locator.getLocatorURI(), configuration);
connector.create();
connector.start();
```

5.4.15.5. Configuration properties

There are four categories of configuration properties supported by the multiplex invoker, the last three of which are specific to the multiplex invoker. Properties in categories 2 and 3 may be configured by appending them to the server's locator URI. Properties in categories 2, 3, and 4 may be configured by putting their values in a configuration `HashMap` and passing the map to a `MultiplexServerInvoker` and/or `MultiplexClientInvoker` constructor, according to the category. Constants for the property names in categories 2, 3, and 4 are found in `org.jboss.remoting.transport.multiplex.Multiplex`. Note that some of them are also found in the older `org.jboss.remoting.transport.multiplex.MultiplexInvokerConstants`, but the use of that class is now deprecated.

1. The following properties are managed by ancestor classes of `MultiplexServerInvoker`. See the discussion under `SocketServerInvoker` for more information.

   socketTimeout - The socket timeout value passed to the `Socket.setSoTimeout()` method and the `ServerSocket.setSoTimeout()` method. The default is 60000 (or 1 minute).

   numAcceptThreads - The number of threads that exist for accepting client connections. The default is 1.

2. The following properties are intended to be passed to a virtual `MultiplexServerInvoker` to configure its multiplex connection. These properties are specific to the multiplex invoker.

   multiplexConnectHost - the name or address of the host to which the multiplex connection should be made.

   multiplexConnectPort - the port to which the multiplex connection should be made.

   serverMultiplexId - a string that associates a `MultiplexServerInvoker` with a `MultiplexClientInvoker` with which it should share a multiplex connection.

   multiplex.maxAcceptErrors - Master and virtual `MultiplexServerInvokers` keep a counter of errors experienced by their server socket, and they terminate when this maximum is exceeded. Note that `SSLHandshakeExceptions` are excluded from the count, since they could indicate a client rather than server error.

3. The following properties are intended to be passed to a virtual `MultiplexClientInvoker` to configure its multiplex connection. These properties are specific to the multiplex invoker.

   multiplexBindHost - the host name or address to which the local end of the multiplex connection should be bound.

   multiplexBindPort - the port to which the local end of the multiplex connection should be bound

   clientMultiplexId - a string that associates a `MultiplexClientInvoker` with a `MultiplexServerInvoker` with which it should share a multiplex connection.

4. There is also a set of properties which are specific to the Multiplex subsystem internal classes. See the Multiplex documentation at

   http://labs.jboss.com/portal/jbossremoting/docs/index.html
   [http://labs.jboss.com/portal/jbossremoting/docs/index.html]

   for more information.

5.4.16. SSL Multiplex Invoker

This transport is essentially identical to the Multiplex transport, except that it will create SSL socket factories and server socket factories by default.

The twist to be found with the multiplex transport is that virtual `MultiplexServerInvokers` use a `VirtualServerSocket`, which is based on a client rather than a server socket, and consequently they act like a client in some ways. In particular, a virtual `MultiplexServerInvoker` will, in some cases, attempt to connect to a remote master `MultiplexServerInvoker`, for which it will need an actual client socket. All of the rules for configuring socket factories apply to the `MultiplexServerInvoker`, which calls the same method that client invokers use to get a socket factory. Moreover, if necessary, it will look for a `ServerSocketFactoryMBean` to get SSL information when configuring a socket factory. See section Socket factories and server socket factories for more information.

5.4.17. Bisocket invoker

The **bisocket transport**, like the multiplex transport, is a bidirectional transport that can function in the presence of restrictions that would prevent a unidirectional transport like socket or http from creating a server to client push callback connection. (See Section Callbacks for more information about callbacks and bidirectional and unidirectional transports.) For example, security restrictions could prevent the application from opening a `ServerSocket` on the client, or firewall restrictions could prevent the server from contacting a `ServerSocket` even if it were possible to create one.

5.4.17.1. Overview

The bisocket client and server invokers inherit most of their functionality from the socket invokers, with the principal exception of overriding a method in the client invoker called `createSocket()`. If the client invoker is on the client side, then `createSocket()` simply calls the super implementation. The heart of the bisocket transport is in handling the case of creating a connection from a callback client invoker on the server side to a callback server invoker on the client side, which is mandated to occur without the use of a `ServerSocket` on the client side. Whenever the bisocket transport is informed by an application of its intention to use push callbacks, the client side creates a secondary "control" connection, and subsequently, whenever the callback client invoker needs to create a connection to the callback server, it sends a request over the control connection asking the client side to establish the connection. The server side of the transport maintains a secondary `ServerSocket` that accepts connection requests from the client side, and whenever a socket is created it is passed to whichever callback client invoker requested it. The client invoker, which inherits the socket transport's connection pool management facility, adds the new socket to its connection pool.

Note that if the control connection were to fail, no new connections could be created for the callback client invoker, and eventually callback transmission could come to a halt. The client and server invokers work together, therefore, to maintain a heartbeat on the control connection and to recreate the control connection automatically should it fail. In particular, the server side sends out ping messages on the control connection, and the client side needs to receive a ping message within some configured window in order to consider the connection to be functional.

In addition to the configuration options inherited from the socket transport, the bisocket transport may be configured with the following parameters, which are defined as constants in the `org.jboss.remoting.transport.bisocket.Bisocket` class. A parameter can be configured on the server side by appending it to the `InvokerLocator` or by adding it to the configuration map passed to the `Connector`'s constructor. On the client side, where all parameters are used by the callback server invoker, there are several options for setting parameter values. If the callback `Connector` is created explicitly, then a parameter can be configured by appending it to the callback `Connector`'s `InvokerLocator` or by adding it to the configuration map passed to the callback `Connector`'s constructor. If the callback `Connector` is created implicitly by the `Client.addListener()` method, then its configuration map is the union of the `Client`'s configuration map and the `metadata` map passed as a parameter to `Client.addListener()`.

IS_CALLBACK_SERVER (actual value is "isCallbackServer"): when a bisocket server invoker receives this parameter with a value of true, it avoids the creation of a `ServerSocket`. Therefore, IS_CALLBACK_SERVER should be used on the client side for the creation of a callback server. The default value is false.

PING_FREQUENCY (actual value is "pingFrequency"): The server side uses this value to determine the interval, in milliseconds, between pings that it will send on the control connection. The client side uses this value to calculate the window in which it must receive pings on the control connection. In particular, the window is ping frequency * ping window factor. See also the definition of PING_WINDOW_FACTOR. The default value is 5000.

PING_WINDOW_FACTOR (actual value is "pingWindowFactor"): The client side uses this value to calculate

the window in which it must receive pings on the control connection. In particular, the window is ping frequency * ping window factor. See also the definition of PING_FREQUENCY. The default value is 2.

MAX_RETRIES (actual value is "maxRetries"): This parameter is relevant only on the client side, where the `BisocketClientInvoker` uses it to govern the number of attempts it should make to get the address and port of the secondary `ServerSocket`, and the `BisocketServerInvoker` uses it to govern the number of attempts it should make to create both ordinary and control sockets. The default value is 10.

5.4.17.2. Details

Using the bisocket transport certainly does not require understanding its implementation details, but some further information is presented in this section for those who might be interested.

In the following discussion, the client side client invoker and the server side server invoker will be referred to simply as "client invoker" and "server invoker." The callback client invoker and callback server invoker will be explicitly identified as such.

The following sequence of events occurs in the course of creating a control connection. For simplicity it is assumed that the `Client` and `Connector` have already been created, and that the callback server is created implicitly by the `Client`. These events are illustrated in Figure 5.1.

1. The application calls `Client.addListener()`.
2. The `Client` creates a callback `Connector` and the callback server invoker registers itself in a static map.
3. The `Client` sends an "addListener" message to the server invoker by way of the client invoker.
4. The client invoker intercepts the "addListener" message, which tells it that a callback server is being created. It retrieves the callback server invoker from the static map and tells it to create a control connection for the callback connection that is being constructed.
5. The callback server invoker sends an internal message to the server invoker requesting the address and port of the secondary `ServerSocket`
6. The callback server invoker connects to the secondary `ServerSocket` to create a `Socket` for the control connection. If it has not already done so, the callback server invoker creates a `TimerTask` which will monitor the state of all of its control connections. (Note that if the callback `Connector` is created explicitly, it could have multiple `InvokerCallbackHandlers` registered with it.)
7. On the server side, the `Socket` just created by the secondary `ServerSocket` is stored in a static map, awaiting the creation of the callback client invoker.
8. The client invoker transmits the "addListener" message to the server invoker.
9. The server invoker creates a callback client invoker.
10. The callback client invoker retrieves the waiting socket and uses it for the control connection.
11. The callback client invoker begins pinging on the control connection.

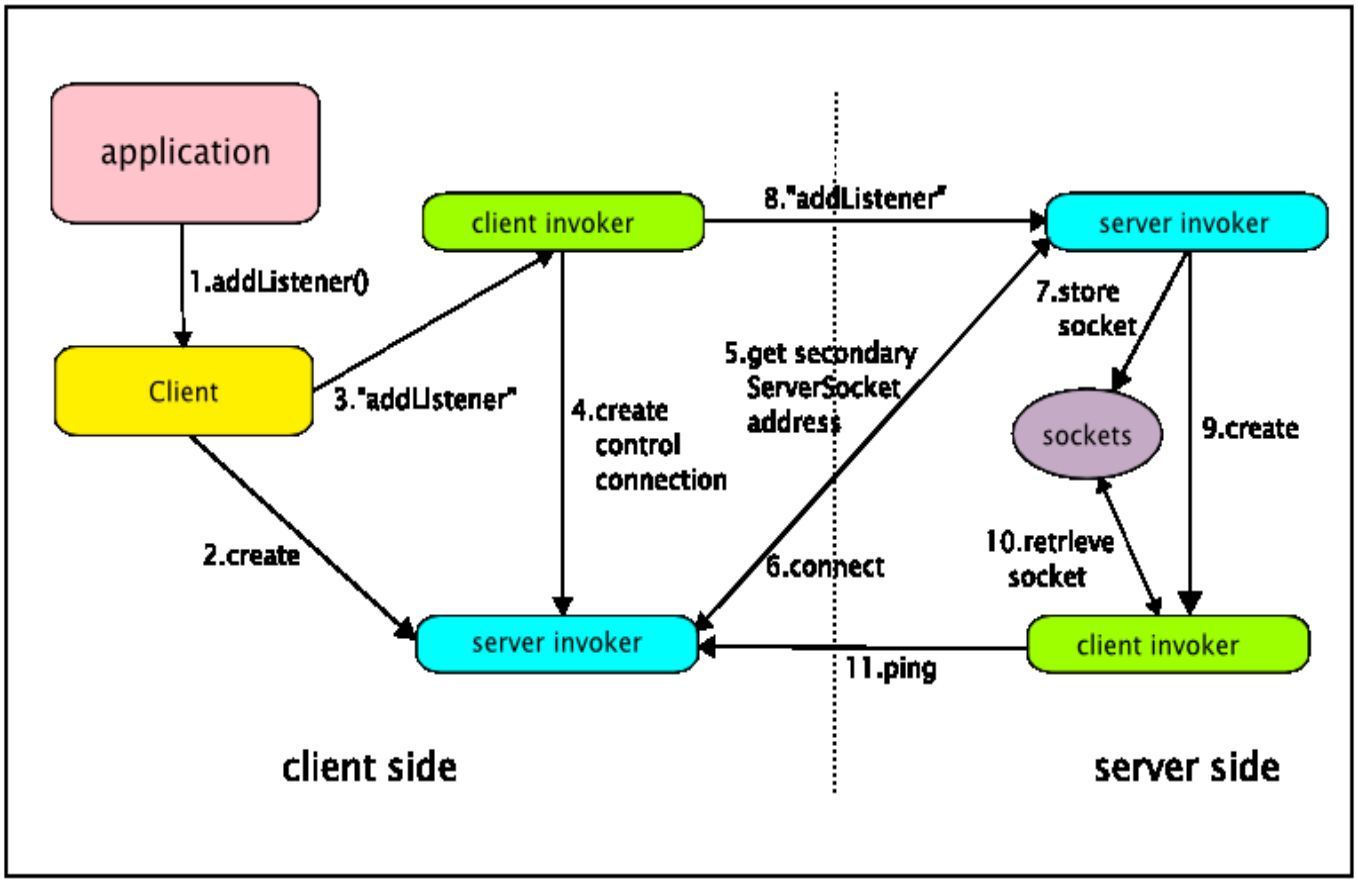


Figure 5.1. Creating a control connection.

The following sequence of events occurs in the course of creating a connection for the callback client invoker to use for sending callbacks. It is illustrated in Figure 5.2.

1. The `ServerInvocationHandler` calls `InvokerCallbackHandler.handleCallback()`.
2. The `InvocationCallbackHandler` calls `invoke()` on the callback `Client`.
3. The `Client` calls `invoke()` on the callback client invoker.
4. If there are no connections in its connection pool, the callback client invoker sends a message on the control connection asking the callback server invoker to connect to the server side secondary `ServerSocket`. It then waits for the `Socket` to appear in a static map.
5. The callback server invoker receives the request and calls upon either a `Socket` constructor or a `SocketFactory` to create a new `Socket`. It passes the new `Socket` to a worker thread to process subsequent callback invocations.
6. The secondary `ServerSocket` creates a new `Socket`, which is placed in a static map.
7. The callback client invoker retrieves the new `Socket`
8. The callback client uses the new `Socket` to transmit a callback, and adds the new connection to its connection

pool for later use.

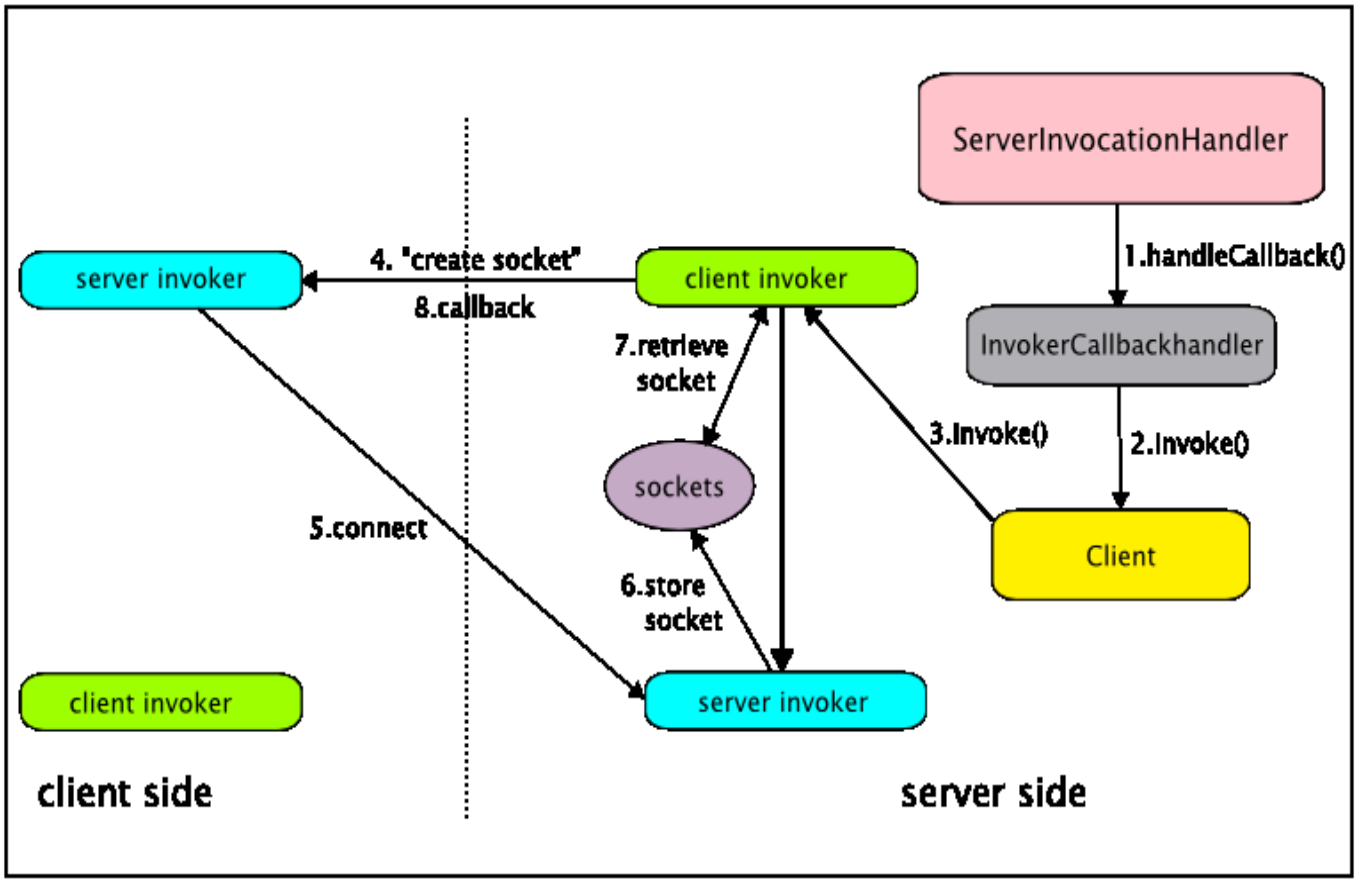


Figure 5.2. Creating a callback connection.

The following sequence of events occurs when a control connection fails. It is illustrated in Figure 5.3.

1. The callback server invoker notices that a ping has not been received during the control connection's current window.
2. The callback server invoker reacquires the host and port of the secondary `ServerSocket`, just in case it has changed.
3. The callback server invoker calls on a `Socket` constructor or `SocketFactory` to create a new `Socket`.
4. The callback server invoker sends an internal message on the new connection directing the server to replace the current control connection with the new connection.
5. After the secondary `ServerSocket` creates a new `Socket`, the `Socket` is passed directly to the client invoker in a method that replaces the old control connection with a new one.

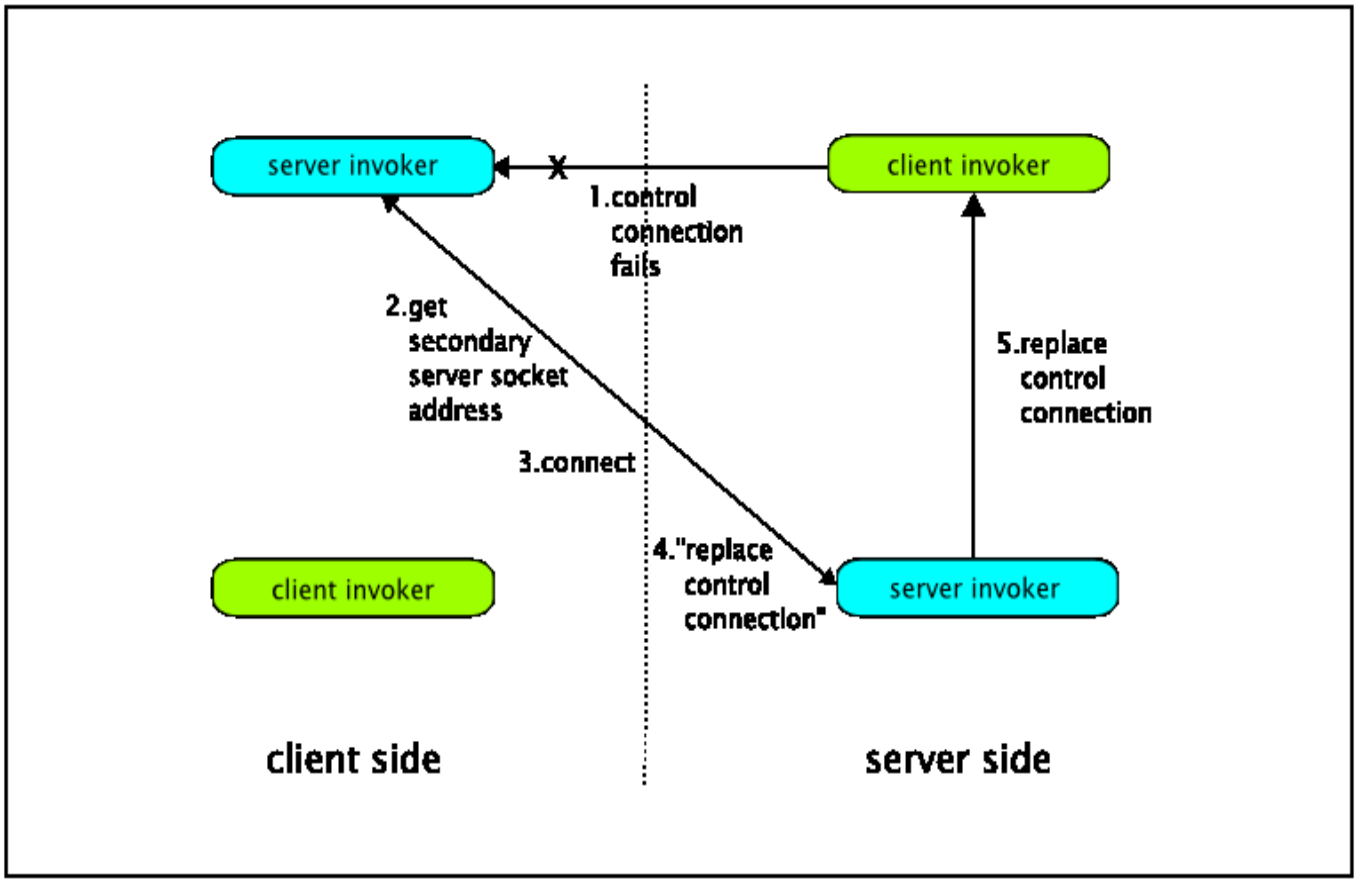


Figure 5.3. Replacing a failed control connection.

5.4.18. SSL Bisocket invoker

The SSL bisocket transport has the same relation to the bisocket transport as the SSL socket transport has to the socket transport. That is, it uses an `SSLServerSocket` and creates `SSLSockets` by default. See Section Socket factories and server socket factories for more information.

SSL bisocket transport supports all the configuration attributes supported by the bisocket transport.

5.5. Marshalling

Marshalling of data can range from extremely simple to somewhat complex, depending on how much customization is needed. The following explains how marshallers/unmarshallers can be configured. Note that this applies for all the different transports, but will use the socket transport for examples.

The easiest way to configure marshalling is to specify nothing at all. This will prompt the remoting invokers to use their default marshaller/unmarshallers. For example, the socket invoker will use the SerializableMarshaller/SerializableUnMarshaller and the http invoker will use the HTTPMarshaller/HTTPUnMarshaller, on both the client and server side.

The next easiest way is to specify the data type of the marshaller/unmarshaller as a parameter to the locator url.

This can be done by simply adding the key word 'datatype' to the url, such as:

```
socket://myhost:5400/?datatype=serializable
```

This can be done for types that are statically bound within the `MarshalFactory`, serializable and http, without requiring any extra coding, since they will be available to any user of remoting. However, is more likely this will be used for custom marshallers (since could just use the default data type from the invokers if using the statically defined types). If using custom marshaller/unmarshaller, will need to make sure both are added programmatically to the `MarshalFactory` during runtime (on both the client and server side). This can be done by the following method call within the MarshalFactory:

```
public static void addMarshaller(String dataType, Marshaller marshaller, UnMarshaller unMarshaller)
```

The dataType passed can be any String value desired. For example, could add custom InvocationMarshaller and InvocationUnMarshaller with the data type of 'invocation'. An example using this data type would then be:

```
socket://myhost:5400/?datatype=invocation
```

One of the problems with using a data type for a custom Marshaller/UnMarshaller is having to explicitly code the addition of these within the MarshalFactory on both the client and the server. So another approach that is a little more flexible is to specify the fully qualified class name for both the Marshaller and UnMarshaller on the locator url. For example:

```
socket://myhost:5400/?datatype=invocation&
            marshaller=org.jboss.invocation.unified.marshall.InvocationMarshaller&
            unmarshaller=org.jboss.invocation.unified.marshall.InvocationUnMarshaller
```

This will prompt remoting to try to load and instantiate the Marshaller and UnMarshaller classes. If both are found and loaded, they will automatically be added to the MarshalFactory by data type, so will remain in memory. Now the only requirement is that the custom Marshaller and UnMarshaller classes be available on both the client and server's classpath.

Another requirement of the actual Marshaller and UnMarshaller classes is that they have a void constructor. Otherwise loading of these will fail.

This configuration can also be applied using the service xml. If using declaration of invoker using the InvokerLocator attribute, can simply add the datatype, marshaller, and unmarshaller parameters to the defined InvokerLocator attribute value. For example:

```
        <attribute name="InvokerLocator">
           <![CDATA[socket://${jboss.bind.address}:8084/?datatype=invocation&
           marshaller=org.jboss.invocation.unified.marshall.InvocationMarshaller&
           unmarshaller=org.jboss.invocation.unified.marshall.InvocationUnMarshaller]]>
        </attribute>
```

If were using config element to declare the invoker, will need to add an attribute for each and include the isParam attribute set to true. For example:

```

```

```
        <invoker transport="socket">
           <attribute name="dataType" isParam="true">invocation</attribute>
           <attribute name="marshaller" isParam="true">
              org.jboss.invocation.unified.marshall.InvocationMarshaller
           </attribute>
           <attribute name="unmarshaller" isParam="true">
                 org.jboss.invocation.unified.marshall.InvocationUnMarshaller
           </attribute>
        </invoker>
```

This configuration is fine if the classes are present within the client's classpath. If they are not, can provide configuration for allowing clients to dynamically load the classes from the server. To do this, can use the parameter 'loaderport' with the value of the port you would like your marshal loader to run on. For example:

```
        <invoker transport="socket">
           <attribute name="dataType" isParam="true">invocation</attribute>
           <attribute name="marshaller" isParam="true">
              org.jboss.invocation.unified.marshall.InvocationMarshaller
           </attribute>
           <attribute name="unmarshaller" isParam="true">
              org.jboss.invocation.unified.marshall.InvocationUnMarshaller
           </attribute>
           <attribute name="loaderport" isParam="true">5401</attribute>
        </invoker>
```

When this parameter is supplied, the Connector will recognize this at startup and create a marshal loader connector automatically, which will run on the port specified. The locator url will be exactly the same as the original invoker locator, except will be using the socket transport protocol and will have all marshalling parameters removed (except the dataType). When the remoting client can not load the marshaller/unmarshaller for the specified data type, it will try to load them from the marshal loader service running on the loader port, including any classes they depend on. This will happen automatically and no coding is required (only the ability for the client to access the server on the specified loader port, so must provide access if running through firewall).

Note. Prior to release 2.2.3, only the parameters in the `InvokerLocator` were used to configure marshallers and unmarshallers. As of release 2.2.3 the parameters in the configuration map passed to an `org.jboss.remoting.Client` or `org.jboss.remoting.transport.Connector` will also be used if the parameter `org.jboss.remoting.Remoting.PASS_CONFIG_MAP_TO_MARSHAL_FACTORY` (actual value "passConfigMapToMarshalFactory") is set to "true".

Http marshalling

By default, the http, https, servlet, and sslservlet transports use `org.jboss.remoting.marshal.http.HTTPMarshaller` and `org.jboss.remoting.marshal.http.HTTPUnMarshaller`, which are distinct from their parent classes `org.jboss.remoting.marshal.serializable.SerializableMarshaller` and `org.jboss.remoting.marshal.serializable.SerializableUnMarshaller` only insofar as they treat Strings as a special case in the interest of efficiency. However, prior to release 2.2.3.SP1, `HTTPUnMarshaller` incorrectly uses the content-type set by the `ServerInvocationHandler` (if any), which would normally apply to the application payload of a response rather than the type of the envelope. As of release 2.2.3.SP1, this behavior remains the default, but it can be corrected by setting the parameter

`org.jboss.remoting.transport.http.HTTPMetadataConstants.USE_REMOTING_CONTENT_TYPE` (actual value "useRemotingContentType") to "true".

Compression marshalling

A compression marshaller/unmarshaller is available as well which uses gzip to compress and uncompress large payloads for wire transfer. The implementation classes are `org.jboss.remoting.marshal.compress.CompressingMarshaller` and `org.jboss.remoting.marshal.compress.CompressingUnMarshaller`. They extend the `org.jboss.remoting.marshal.serializable.SerializableMarshaller` and `org.jboss.remoting.marshal.serializable.SerializableUnMarshaller` interfaces and maintain the same behavior with the addition of compression.

5.6. Callbacks

5.6.1. Callback overview

Although this section covers callback configuration, it will be useful to begin with a little general information about callbacks within Remoting. In addition to the ordinary remote method invocation model, in which invocation results are returned synchronously, Remoting also supports an invocation model in which the server asynchronously generates information to be returned to the client.

There are two models for callbacks, **push callbacks** and **pull callbacks**. In the push model, the client registers a client side callback server with the target server. When the target server has a callback to deliver, it will call on the callback server directly and send the callback message. The other model, pull callbacks, allows the client to call on the target server to collect the callback messages waiting for it.

5.6.1.1. Callback connections

A **callback connection** is initiated by the invocation of one of the overloaded `addListener()` methods in the `org.jboss.remoting.Client` class, as described below in Section Registering callback handlers. The creation of a callback connection results in a server side call to the

```
public void addListener(InvokerCallbackHandler callbackHandler);
```

method of the application's `org.jboss.remoting.ServerInvocationHandler`. The `org.jboss.remoting.callback.InvokerCallbackHandler` parameter (actual type `org.jboss.remoting.callback.ServerInvokerCallbackHandler`) is the server side representation of the callback connection, essentially a proxy for the client side `InvokerCallbackHandler` passed to the `addListener()` method. The `ServerInvocationHandler` is free to do whatever it wants with the `InvokerCallbackHandler`, but a typical practice would be to keep a list of them and transmit each generated callback to some or all of them.

Note. As of release 2.2.2.SP12,, `ServerInvokerCallbackHandler` has a new method, `shutdown()`, which can be used by, for example, the `ServerInvocationHandler` with which it is registered, or by an `org.jboss.remoting.ConnectionListener` upon being informed of a connection failure, to clean up the `ServerInvokerCallbackHandler` and references thereto.

The client side of a callback connection is identified in one of two ways, according to whether there is a callback `Connector` associated with the connection. If the connection has a callback `Connector`, then it is identifed by the combination of the `Connector` and the `InvokerCallbackHandler`. It follows that if an `InvokerCallbackHandler` is registered twice with the same `Connector` (through a call to `Client.addListener()`), only a single callback connection is created. That is, the second call has no effect. If there is no callback `Connector`, which is the case for pull callbacks and simulated push callbacks (see Section Registering callback handlers), then the callback connection is identified by the combination of the `Client` on which `addListener()` was invoked and the `InvokerCallbackHandler`. It follows that if an `InvokerCallbackHandler` is registered twice with the same `Client` for pull or simulated push callbacks, only a single callback connection is created. That is, the second call has no effect.

Each callback connection is tagged with a unique identifier, which can be retrieved from the `InvokerCallbackHandler` passed to `ServerInvocationHandler.addListener()` by casting it to type `org.jboss.remoting.callback.ServerInvokerCallbackHandler` and calling `getCallbackSessionId()`. It is also possible to retrieve the unique identifier of the `Client` upon which `addListener()` was invoked by casting the `InvokerCallbackHandler` to type `ServerInvokerCallbackHandler` and calling `getClientSessionId()`.

5.6.1.2. Transmitting callbacks

Once the `ServerInvocationHandler` has generated information to be sent to the client, it can be packaged in an `org.jboss.remoting.callback.Callback` and transmitted on one or more callback connections in one of two ways. One way to transmit a callback is by invoking the

```
public void handleCallback(Callback callback) throws HandleCallbackException;
```

method of `InvokerCallbackHandler`. The subsequent disposition of the callback depends on whether the callback connection is configured for push or pull callbacks. For a pull callback connection, the `Callback` is simply stored on the server, and for a push callback connection, `handleCallback()` is analogous to (and is implemented by) an ordinary `Client.invoke()` invocation.

An alternative method of transmitting a callback is by casting an `InvokerCallbackHandler` to type `org.jboss.remoting.callback.AsynchInvokerCallbackHandler` and invoking one of the overloaded `handleCallbackOneway()` methods

```
public void handleCallbackOneway(Callback callback) throws HandleCallbackException;

public void handleCallbackOneway(Callback callback, boolean serverSide)) throws HandleCallbackE
```

of `AsynchInvokerCallbackHandler`. (Note that all `InvokerCallbackHandlers` passed in to `ServerInvocationHandler.addListener()` implement `AsynchInvokerCallbackHandler`.) For a pull callback connection `handleCallbackOneway()` has the same behavior as `handleCallback()`, but for a push callback connection it is analogous to (and implemented by) a `Client.invokeOneway()` invocation. The `serverSide` parameter is analogous to the `clientSide` parameter in the

```
public void invokeOneway(final Object param, final Map sendPayload, boolean clientSide) throws
```

method of `org.jboss.remoting.Client`. That is, if `serverSide` is true, then the oneway invocation is handed off to a separate thread on the server side and the call to `handleCallbackOneway()` returns immediately. If `serverSide` is false, then callback `Client` makes an invocation on the callback server, which hands the invocation off to a separate thread on the client side and returns, after which the call to `handleCallbackOneway()` returns.

5.6.1.3. Callback stores.

For pull callbacks (and also simulated push callbacks - see Section Registering callback handlers), the server has to manage callback messages until the client calls to collect them. Since the server has no control of when the client will call to get the callbacks, it has to be aware of memory constraints as it manages a growing number of callbacks. The way the callback server does this is through use of a **persistence policy**.

The persistence policy indicates at what point the server has too little free memory available and therefore the callback message should be put into a persistent store. This policy can be configured via the `memPercentCeiling` attribute (see more on configuring this below).

By default, the persistent store used by the invokers is the `org.jboss.remoting.NullCallbackStore`. The NullCallbackStore will simply throw away the callback to help avoid running out of memory. When the persistence policy is triggered and the NullCallbackStore is called upon to store the callback, the invocation handler making the call will be thrown an IOException with the message:

and there will be an error in the log stating which object was lost. In this same scenario, the client will get an instance of the `org.jboss.remoting.NullCallbackStore.FailedCallback` class when they call to get their callbacks. This class will throw a RuntimeException with the following message when `getCallbackObject()` is called:

Also, the payload of the callback will be the same string. The client will also get any valid callbacks that were kept in memory before the persistence policy was triggered.

An example case when using the NullCallbackStore might be when callback objects A, B, and C are stored in memory because there is enough free memory. Then when callback D comes, the persistence policy is triggered and the NullCallbackStore is asked to persist callback D. The NullCallbackStore will throw away callback D and create a FailedCallback object to take its place. Then callback E comes, and there is still too little free memory, so that is thrown away by the NullCallbackStore.

Then the client calls to get its callbacks. It will receive a List containing callbacks A, B, C and the FailedCallback. When the client asks the FailedCallback for its callback payload, it will throw the aforementioned exception.

Besides the default NullCallbackStore, there is a truly persistent CallbackStore, which will persist callback messages to disk so they will not be lost. The description of the CallbackStore is as follows:

Callback store configuration

CallbackStore is also a service mbean, so can be run as a service within JBoss AS or stand alone.

Remoting also offers the `BlockingCallbackStore`, which is described as follows.

Custom callback stores can also be implemented and defined within configuration. The only requirement is that it implements the org.jboss.remoting.SerializableStore interface and has a void constructor (only in the case of using a fully qualified classname in configuration).

Once a callback client has been removed as a listener, all persisted callbacks will be removed from disk.

5.6.1.4. Callback acknowledgements

Unlike the `Client.invoke()` method, `InvokerCallbackHandler.handleCallback()` has a void return type, so it does not provide a way of knowing if the callback has been received by the client. In fact, a void return type is appropriate since the immediate effect of a call to `InvokerCallbackHandler.handleCallback()` may be no more than storing the callback for later retrieval. However, it may be useful for the application to be informed when the callback has made its way to the client, and Remoting has a listener mechanism that can provide callback acknowledgements.

An object that implements the `org.jboss.remoting.callback.CallbackListener` interface

```
public interface CallbackListener
{
   /**
    * @param callbackHandler InvokerCallbackHandler that handled this callback
    * @param callbackId id of callback being acknowledged
    * @param response either (1) response sent with acknowledgement or (2) null
    */
   void acknowledgeCallback(InvokerCallbackHandler callbackHandler, Object callbackId, Object res
}
```

may be registered to receive an acknowledgement for a particular callback by adding it to the callback's `returnPayload` map with the key `org.jboss.remoting.callback.ServerInvokerCallbackHandler.CALLBACK_LISTENER` (actual value "callbackListener"). It is also necessary to assign an identifier to the callback by adding some unique object, recognizable by the application, to the callback's `returnPayload` map with the key `ServerInvokerCallbackHandler.CALLBACK_ID` (actual value "callbackId"). This identifier will be passed as the `callbackId` parameter of the `CallbackListener.acknowledgeCallback()` method.

There are two ways in which callbacks can be acknowledged:

1. explicit acknowledgements, and
2. automatic acknowledgements.

Note that automatic acknowledgements are available only for push callbacks and simulated push callbacks (see Section Registering callback handlers) transmitted by the `InvokerCallbackHandler.handleCallback()` method.

Callbacks may be acknowledged explicitly by the client side application code by calling one of the overloaded `acknowledgeCallback()` and `acknowledgeCallbacks()` methods

```
public int acknowledgeCallback(InvokerCallbackHandler callbackHandler, Callback callback) thro
public int acknowledgeCallback(InvokerCallbackHandler callbackHandler, Callback callback, Obje
public int acknowledgeCallbacks(InvokerCallbackHandler callbackHandler, List callbacks) throws
public int acknowledgeCallbacks(InvokerCallbackHandler callbackHandler, List callbacks, List r
```

of the `Client` class. In each case the `callbackHandler` parameter is the client side `InvokerCallbackHandler` which received the callback. The first two and the latter two methods acknowledge a single callback and a list of callbacks, respectively. In the latter case, each of the callbacks must have the same registered `CallbackListener`. The second and fourth methods also allow a response value to be associated with each callback acknowledgement, which will passed as the `response` parameter of the `CallbackListener.acknowledgeCallback()` method. For the fourth method, the lengths of the `callbacks` list and the `responses` list must be the same.

It is also possible to request that Remoting automatically supply acknowledgements for push callbacks and simulated push callbacks by adding the key `ServerInvokerCallbackHandler.REMOTING_ACKNOWLEDGES_PUSH_CALLBACKS` (actual value "remotingAcknowledgesPushCallbacks") to the callback's `returnPayload` map with the value of true, along with the `ServerInvokerCallbackHandler.CALLBACK_LISTENER` and `ServerInvokerCallbackHandler.CALLBACK_ID` entries. The acknowledgement is generated after the callback has been delivered by a call to `handleCallback()` on the client side `InvokerCallbackHandler`.

For an example of code that uses callback acknowledgements, see the classes in the package `org.jboss.remoting.samples.callback.acknowledgement`.

5.6.2. Registering callback handlers.

There are several ways in which callback handlers can be configured. The main distinction in type of callback setup is whether the callbacks will be push (asynchronous) or pull (synchronous) callbacks.

5.6.2.1. Pull callbacks.

A pull callback connection is implemented by an object (an `org.jboss.remoting.callback.ServerInvokerCallbackHandler`) on the server side which stores information that is generated asynchronously on the server and subsequently retrieved by the client. It is set up by invoking one of the following overloaded `addListener()` methods in the `Client` class:

```
public void addListener(InvokerCallbackHandler) throws Throwable;
public void addListener(InvokerCallbackHandler callbackHandler, InvokerLocator clientLocator) t
public void addListener(InvokerCallbackHandler callbackHandler,  InvokerLocator clientLocator,
```

where, in the latter two cases, the `clientLocator` parameter is set to null.

The callbacks stored for a pull callback connection may be retrieved by calling the

```
```

```
public List getCallbacks(InvokerCallbackHandler callbackHandler) throws Throwable
```

method of the `Client` class. Note that for pull callbacks, the `InvokerCallbackHandler` registered on the client side doesn't really participate in the handling of callbacks. However, when `client.getCallbacks(callbackHandler)` is called for a particular `Client` and `InvokerCallbackHandler`, the two objects together identify a particular callback connection.

Note. As of Remoting release 2.2.2.GA, there are two versions of pull callbacks: non-blocking (original) and blocking (new). In the original, non-blocking mode, a call to `Client.getCallbacks()` will return more or less immediately, whether or not any callbacks are waiting on the server side. In the new, blocking mode, the call will block on the server side until either it times out or a callback becomes available. The blocking mode eliminates the overhead of busy polling. Blocking and non-blocking mode are configured on a per-invocation basis by setting `org.jboss.remoting.ServerInvoker.BLOCKING_MODE` (actual value "blockingMode") to either `ServerInvoker.BLOCKING` (actual value "blocking") or `ServerInvoker.NONBLOCKING` (actual value "nonblocking") in the metadata map passed to

```
public List getCallbacks(InvokerCallbackHandler callbackHandler, Map metadata) throws Throwable
```

in `org.jboss.remoting.Client`. The default value is `ServerInvoker.NONBLOCKING`. The blocking timeout value may be configured in two ways:

1. the `Connector` can be configured with a default value; and
2. a per-invocation timeout value can be configured with the key `ServerInvoker.BLOCKING_TIMEOUT` in the metadata map passed to `Client.getCallbacks()`.

In the absence of any configured timeout, the default value is 5000 ms.

5.6.2.2. Push callbacks.

A push callback connection is implemented by a pair of objects, one on the server side and one on the client side, which facilitate transmitting to the client some information which has been generated asynchronously on the server. There are two versions of push callbacks: **true push callbacks** and **simulated push callbacks**, also known as **polled callbacks**.

In the case of true push callbacks, there is a Remoting object on the server side (an `org.jboss.remoting.callback.ServerInvokerCallbackHandler`) which uses a `Client` to make invocations to the client side. On the client side there is a `Connector` and an implementation of the `org.jboss.remoting.callback.InvokerCallbackHandler` interface which functions as an invocation handler for callbacks. Like implementations of `org.jboss.remoting.ServerInvocationHandler` on the server side, implementations of `InvokerCallbackHandler` are supplied by the application. When a `ServerInvocationHandler` generates a callback object, it will be sent to the callback `Connector`, which will, in turn, deliver it to the `InvokerCallbackHandler`

For simulated push callbacks, the server side Remoting object stores callbacks for later retrieval by the client, exactly as in the case of pull callbacks. However, there is a Remoting poller (an `org.jboss.remoting.callback.CallbackPoller`) on the client side which periodically retrieves the callbacks and,

as in the case of true push callbacks, delivers them to the `InvokerCallbackHandler`.

There are two ways to set up push callback handling, each of which entails the use of one of the overloaded `addListener()` methods in the `Client` class:

1. explicit creation of a `Connector`
2. implicit configuration.

In the first case, the application creates a `Connector` and passes its `InvokerLocator`, along with an implementation of `InvokerCallbackHandler`, to one of the following versions of `addListener()`:

```
        public void addListener(InvokerCallbackHandler callbackHandler, InvokerLocator clientLocator) t
        public void addListener(InvokerCallbackHandler callbackHandler,InvokerLocator clientLocator, Ob
```

Because there is a `Connector`, explicit configuration always results in true push callbacks.

In the case of implicit configuration, only the `InvokerCallbackHandler` is passed and Remoting takes care of the rest. One of the following versions of `addListener()` is used:

```
        public void addListener(InvokerCallbackHandler callbackhandler, Map metadata) throws Throwable;
        public void addListener(InvokerCallbackHandler callbackhandler, Map metadata, Object callbackHa
        public void addListener(InvokerCallbackHandler callbackhandler, Map metadata, Object callbackHa
```

Note that the latter three methods are distinguished from the first two by the presence of the `metadata` parameter, which can be used to configure the callback connection. Depending on the transport being used and the parameters supplied to `addListener()`, Remoting will set up either true or simulated push callbacks. If the client is in an environment where the server will be allowed to establish a connection to the client, then the final version of `addLIstener()` could be used with the `serverToClient` parameter set to true. In this case, regardless of the transport, Remoting will automatically create a callback `Connector` on behalf of the user, which behaves just as though the user had created it and passed the `InvokerLocator` as a parameter to `addListener()`.

Note. Since release 2.2.2.SP12, if the first instance of the parameter `org.jboss.remoting.Client.USE_ALL_PARAMS` (actual value "useAllParams") found successively in the `InvokerLocator`, the `Client`'s configuration map, or the metadata map is set to "true", then polled callbacks can be configured by parameters in the `InvokerLocator` and `Client`'s configuration map as well as the metadata map passed to `addListener()`.

If the client is in an environment where the server is not allowed to establish a network connection to the client (e.g. firewall rules disallow it or security rules prohibit the creation of a `ServerSocket`), then there are two options. One is to use one of the **bidirectional** transports, each of which has a strategy for the creation of a connection from the server to the client without connecting a client `Socket` to a `ServerSocket`. There are currently three bidirectional transports: local (i.e., the client and server reside in the same JVM), bisocket, and multiplex. When one of the second set of `addListener()` methods is invoked for a bidirectional transport, it will create a callback `Connector`, even if `serverToClient` is set to false. The other option is to use any of the **unidirectional** transports (socket, http, rmi) with `serverToClient` set to false (which is the default value if it is not an explicit parameter), in which case,

Remoting will configure polled callbacks.

The implicitly created callback `Connectors` are available for reuse. Each `Client` maintains a set of all callback `Connectors` it has implicitly created for each `InvokerCallbackHandler` that is passed in by way of one of the `addListener()` methods. For example,

```
InvokerCallbackHandler callbackHandler = new SampleCallbackHandler();
client.addListener(callbackHandler, new HashMap(), null, true);
client.addListener(callbackHandler, new HashMap(), null, true);
```

would result in a set of two callback `Connectors` associated with `callbackHandler`. These sets of callback `Connectors` are accessible by way of the `Client` method

```
public Set getCallbackConnectors(InvokerCallbackHandler callbackHandler);
```

A callback `Connector` could be reused as in the following code:

```
InvokerCallbackHandler callbackHandler1 = new SampleCallbackHandler();
client.addListener(callbackHandler1, new HashMap(), null, true);
Set callbackConnectors = client.getCallbackConnectors(callbackHandler1);
Connector callbackConnector = (Connector) callbackConnectors.iterator().next();
InvokerCallbackHandler callbackHandler2 = new SampleCallbackHandler();
client.addListener(callbackHandler2, callbackConnector.getLocator());
```

which would result in the implicitly created callback `Connector` having two registered `InvokerCallbackHandlers`. Note, by the way, that if the `InvokerCallbackHandler` were reused as in the following:

```
InvokerCallbackHandler callbackHandler1 = new SampleCallbackHandler();
client.addListener(callbackHandler1, new HashMap(), null, true);
Set callbackConnectors = client.getCallbackConnectors(callbackHandler1);
Connector callbackConnector = (Connector) callbackConnectors.iterator().next();
client.addListener(callbackHandler1, callbackConnector.getLocator());
```

then only one callback connection would be created, because a single (`Connector`, `InvokerCallbackHandler`) pair can be associated with only one callback connection.

Note. As of Remoting release 2.2.2.GA, there are two versions of pull callbacks: non-blocking (original) and blocking (new). For more information, see Pull callbacks. Since the `CallbackPoller` uses pull callbacks, this distinction is relevant to polled callbacks as well. The default behavior of `CallbackPoller` is to use non-blocking mode, but blocking mode can be requested by using the key `ServerInvoker.BLOCKING_MODE` set to `ServerInvoker.BLOCKING` in the metadata map passed to `Client.addListener()`.

There are nine parameters that can be passed to `addListener()` in the `metadata` map which are specific to push callback configuration. The first three apply to push callbacks and the latter six apply to polled callbacks. For convenience, the keys related to push callbacks are defined as constants in the `org.jboss.remoting.Client` class, and the keys related to polled callbacks are defined in the `org.jboss.remoting.callback.CallbackPoller` class(with the exception of `ServerInvoker.BLOCKING_MODE` and `ServerInvoker.BLOCKING_TIMEOUT`).

CALLBACK_SERVER_PROTOCOL (actual value is "callbackServerProtocol): the transport protocol to be used for callbacks. By default it will be the protocol used by the `Client` upon which `addListener()` is invoked.

CALLBACK_SERVER_HOST (actual value is "callbackServerHost"): the host name to be used by the callback server. By default it will be the result of calling `InetAddress.getLocalHost().getHostAddress()`.

CALLBACK_SERVER_PORT (actual value is "callbackServerPort"): the port to be used by the callback server. By default it will be a randomly chosen unused port.

CALLBACK_POLL_PERIOD (actual value is "callbackPollPeriod"): the interval in milliseconds between attempts to download callbacks from the server.

CALLBACK_SCHEDULE_MODE (actual value is "scheduleMode"): may be set to either `CallbackPoller.SCHEDULE_FIXED_RATE` (actual value "scheduleFixedRate") or `CallbackPoller.SCHEDULE_FIXED_DELAY` (actual value "scheduleFixedDelay"). In either case, polling will take place at approximately regular intervals, but in the former case the scheduler will attempt to perform each poll CALLBACK_POLL_PERIOD milliseconds after the previous attempt, and in the latter case the scheduler will attempt to schedule polling so that the *average* interval will be approximately CALLBACK_POLL_PERIOD milliseconds. `CallbackPoller.SCHEDULE_FIXED_RATE` is the default.

REPORT_STATISTICS (actual value is "reportStatistics"): The presence of this key in `metadata`, regardless of its value, will cause the `CallbackPoller` to print statistics that might be useful for configuring the other parameters..

MAX_ERROR_COUNT (actual value is "maxErrorCount"): determines the maximum number of errors that may be experienced during polling before `CallbackPoller` will shut itself down. The default value is "5".

SYNCHRONIZED_SHUTDOWN (actual value is "doSynchronizedShutdown"): if set to "true", `CallbackPoller.stop()` will wait for `Client.getCallbacks()` to return, and if set to "false" it will not wait. For blocking polled callbacks, the default value is "false" and for non-blocking polled callbacks, the default value is "true".

BLOCKING_MODE (actual value is "blockingMode"): if set to `ServerInvoker.BLOCKING` (actual value "blocking"), `CallbackPoller` will do blocking polled callbacks, and if set to `ServerInvoker.NONBLOCKING` (actual value "nonblocking"), `CallbackPoller` will do non-blocking polled callbacks.

Note that all of the elements in `metadata` will be passed to the callback `Connector` and appended to its `InvokerLocator`.

Note. As of Remoting release 2.2.2.GA, it is possible to configure a server side timeout value for sending push callbacks that is distinct from the timeout value used by the server. The parameter is `org.jboss.remoting.callback.ServerInvokerCallbackHandler.CALLBACK_TIMEOUT` (actual value "callbackTimeout"), and it should be used to configure the `Connector`. In the absence of `ServerInvokerCallbackHandler.CALLBACK_TIMEOUT`, the timeout value configured for the `Connector` will be used.

5.6.3. Unregistering callback handlers

Callback connections are torn down through a call to the method

```
public void removeListener(InvokerCallbackHandler callbackHandler) throws Throwable;
```

in the `org.jboss.remoting.Client` class. A `Client` can unregister only those `InvokerCallbackHandlers` that it originally registered.

It is good practice to eliminate callback connections when they are no longer needed. For example, callback `Connectors` can, depending on the transport, occupy TCP ports, and `CallbackPollers` will continue to poll as long as a connection exists.

5.6.4. Callback store configuration.

All callback store configuration will need to be defined within the server invoker configuration, since the server invoker is the parent that creates the callback stores as needed (when client registers for pull callbacks). Example service xml files are included below.

The following general callback store parameters may be configured. They are defined as constants in the `org.jboss.callback.ServerInvokerCallbackHandler` class.

CALLBACK_MEM_CEILING (actual value is "callbackMemCeiling"): the percentage of free memory available before callbacks will be persisted. If the memory heap allocated has reached its maximum value and the percent of free memory available is less than the callbackMemCeiling, this will trigger persisting of the callback message. The default value is 20.

Note: The calculations for this is not always accurate. The reason is that total memory used is usually less than the max allowed. Thus, the amount of free memory is relative to the total amount allocated at that point in time. It is not until the total amount of memory allocated is equal to the max it will be allowed to allocate. At this point, the amount of free memory becomes relevant. Therefore, if the memory percentage ceiling is high, it might not trigger until after free memory percentage is well below the ceiling.

CALLBACK_STORE_KEY (actual value is "callbackStore"): specifies the callback store to be used. The value can be either an MBean ObjectName or a fully qualified class name. If using class name, the callback store implementation must have a void constructor. The default is to use the NullCallbackStore.

The following parameters specific to `CallbackStore` can be configured via the invoker configuration as well. They are defined as constants in the `CallbackStore` class.

FILE_PATH_KEY (actual value is "StoreFilePath"): indicates to which directory to write the callback objects. The default value is the property value of 'jboss.server.data.dir' and if this is not set, then will be 'data'. Will then append 'remoting' and the callback client's session id. An example would be 'data\remoting\5c4o05l-9jijyx-e5b6xyph-1-e5b6xyph-2'.

FILE_SUFFIX_KEY (actual value is "StoreFileSuffix"): indicates the file suffix to use for the callback objects written to disk. The default value is 'ser'.

Sample service configuration

Socket transport with callback store specified by class name and memory ceiling set to 30%:

```
          <mbean code="org.jboss.remoting.transport.Connector"
                 name="jboss.remoting:service=Connector,transport=Socket"
                 display-name="Socket transport Connector">
```

```
            <attribute name="Configuration">
               <config>
                  <invoker transport="socket">
                     <attribute name="callbackStore">org.jboss.remoting.callback.CallbackStore</attribu
                     <attribute name="callbackMemCeiling">30</attribute>
                  </invoker>
                  <handlers>
                     <handler subsystem="test">
                        org.jboss.remoting.callback.pull.memory.CallbackInvocationHandler
                     </handler>
                  </handlers>
               </config>
            </attribute>
         </mbean>
```

Socket transport with callback store specified by MBean ObjectName and declaration of CallbackStore as service:

```
         <mbean code="org.jboss.remoting.callback.CallbackStore"
               name="jboss.remoting:service=CallbackStore,type=Serializable"
               display-name="Persisted Callback Store">

            <!-- the directory to store the persisted callbacks into -->
            <attribute name="StoreFilePath">callback_store</attribute>
            <!-- the file suffix to use for each callback persisted to disk -->
            <attribute name="StoreFileSuffix">cbk</attribute>
         </mbean>

         <mbean code="org.jboss.remoting.transport.Connector"
               name="jboss.remoting:service=Connector,transport=Socket"
               display-name="Socket transport Connector">

            <attribute name="Configuration">
               <config>
                  <invoker transport="socket">
                     <attribute name="callbackStore">
                        jboss.remoting:service=CallbackStore,type=Serializable
                     </attribute>
                  </invoker>
                  <handlers>
                     <handler subsystem="test">
                        org.jboss.remoting.callback.pull.memory.CallbackInvocationHandler
                     </handler>
                  </handlers>
               </config>
            </attribute>
         </mbean>
```

Socket transport with callback store specified by class name and the callback store's file path and file suffix defined:

```
         <mbean code="org.jboss.remoting.transport.Connector"
               name="jboss.remoting:service=Connector,transport=Socket"
               display-name="Socket transport Connector">

            <attribute name="Configuration">
               <config>
                  <invoker transport="socket">
                     <attribute name="callbackStore">org.jboss.remoting.callback.CallbackStore</attribu
                     <attribute name="StoreFilePath">callback</attribute>
```

```
                  <attribute name="StoreFileSuffix">cst</attribute>
               </invoker>
               <handlers>
                  <handler subsystem="test">
                     org.jboss.remoting.callback.pull.memory.CallbackInvocationHandler
                  </handler>
               </handlers>
            </config>
         </attribute>
      </mbean>
```

5.6.5. Callback Exception Handling

Since performing callbacks can sometimes fail, due to network errors or errors produced by the client callback handler, there needs to be a mechanism for managing exceptions when delivering callbacks. This is handled via use of the `org.jboss.remoting.callback.CallbackErrorHandler` interface. Implementations of this interface can be registered with the Connector to control the behavior when callback exceptions occur.

The implementation of the CallbackErrorHandler interface can be specified by setting the 'callbackErrorHandler' attribute to either the ObjectName of an MBean instance of the CallbackErrorHandler which is already running and registered with the MBeanServer, or can just specify the fully qualified class name of the CallbackErrorHandler implementation (which will be constructed on the fly and must have a void parameter constructor). The full server invoker configuration will be passed along to the CallbackErrorHandler, so if want to add extra configuration information in the invoker's configuration for the callback error handler, it will be available. If no callback error handler is specified via configuration, `org.jboss.remoting.callback.DefaultCallbackErrorHandler` will be used by default. This implementation will allow up to 5 exceptions to occur when trying to deliver a callback message from the server to the registered callback listener client (regardless of what the cause of the exception is, so could be because could not connect or could be because the client actually threw a valid exception). After the DefaultCallbackErrorHandler receives its fifth exception, it will remove the callback listener from the server invoker handler and shut down the callback listener proxy on the server side. The number of exceptions the DefaultCallbackErrorHandler will allow before removing the listener can by configured by the 'callbackErrorsAllowed' attribute.

Note. As of Remoting release 2.2.2.SP4, an `org.jboss.remoting.callback.ServerInvokerCallbackHandler`, which manages both push and pull callbacks on the server side, can register to be informed of a failure on the connection to the client that it is servicing. In particular, if there is a lease registered for the connection for that particular client, then the `ServerInvokerCallbackHandler` can be registered as a `org.jboss.remoting.ConnectionListener` for that lease. The default behavior is to do the registration, but the parameter `org.jboss.remoting.ServerInvoker.REGISTER_CALLBACK_LISTENER` (actual value "registerCallbackListener") may be set to "false" to prevent registration. If leasing is enabled and registration is turned on, a `ServerInvokerCallbackHandler` will shut itself down upon being informed of a connection failure. For more information about leasing, see Network Connection Monitoring.

5.7. Socket factories and server socket factories

All current transports depend on sockets and server sockets, and the ability to specify their implementation classes provides considerable power in configuring Remoting. Notably, SSL sockets and server sockets are the basis of secure communications in Remoting. This section covers the configuration of socket factories and server socket

factories on both the server side and the client side, and then focuses on SSL configuration.

5.7.1. Server side programmatic configuration

All server invokers use server sockets, and it makes sense, therefore, to be able to configure server invokers with server socket factories. It is also true, though less obvious. that server invokers create sockets (other than by way of server sockets). When a server invoker makes a push callback to a client, it creates a client invoker, which creates a socket. Moreover, some server invokers, e.g., the RMI server invoker, have their own idiosyncratic uses for socket factories. Remoting offers a number of ways of configuring socket factories and server socket factories, and these apply to all transports (except for the servlet invokers).

5.7.1.1. Server socket factories.

For `ServerSocketFactory`s, there are ten options for programmatic configuration:

1. Get the `ServerInvoker` by calling `Connector.getServerInvoker()` and call `ServerInvoker.setServerSocketFactory()`.

2. Call `Connector.setServerSocketFactory()`.

3. Put a constructed `ServerSocketFactory` in a configuration map, using key `Remoting.CUSTOM_SERVER_SOCKET_FACTORY`, and pass the map to one of the `Connector` constructors.

4. Create an xml document with root element `<config>`, setting the `<serverSocketFactory>` attribute to the name of a `ServerSocketFactoryMBean` and pass the document to `Connector.setConfiguration()`. For example:

```
StringBuffer buf = new StringBuffer();
buf.append("<?xml version=\"1.0\"?>\n");
buf.append("<config>");
buf.append("   <invoker transport=\"sslsocket\">");
buf.append("      <attribute name=\"serverBindAddress\">" + getHostName() + "</at
buf.append("      <attribute name=\"serverBindPort\">" + freeport + "</attribute>
buf.append("      <attribute name=\"serverSocketFactory\">" + socketFactoryObjNam
buf.append("   </invoker>");
buf.append("</config>");
ByteArrayInputStream bais = new ByteArrayInputStream(buf.toString().getBytes());
Document xml = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(ba
connector.setConfiguration(xml.getDocumentElement());
```

5. Create an xml document with root element `<config>`, setting the `<serverSocketFactory>` attribute to the class name of a `ServerSocketFactory` and pass the document to `Connector.setConfiguration()`. The `<serverSocketFactory>` class must have a default constructor, which will be used to create a `ServerSocketFactory`.

6. Put the `ObjectName` of a `ServerSocketFactoryMBean` in a configuration map, using key `ServerInvoker.SERVER_SOCKET_FACTORY`, and pass the map to one of the `Connector` constructors.

7. Put the class name of a `ServerSocketFactory` in a configuration map, using key `ServerInvoker.SERVER_SOCKET_FACTORY`, and pass the map to one of the `Connector` constructors. The

<serverSocketFactory> class must have a default constructor, which will be used to create a ServerSocketFactory.

8. Put a set of SSL parameters, using the keys in org.jboss.remoting.security.SSLSocketBuilder, in a configuration map and pass the map to one of the Connector constructors. These will be used by SSLSocketBuilder (see below) to create a CustomSSLServerSocketFactory.

9. Configure an appropriate set of SSL system properties and use one of the SSL transports (https, sslmultiplex, sslrmi, or sslsocket). The properties will be used to create some kind of SSLServerSocketFactory, as determined by the transport.

10. Use one of the non-SSL transports and do nothing. A default ServerSocketFactory will be constructed.

These options are essentially in descending order of precedence. If options 3 and 6, for example, are both used, the factory passed in option 3 will prevail. Options 4 and 5 are mutually exclusive, as are options 6 and 7. Options 1, 2, 3, 5, and 7 are illustrated in FactoryConfigSample and options 4, 6, 8, and 9 are illustrated in FactoryConfigSSLSample, both of which are in package org.jboss.remoting.samples.config.factories.

Timing considerations. The ServerInvoker, for any transport, is created during the call to Connector.create(), before which option 1 is unavailable. Option 2, on the other hand, is only available before the call to Connector.create(). Once the ServerInvoker has been created, it selects a ServerSocketFactory, according to the rules enumerated above, during the create() phase. For all current transports, the actual ServerSocket is created during the call to Connector.start(), so that a call to ServerInvoker.setServerSocketFactory() (option 1) can override the selected ServerSocketFactory until Connector.start() is called.

5.7.1.2. Socket factories

For SocketFactorys, there are also ten options for programmatic configuration, and they are essentially the same as the previous ten. Note, however, that options 5 and 6 are reversed. This is because an ServerSocketFactoryMBean, if it exists, is given precedence over class names:

1. Call Connector.setSocketFactory().

2. Get the ServerInvoker by calling Connector.getServerInvoker() and call ServerInvoker.setSocketFactory().

3. Put a constructed SocketFactory in a configuration map, using key Remoting.CUSTOM_SOCKET_FACTORY, and pass the map to one of the Connector constructors.

4. Create an xml document with root element <config>, setting the <serverSocketFactory> attribute to the name of a ServerSocketFactoryMBean and pass the document to Connector.setConfiguration(). If the MBean has type SSLServerSocketFactoryServiceMBean, its configuration information will be gathered and used to construct a CustomSSLSocketFactory. **Note.** This method is guaranteed to work only for callback client invokers. For other, transport specific, socket factory uses, the transport may or may not use this information.

5. Put the ObjectName of a ServerSocketFactoryMBean in a configuration map, using key ServerInvoker.SERVER_SOCKET_FACTORY, and pass the map to one of the Connector constructors. If the MBean has type SSLServerSocketFactoryServiceMBean, its configuration information will be gathered and used to construct a CustomSSLSocketFactory. **Note.** This method is guaranteed to work only for callback client invokers.

For other, transport specific, socket factory uses, the transport may or may not use this information.

6. Create an xml document with root element `<config>`, setting the `<socketFactory>` attribute to the class name of a `SocketFactory` and pass the document to `Connector.setConfiguration()`. For example:

```
StringBuffer buf = new StringBuffer();
buf.append("<?xml version=\"1.0\"?>\n");
buf.append("<config>");
buf.append("   <invoker transport=\"sslsocket\">");
buf.append("      <attribute name=\"serverBindAddress\">" + getHostName() + "</at
buf.append("      <attribute name=\"serverBindPort\">" + freeport + "</attribute>
buf.append("      <attribute name=\"socketFactory\">" + socketFactoryClassname +
buf.append("   </invoker>");
buf.append("</config>");
ByteArrayInputStream bais = new ByteArrayInputStream(buf.toString().getBytes());
Document xml = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(ba
connector.setConfiguration(xml.getDocumentElement());
```

   The `SocketFactory` class must have a default constructor, which will be used to create a `SocketFactory`.

7. Put the class name of a `SocketFactory` in a configuration map, using key `Remoting.SOCKET_FACTORY_CLASS_NAME`, and pass the map to one of the `Connector` constructors. The `SocketFactory` class must have a default constructor.

8. Put a set of SSL parameters, using the keys in `org.jboss.remoting.security.SSLSocketBuilder`, in a configuration map and pass the map to one of the `Connector` constructors. These will be used by `SSLSocketBuilder` (see below) to create a `CustomSSLSocketFactory`.

9. Configure an appropriate set of SSL system properties and use one of the SSL transports (https, sslmultiplex, sslrmi, or sslsocket). The properties will be used to create some kind of `SSLSocketFactory`, as determined by the transport.

10. Use one of the non-SSL transports and do nothing. Ordinary `Sockets` will be used.

Again, these are essentially in descending order of precedence. Options 1, 2, 3, 6, and 7 are illustrated in `FactoryConfigSample` and options 4, 5, 8, and 9 are illustrated in `FactoryConfigSSLSample`, both of which are in package `org.jboss.remoting.samples.config.factories`.

Timing considerations. A new `Client`, with a client invoker, is created on the server side whenever a callback listener is registered by a call to `Client.addListener()`. If a `SocketFactory` is supplied by any of options 1 to 5, it will be passed to the `Client`. Otherwise, any information from options 6 to 9 will be passed to the client invoker, which will create a `SocketFactory` according to the rules given below in the section on client side socket factory configuration. Once `Connector.create()` has been called, `ServerInvoker.setSocketFactory()`, may be called at any time to determine the `SocketFactory` used by the next callback client invoker.

5.7.2. Client side programmatic configuration

On the client side it is possible to configure socket factories for client invokers and to configure server socket factories for callback server invokers. Configuration on the client side is largely the same as configuration on the server side, with the exception that no `MBeanServer` is assumed to be present, and the `Client` has no facilities for parsing xml documents.

5.7.2.1. Server socket factories.

For ServerSocketFactorys in callback server invokers, there are eight options for programmatic configuration, which are identical to options 1-3, 5 and 7-10 on the server side (we don't assume the existence of an MBeanServer on the client side:

1. Get the ServerInvoker by calling Connector.getServerInvoker() and call ServerInvoker.setServerSocketFactory().

2. Call Connector.setServerSocketFactory().

3. Put a constructed ServerSocketFactory in a configuration map, using key Remoting.CUSTOM_SERVER_SOCKET_FACTORY, and pass the map to one of the Connector constructors.

4. Create an xml document with root element <config>, setting the <serverSocketFactory> attribute to the class name of a ServerSocketFactory and pass the document to Connector.setConfiguration(). For example:

```
StringBuffer buf = new StringBuffer();
buf.append("<?xml version=\"1.0\"?>\n");
buf.append("<config>");
buf.append("   <invoker transport=\"sslsocket\">");
buf.append("      <attribute name=\"serverBindAddress\">" + getHostName() + "</at
buf.append("      <attribute name=\"serverBindPort\">" + freeport + "</attribute>
buf.append("      <attribute name=\"serverSocketFactory\">" + serverSocketFactory
buf.append("   </invoker>");
buf.append("</config>");
ByteArrayInputStream bais = new ByteArrayInputStream(buf.toString().getBytes());
Document xml = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(ba
connector.setConfiguration(xml.getDocumentElement());
```

   The ServerSocketFactory class must have a default constructor, which will be used to create a ServerSocketFactory.

5. Put the class name of a ServerSocketFactory in a configuration map, using key ServerInvoker.SERVER_SOCKET_FACTORY, and pass the map to one of the Connector constructors. The ServerSocketFactory class must have a default constructor, which will be used to create a ServerSocketFactory.

6. Put a set of SSL parameters, using the keys in org.jboss.remoting.security.SSLSocketBuilder, in a configuration map and pass the map to one of the Connector constructors. These will be used by SSLSocketBuilder (see below) to create a CustomSSLServerSocketFactory.

7. Configure an appropriate set of SSL system properties and use one of the SSL transports (https, sslmultiplex, sslrmi, or sslsocket). The properties will be used to create some kind of SSLServerSocketFactory, as determined by the transport.

8. Use one of the non-SSL transports and do nothing. A default ServerSocketFactory will be constructed.

These options are essentially in descending order of precedence. For example, if options 3 and 5, for example, are both used, the factory passed in options 3 will prevail. Options 1, 2, 3, 4, and 5 are illustrated in FactoryConfigSample and options 6 and 7 are illustrated in FactoryConfigSSLSample, both of which are in package org.jboss.remoting.samples.config.factories.

Timing considerations. See the discussion in the section on the creation of server socket factories on the server side.

5.7.2.2. Socket factories.

For `SocketFactory`s in client invokers, there are seven options for programmatic configuration, and they are essentially the same as 1-3 and 5-8 in the previous section (`Client` has no facility for parsing xml documents:

1. Get the `ClientInvoker` by calling `Client.getInvoker()` and call `ClientInvoker.setSocketFactory()`.
2. Call `Client.setSocketFactory()`.
3. Put a constructed `SocketFactory` in a configuration map, using key `Remoting.CUSTOM_SOCKET_FACTORY`, and pass the map to one of the `Client` constructors.
4. Put the class name of a `SocketFactory` in a configuration map, using key `Remoting.SOCKET_FACTORY_CLASS_NAME`, and pass the map to one of the `Client` constructors. The `SocketFactory` class must have a default constructor, which will be used to create a `SocketFactory`.
5. Put a set of SSL parameters, using the keys in `org.jboss.remoting.security.SSLSocketBuilder`, in a configuration map and pass the map to one of the `Client` constructors. These will be used by `SSLSocketBuilder` (see below) to create a `CustomSSLSocketFactory`.
6. Configure an appropriate set of SSL system properties and use one of the SSL transports (https, sslmultiplex, sslrmi, or sslsocket). The properties will be used to create some kind of `SSLSocketFactory`, as determined by the transport.
7. Use one of the non-SSL transports and do nothing. Ordinary `Sockets` will be used.

Again, these are essentially in descending order of precedence. Options 1, 2, 3, and 4 are illustrated in `FactoryConfigSample` and options 5 and 6 are illustrated in `FactoryConfigSSLSample`, both of which are in package `org.jboss.remoting.samples.config.factories`.

Timing considerations. A `SocketFactory` is created in the constructor for `RemoteClientInvoker`, the ancestor of all current remote client invokers (that is, all client invokers except `LocalClientInvoker`, which can make a call by reference on a server invoker in the same JVM), but it is currently used only by SSL transports, for which the timing considerations vary.

1. **https:** `HTTPSClientInvoker` sets the socket factory on its `HttpsURLConnection` each time `Client.invoke()` is called. Option 1 may be used to reset the `SocketFactory` for future invocations at any time.
2. **sslmultiplex:** Whichever of `SSLMultiplexClientInvoker` or `SSLMultiplexServerInvoker` first gets sufficient bind and connect information to create a priming socket (see the section on the multiplex invoker for a discussion of priming sockets) passes the current `SocketFactory` to be used to create the actual socket that supports the multiplexed connection. This happens during the call to either `Client.connect()` or `Connector.create()`. Once the actual socket is created, no further configuration is possible
3. **sslrmi:** A `SocketFactory` is either created or configured for future creation during `Client.create()`. No further configuration is possible.

4. **sslsocket:** `SSLSocketClientInvoker` uses the current `SocketFactory` to create a new socket whenever it runs out of available pooled connections. Option 1 may be used to reset the `SocketFactory` for future connections at any time.

5.7.3. Server side configuration in the JBoss Application Server

Everything in the previous two sections applies to configuring socket and server socket factories in any environment, including inside the JBoss Application Server (JBossAS), but JBossAS adds some new options. In particular, the `SARDeployer` (see *The JBoss 4 Application Server Guide* on the labs.jboss.org web site) can read information from a `*-service.xml` file, as discussed above in the section "General Connector and Invoker configuration," and use it to configure MBeans such as `Connectors`.

An example of a service xml that covers all the different transport and service configurations can be found within the example-service.xml file under the etc directory of the JBoss Remoting distribution.

The server socket factory to be used by a server invoker can be set via configuration within the service xml. To do this, the serverSocketFactory attribute will need to be set as a sub-element of the invoker element (this cannot be done if just specifying the invoker configuration using the InvokerLocator attribute). The attribute value must be either

1. the JMX ObjectName of an MBean that implements the `org.jboss.remoting.security.ServerSocketFactoryMBean` interface, or

2. the class name of a `ServerSocketFactory` with a default constructor.

An example of the first case would be:

```
<mbean code="org.jboss.remoting.transport.Connector"
      name="jboss.remoting:service=Connector,transport=Socket"
      display-name="Socket transport Connector">

   <attribute name="Configuration">
      <config>
         <invoker transport="sslsocket">
            <attribute name="serverSocketFactory">
               jboss.remoting:service=ServerSocketFactory,type=SSL
            </attribute>
            <attribute name="numAcceptThreads">1</attribute>
```

The `serverSocketFactory` attribute is processed as follows:

1. Take its String value, create an `ObjectName` from it, and look up an MBean with that name from the `MBeanServer` that the invoker has been registered with (by way of the `Connector`). If an MBean with that name is found, create a proxy to it of type `org.jboss.remoting.security.ServerSocketFactoryMBean`. (Technically, a user could set the `serverSocketFactory` property with the locator url, but the preferred method is to use the explicit configuration via the invoker element's attribute, as discussed above.)

2. If no MBean is found with a matching `ObjectName`, treat the `serverSocketFactory` attribute as a class name and try to create an instance using the default constructor.

The JBossRemoting project provides an implementation of the ServerSocketFactoryMBean that can be used and should provide most of the customization features that would be needed. More on this implementation later.

Note that these two options correspond exactly to options 4 and 5 in section Server socket factories (on the server side), which is how these two new options are implemented.

Timing considerations. If a `Connector` is accessed by way of the `MBeanServer`, then most of the options for configuring the server socket factory discussed in Server socket factories are irrelevant since `ConnectorMBean` does not expose methods for using them. However, when a `Connector` that is registered with an `MBeanServer` creates a server invoker during a call to `Connector.create()`, it also registers the server invoker with the same `MBeanServer`, which means that the server invoker is accessible by way of its `ObjectName`, which has the form

```
jboss.remoting:service=invoker,transport=socket,host=www.jboss.com,port=8765
```

for example, followed by additional parameter=value pairs. (See the jmx-console for a running instance of JBossAS at http://localhost:8080/jmx-console/ to see examples of server invoker `ObjectNames`.) Now, if another MBean is configured in a `*-service.xml` file to be dependent on the server invoker MBean, e.g.

```
<mbean code="org.jboss.BlueMonkey" name="jboss.remoting:bluemonkey,name=diamond">
   <depends optional-attribute-name="serverInvoker">
       jboss.remoting:service=invoker,transport=socket,host=www.jboss.com,port=8765
   </depends>
</mbean>
```

then `org.jboss.BlueMonkey.create()` will have access to the designated server invoker after the invoker has been created but before it has been started, which means that `ServerInvoker.setServerSocketFactory()` will be effective. (See the *The JBoss 4 Application Server Guide*, Chapter 2, for more information about the life cycle of JBoss MBeans.)

5.7.4. Client side configuration from the JBoss Application Server

As described in section Declarative configuration the declarative xml files used by the Application Server can be used to configure the client by way of the parameters included in the `InvokerLocator`. However, a peculiarity in the way socket factories are created on the client restricts the parameters taken into consideration to those passed in the configuration map passed to the `org.jboss.remoting.Client` constructor. The following two parameters, introduced in releases 2.2.2.SP9 and 2.2.3, respectively, eliminate that restriction.

org.jboss.remoting.Remoting.SOCKET_FACTORY_NAME (actual value is 'socketFactory') - key for giving the name of the socket factory class to be used by clients.

org.jboss.remoting.Remoting.USE_ALL_SOCKET_FACTORY_PARAMS (actual value is 'useAllSocketFactoryParams') - key for indicating that all socket factory parameters in the `InvokerLocator` should be used by clients.

5.7.5. Socket creation listeners

Every Remoting transport uses `Sockets`, but the creation and management of the `Sockets` is generally inaccessible from the application code. Remoting has a hook that can provide access to `Sockets`, in the form of a listener inter-

face in the `org.jboss.remoting.socketfactory` package:

```
public interface SocketCreationListener
{
   /**
    * Called when a socket has been created.
    *
    * @param socket socket that has been created
    * @param source SocketFactory or ServerSocket that created the socket
    * @throws IOException
    */
   void socketCreated(Socket socket, Object source) throws IOException;
}
```

Socket creation listeners can be registered to be informed every time a socket is created by a `SocketFactory` or `ServerSocket`. The mechanisms for registering listeners are the usual ones, e.g., by putting them in configuration maps passed to client and server invokers. (See Section General transport configuration for a general discussion of parameter configuration in Remoting.) In any case they should be associated with one of the following keys from `org.jboss.remoting.Remoting`:

```
/**
 * Key for the configuration map passed to a Client or Connector to indicate
 * a socket creation listener for sockets created by a SocketFactory.
 */
public static final String SOCKET_CREATION_CLIENT_LISTENER = "socketCreationClientListener";

/**
 * Key for the configuration map passed to a Client or Connector to indicate
 * a socket creation listener for sockets created by a ServerSocket.
 */
public static final String SOCKET_CREATION_SERVER_LISTENER = "socketCreationServerListener";
```

The value associated with either of these keys can be an actual object, or, to facilitate configuration by `InvokerLocator` or xml, it can be the name of a class that implements `SocketCreationListener` and has a default constructor

Note that client and server invokers always use the respective keys SOCKET_CREATION_CLIENT_LISTENER and SOCKET_CREATION_SERVER_LISTENER, whether they are on the client side or server side. For example, a callback client invoker would be configured by putting a listener with the key SOCKET_CREATION_CLIENT_LISTENER in the configuration map passed to the server side `Connector`, which will find its way to the callback client invoker when a callback handler is registered.

The creation listener facility currently is supported by the following transports: bisocket, sslbisocket, https, multiplex, sslmultiplex, rmi, sslrmi, socket, and sslsocket. It is not supported by http because `HttpURLConnection` does not expose its socket factory (though `HttpsURLConnection` does). It is not supported by the servlet transport because invocations with the servlet transport go through a servlet container, which is outside the scope of Remoting.

5.7.6. SSL transports

There are now four transports that support SSL: https, sslmultiplex, sslrmi, and sslsocket (plus sslservlet, which is not relevant here). All of the preceding discussion applies to each of these, and, moreover, they are all extensions of their non-ssl counterparts, so only some ssl specific information will be added here.

https

Configuration of the https transport is a bit different from that of the other transports since the implementation is based off the Tomcat connectors. One difference is that, in order to use SSL connections, the `SSLImplementation` attribute must be set and must always have the value `org.jboss.remoting.transport.coyote.ssl.RemotingSSLImplementation`. The `SSLImplementation` is used by the Tomcat connector to create `ServerSocketFactorys`, and `RemotingSSLImplementation` presents Tomcat with the `ServerSocketFactory` configured according to the options described above.

An example of setting up https via service.xml configuration would be:

```
    <mbean code="org.jboss.remoting.transport.Connector"
        name="jboss.remoting:service=Connector,transport=HTTPS"
        display-name="HTTPS transport Connector">

        <attribute name="Configuration">
           <config>
              <invoker transport="https">
                 <attribute name="serverSocketFactory">jboss.remoting:service=ServerSocketFactory,type=S
                 <attribute name="SSLImplementation">org.jboss.remoting.transport.coyote.ssl.RemotingSSL
                 <attribute name="serverBindAddress">${jboss.bind.address}</attribute>
                 <attribute name="serverBindPort">6669</attribute>
              </invoker>
              <handlers>
                 <handler subsystem="mock">org.jboss.test.remoting.transport.mock.MockServerInvocationHa
              </handlers>
           </config>
        </attribute>
        <!-- This depends is included because need to make sure this mbean is running before configure i
        <depends>jboss.remoting:service=ServerSocketFactory,type=SSL</depends>
     </mbean>
```

See section SSLServerSocketFactoryService below for a discussion of the "jboss.remoting:service=ServerSocketFactory,type=SSL" MBean that appears in this configuration element.

Note that the configuration for SSL support only works when using the java based http processor and not with the APR based transport. See section HTTP Invoker for more information on using the APR based transport.

sslmultiplex

The sslmultiplex server invoker inherits from the socket server invoker a method with signature

```
        public void setNewServerSocketFactory(ServerSocketFactory serverSocketFactory)
```

which supports dynamic replacement of server socket factories. The principal motivation for this facility is to be able to swap in a new `SSLServerSocketFactory` configured with an updated keystore.

sslrmi

The extra twist in the sslrmi invoker is that the server invoker creates the (client) socket factory and packages it with its own stub, from which it follows that the socket factory must be serializable. If the sslrmi server invoker is allowed to create an `SSLSocketFactory` from SSL parameters, it will take care to create a serializable socket factory. In particular, the server invoker creates a copy of

`org.jboss.remoting.transport.rmi.ssl.SerializableSSLClientSocketFactory`, which is essentially just a holder for the configuration map passed to the server invoker, with any parameters removed which concern trust store and key store configuration. On the client side, when an sslrmi client invoker is created, it stores its own configuration map in a static variable which the transferred `SerializableSSLClientSocketFactory` can retrieve and merge with the configuration information it brought with it from the server. In particular, if a socket factory is explicitly passed to the client invoker, then `SerializableSSLClientSocketFactory` will use it. If not, then `SerializableSSLClientSocketFactory` will use any key store and trust store information passed to the client to create and configure a socket factory.

Note. If instead of using `SerializableSSLClientSocketFactory`, a socket factory is passed in to the server invoker by one of the methods discussed above, then the user is responsible for supplying a serializable socket factory.

sslsocket

In addition to the various configuration options discussed above, the sslsocket transport exposes the

```
public void setServerSocketFactory(ServerSocketFactory serverSocketFactory)
```

method as a JMX operation.

Also, the sslsocket server invoker inherits from the socket server invoker a method with signature

```
            public void setNewServerSocketFactory(ServerSocketFactory serverSocketFactory)
```

which supports dynamic replacement of server socket factories. The principal motivation for this facility is to be able to swap in a new `SSLServerSocketFactory` configured with an updated keystore.

5.7.7. SSLSocketBuilder

Throughout this section reference has been made to SSL socket factory and server socket factory configuration parameters. This subsection will introduce these parameters in the context of configuring `org.jboss.remoting.security.SSLSocketBuilder`, Remoting's flexible, highly customizable master factory for creating socket and server socket factories. It can be used programmatically on both the client and server side, and it is also a service MBean, so it can be configured and started from within a service xml in a JBossAS environment.

Once a `SSLSocketBuilder` has been constructed and configured, a call to its method `createSSLServerSocketFactory()` will return a custom instance of a `SSLServerSocketFactory`, and a call to `createSSLSocketFactory()` will return a custom instance of `SSLSocketFactory`.

There are two modes in which the `SSLSocketBuilder` can be run. The first is the default mode where all that is needed is to declare the `SSLSocketBuilder` and set the system properties `javax.net.ssl.keyStore` and `javax.net.ssl.keyStorePassword`. This will use the JVM vendor's default configuration for creating the SSL server socket factory.

In order to customize any of the SSL properties, the first requirement is that the default mode is turned off. This is **IMPORTANT** because otherwise, if the default mode is not explicitly turned off, all other settings will be IGNORED, even if they are explicitly set. To turn off the default mode via service xml configuration, set the `UseSSLServerSocketFactory` attribute to false. This can also be done programmatically by calling the

`setUseSSLServerSocketFactory()` and passing false as the parameter value.

There are two ways to configure a `SSLSocketBuilder`

1. set its bean attributes, either programmatically or by xml configuration, or
2. pass to a `SSLSocketBuilder` constructor a configuration map with keys defined in the `SSLSocketBuilder` class.

The configuration properties for `SSLSocketBuilder`are as follows. Note that in the "key name" column, the name in capital letters (e.g., REMOTING_CLIENT_AUTH_MODE) is a handy constant in the `SSLSocketBuilder` class with the value given in parentheses (e.g. "org.jboss.remoting.clientAuthMode"). It is the latter that should be used for declarative configuration in, for example, MBean descriptors.

Table 5.1. `SSLSocketBuilder` configuration parameters.

attribute	key name	type	default	description
ClientAuthMode	REMOTING_CLIENT_AUTH_MODE ("org.jboss.remoting.clientAuthMode")	String	need	Determines if sockets need or want client authentication. This configuration option is only useful for sockets in the server mode. Value may be "none", "want", or "need".
KeyAlias	REMOTING_KEY_ALIAS ("org.jboss.remoting.keyAlias")	String		The preferred identity in key store to be used by key managers
KeyPassword	REMOTING_KEY_PASSWORD ("org.jboss.remoting.keyPassword")	String		Sets the password to use for the keys within the key store. This only needs to be set if `setUseSSLServerSocketFactory()` is set to false (otherwise will be ignored). If this value is not set, but the key store password is, it will use that value for the

attribute	key name	type	default	description
				key password.
KeyStoreAlgorithm	REMOTING_KEY_STORE_ALGORITHM ("org.jboss.remoting.keyStoreAlgorithm")	String	SunX509	The algorithm for the key manager factory.
KeyStorePassword	REMOTING_KEY_STORE_PASSWORD ("org.jboss.remoting.keyStorePassword")	String		The password to use for the key store. This only needs to be set if `setUseSSLServerSocketFactory()` is set to false (otherwise will be ignored). The value passed will also be used for the key password if the latter attribute is not explicitly set.
KeyStoreType	REMOTING_KEY_STORE_TYPE ("org.jboss.remoting.keyStoreType")	String	JKS	The type to be used for the key store. Some acceptable values are JKS (Java Keystore - Sun's keystore format), JCEKS (Java Cryptography Extension keystore - More secure version of JKS), and PKCS12 (Public-Key Cryptography Standards #12 keystore - RSA's Personal Information Exchange Syntax Standard). These are not case sensitive.

attribute	key name	type	default	description
KeyStoreURL	REMOTING_KEY_STORE_FILE_PATH ("org.jboss.remoting.keyStore")	String		Property used to define where `SSLSocketBuilder` will look for the keystore file. This can be relative to the thread's class-loader or can be an absolute path on the file system or can be a URL.
Provider	none	java.security.Provider		Java Security API implementation to use.
ProviderName	REMOTING_SSL_PROVIDER_NAME ("org.jboss.remoting.sslProviderName")	String		Name of Java Security API implementation to use.
SecureRandom	none	java.security.SecureRandom	`new SecureRandom()`	Random number generator to use.
SecureSocketProtocol	REMOTING_SSL_PROTOCOL ("org.jboss.remoting.sslProtocol")	String	TLS	The protocol for the `SSLContext`. Some acceptable values are TLS, SSL, and SSLv3
ServerAuthMode	REMOTING_SERVER_AUTH_MODE ("org.jboss.remoting.serverAuthMode")	boolean/String	true	Determines if a client should attempt to authenticate a server certificate as one it trusts.
ServerSocketUseClientMode	REMOTING_SERVER_SOCKET_USE_CLIENT_MODE	boolean/String	false	Determines if the server sockets will be in client or server mode.

attribute	key name	type	default	description
	("org.jboss.remoting.serversocket.useClientMode")			
SocketUseClientMode	REMOTING_SOCKET_USE_CLIENT_MODE ("org.jboss.remoting.socket.useClientMode")	boolean/String	true	Determines if the sockets will be in client or server mode.
TrustStoreAlgorithm	REMOTING_TRUST_STORE_ALGORITHM ("org.jboss.remoting.trustStoreAlgorithm")	String	value of `KeyStoreAlgorithm`, or SunX509 if `KeyStoreAlgorithm` is not set	trust store key management algorithm
TrustStorePassword	REMOTING_TRUST_STORE_PASSWORD ("org.jboss.remoting.trustStorePassword")	String		trust store password
TrustStoreType	REMOTING_TRUST_STORE_TYPE ("org.jboss.remoting.trustStoreType")	String	value of `KeyStoreType`, or JKS if `KeyStoreType` is not set	type of trust store
TrustStoreURL	REMOTING_TRUST_STORE_FILE_PATH ("org.jboss.remoting.trustStore")	String		location of trust store
UseSSLServerSocketFactory	none	boolean	true	Determines if default `SSLServerSocketFactory` should be created.
UseSSLSocketFact-	none	boolean	true	Determines if de-

attribute	key name	type	default	description
ory				fault SSLSocket-Factory should be created.

Note. If any of the attributes KeyStoreURL, KeyStorePassword, KeyStoreType, TrustStoreURL, TrustStorePassword, or TrustStoreType are left unconfigured, SSLSocketBuilder will also examine the corresponding standard SSL system properties "javax.net.ssl.keyStore", "javax.net.ssl.keyStorePassword", "javax.net.ssl.keyStoreType", "javax.net.ssl.trustStore", "javax.net.ssl.trustStorePassword", "javax.net.ssl.trustStoreType". In the cases of KeyStoreType and TrustStoreType, SSLSocketBuilder will then go on to use default values after checking the system properties.

The following is an example of configuring a SSLSocketBuilder and using it to create a custom SSLSocketFactory:

```
protected SSLSocketFactory getSocketFactory() throws Exception
{
   HashMap config = new HashMap();
   config.put(SSLSocketBuilder.REMOTING_KEY_STORE_TYPE, "JKS");
   String keyStoreFilePath = getKeystoreFilePath();
   config.put(SSLSocketBuilder.REMOTING_KEY_STORE_FILE_PATH, keyStoreFilePath);
   config.put(SSLSocketBuilder.REMOTING_KEY_STORE_PASSWORD, "unit-tests-server");
   config.put(SSLSocketBuilder.REMOTING_SSL_PROTOCOL, "SSL");
   SSLSocketBuilder builder = new SSLSocketBuilder(config);
   builder.setUseSSLSocketFactory(false);
   return builder.createSSLSocketFactory();
}
```

More examples of configuring SSLSocketBuilder can be found in the class FactoryConfigSSLSample in the package org.jboss.remoting.samples.config.factories.

The following is an example of configuring SSLSocketBuilder in a *-service.xml file:

```
   <!-- This service is used to build the SSL Server socket factory -->
   <!-- This will be where all the store/trust information will be set. -->
   <!-- If do not need to make any custom configurations, no extra attributes -->
   <!-- need to be set for the SSLSocketBuilder and just need to set the -->
   <!-- javax.net.ssl.keyStore and javax.net.ssl.keyStorePassword system properties. -->
   <!-- This can be done by just adding something like the following to the run -->
   <!-- script for JBoss -->
   <!-- (this one is for run.bat): -->
   <!-- set JAVA_OPTS=-Djavax.net.ssl.keyStore=.keystore -->
   <!-- -Djavax.net.ssl.keyStorePassword=opensource %JAVA_OPTS% -->
   <!-- Otherwise, if want to customize the attributes for SSLSocketBuilder, -->
   <!-- will need to uncomment them below. -->
   <mbean code="org.jboss.remoting.security.SSLSocketBuilder"
          name="jboss.remoting:service=SocketBuilder,type=SSL"
          display-name="SSL Server Socket Factory Builder">
      <!-- IMPORTANT - If making ANY customizations, this MUST be set to false. -->
      <!-- Otherwise, will used default settings and the following attributes will be ignored. -
      <attribute name="UseSSLServerSocketFactory">false</attribute>
      <!-- This is the url string to the key store to use -->
      <attribute name="KeyStoreURL">.keystore</attribute>
```

```
            <!-- The password for the key store -->
            <attribute name="KeyStorePassword">opensource</attribute>
            <!-- The password for the keys (will use KeystorePassword if this is not set explicitly. -
            <attribute name="KeyPassword">opensource</attribute>
            <!-- The protocol for the SSLContext. Default is TLS. -->
            <attribute name="SecureSocketProtocol">TLS</attribute>
            <!-- The algorithm for the key manager factory. Default is SunX509. -->
            <attribute name="KeyManagementAlgorithm">SunX509</attribute>
            <!-- The type to be used for the key store. -->
            <!-- Defaults to JKS. Some acceptable values are JKS (Java Keystore - Sun's keystore forma
            <!-- JCEKS (Java Cryptography Extension keystore - More secure version of JKS), and -->
            <!-- PKCS12 (Public-Key Cryptography Standards #12 keystore - RSA's Personal Information E
            <!-- These are not case sensitive. -->
            <attribute name="KeyStoreType">JKS</attribute>
         </mbean>
```

It is also possible to set the default socket factory to be used when not using customized settings (meaning UseSSLSocketFactory property value is true, which is the default). This can be done by setting system property of org.jboss.remoting.defaultSocketFactory to the fully qualified class name of the javax.net.SocketFactory implementation to use. Will then call the getDefault() method on that implementation to get the SocketFactory instance to use.

5.7.8. SSLServerSocketFactoryService

Although any server socket factory can be set for the various transports, there is a customizable server socket factory service provided within JBossRemoting that supports SSL. This is the `org.jboss.remoting.security.SSLServerSocketFactoryService` class. The `SSLServerSocketFactoryService` class extends the `javax.net.ServerSocketFactory` class and also implements the `SSLServerSocketFactoryServiceMBean` interface (so that it can be set using the `socketServerFactory` attribute described previously). Other than providing the proper interfaces, this class is a simple wrapper around the `org.jboss.remoting.security.SSLSocketBuilder` class.

The following is an example of configuring `SSLServerSocketFactoryService` in a *-service.xml file. Note that it depends on the `SSLSocketBuilder` MBean defined in the xml fragment above:

```
      <!-- This service provides the exact same API as the ServerSocketFactory, so -->
      <!-- can be set as an attribute of that type on any MBean requiring an ServerSocketFactory. -->
      <mbean code="org.jboss.remoting.security.SSLServerSocketFactoryService"
            name="jboss.remoting:service=ServerSocketFactory,type=SSL"
            display-name="SSL Server Socket Factory">
         <depends optional-attribute-name="SSLSocketBuilder"
            proxy-type="attribute">jboss.remoting:service=SocketBuilder,type=SSL</depends>
      </mbean>
```

5.7.9. General Security How To

Since we are talking about keystores and truststores, this section will quickly go over how to quickly generate a test keystore and truststore for testing. This is not intended to be a full security overview, just an example of how I originally created mine for testing.

To get started, will need to create key store and trust store.

Generating key entry into keystore:

```
C:\tmp\ssl>keytool -genkey -alias remoting -keyalg RSA
Enter keystore password: opensource
What is your first and last name?
[Unknown]: Tom Elrod
What is the name of your organizational unit?
[Unknown]: Development
What is the name of your organization?
[Unknown]: JBoss Inc
What is the name of your City or Locality?
[Unknown]: Atlanta
What is the name of your State or Province?
[Unknown]: GA
What is the two-letter country code for this unit?
[Unknown]: US
Is CN=Tom Elrod, OU=Development, O=JBoss Inc, L=Atlanta, ST=GA, C=US correct?
[no]: yes

Enter key password for <remoting>
(RETURN if same as keystore password):
```

Since did not specify the -keystore filename parameter, created the keystore in $HOME/.keystore (or C:\Documents and Settings\Tom\.keystore).

Export the RSA certificate (without the private key)

```
C:\tmp\ssl>keytool -export -alias remoting -file remoting.cer
Enter keystore password: opensource
Certificate stored in file <remoting.cer>
```

Import the RSE certificate into a new truststore file.

```
C:\tmp\ssl>keytool -import -alias remoting -keystore .truststore -file remoting.cer
Enter keystore password: opensource
Owner: CN=Tom Elrod, OU=Development, O=JBoss Inc, L=Atlanta, ST=GA, C=US
Issuer: CN=Tom Elrod, OU=Development, O=JBoss Inc, L=Atlanta, ST=GA, C=US
Serial number: 426f1ee3
Valid from: Wed Apr 27 01:10:59 EDT 2005 until: Tue Jul 26 01:10:59 EDT 2005
Certificate fingerprints:
MD5: CF:D0:A8:7D:20:49:30:67:44:03:98:5F:8E:01:4A:6A
SHA1: C6:76:3B:6C:79:3B:8D:FD:FB:4F:33:3B:25:C9:01:9D:50:BF:9F:8A
Trust this certificate? [no]: yes
Certificate was added to keystore
```

Now have two files, .keystore for the server and .truststore for the client.

5.7.10. Troubleshooting Tips

Common errors when using server socket factory:

```
javax.net.ssl.SSLException: No available certificate corresponds to the SSL cipher suites which are enabl
```

The 'javax.net.ssl.keyStore' system property has not been set and are using the default SSLServerSocketFactory.

```
java.net.SocketException: Default SSL context init failed: Cannot recover key
```

The 'javax.net.ssl.keyStorePassword' system property has not been set and are using the default SSLServerSocket-Factory.

```
java.io.IOException: Can not create SSL Server Socket Factory due to the url to the key store not being s
```

The default SSLServerSocketFactory is NOT being used (so custom configuration for the server socket factory) and the key store url has not been set.

```
java.lang.IllegalArgumentException: password can't be null
```

The default SSLServerSocketFactory is NOT being used (so custom configuration for the server socket factory) and the key store password has not been set.

5.8. Timeouts

The handling of timeouts in Remoting is surveyed in this section. On the whole, timing out network connections is handled differently by each transport, but there are some transport independent methods for timeout configuration, extended by some transport specific methods.

5.8.1. General timeout configuration

As with all configuration parameters, there are several avenues for specifying parameter values. See Section General transport configuration for a general discussion of parameter configuration in Remoting. The transport independent key for setting timeouts is "timeout", also available as `org.jboss.remoting.ServerInvoker.TIMEOUT`. All server invokers also have the getter/setter methods

```
public int getTimeout();

public void setTimeout(int timeout);
```

where the values are given in milliseconds. The default timeout value is 60000 for server invokers.

5.8.2. Per invocation timeouts

Beginning with release 2.2.0, some Remoting transports offer a per invocation transport facility, which allows a timeout value to be set for a particular invocation, overriding the client invoker's previously configured timeout value. The per invocation timeout is set by passing the `String` representation of the timeout value in the invocation's metadata map, using the key "timeout". For example,

```
HashMap metadata = new HashMap();
metadata.put("timeout", "2000");
client.invoke("testInvocation", metadata);
```

will allow approximately 2 seconds for this particular invocation, after which the timeout value will be reset to its previously configured value.

Each transport that supports per invocation timeouts handles them a little differently. More details are given below.

5.8.3. Transport specific timeout handling

5.8.3.1. Socket and bisocket transports

These two transports are handled together because bisocket inherits most of its timeout handling from socket. The discussion also applies to their SSL versions, sslbisocket and sslsocket. On the server side, the timeout value, whatever the source of its value, is used to set the timeout value of all `Sockets` managed by the server invoker's worker threads. On the client side, the configured timeout value is used to limit the time required by `Socket.connect()` when a new `Socket` is created, as well as to set the `Socket` timeout value for all connections in its connection pool.

The socket and bisocket transports support per invocation timeouts. The processing subject to the timeout period starts when the client invoker begins to acquire a network connection and extends to the point at which it begins reading the reponse to the invocation. Note that the acquisition of the network connection might involve multiple attempts to connect to the server.

5.8.3.1.1. Write timeouts

The socket timeout facility offered by the JDK applies only to `read` operations on the socket. As of release 2.5.2, the socket and bisocket (and also sslsocket and sslbisocket) transports offer a **write timeout** facility. When a client or server is configured, in any of the usual ways, with the parameter `org.jboss.remoting.transport.socket.SocketWrapper.WRITE_TIMEOUT` (actual value "writeTimeout") set to a positive value (in milliseconds), all write operations will time out if they do not complete within the configured period. When a write operation times out, the socket upon which the write was invoked will be closed, which is likely to result in a `java.net.SocketException`.

Note. A `SocketException` is considered to be a "retriable" exception, so, if the parameter "numberOfCallRetries" is set to a value greater than 1, an invocation interrupted by a write timeout can be retried.

Note. The write timeout facility applies to writing of both invocations and responses. It applies to push callbacks as well.

5.8.3.2. HTTP transport

The http server invoker looks for a configured timeout value at initialization time, which it uses to set the "connectionTimeout" property on its Tomcat connector. (See Section HTTP Invoker for more information.) Note that subsequent calls to `setTimeout()` will have no effect.

The http client invoker treats timeouts configured for the connection and per invocation timeouts the same, since it opens a new `HttpURLConnection` with each invocation. Any nonnegative per invocation timeout value will override a timeout value configured at client invoker creation time.

If the application is using a jdk of generation 1.5 or later, then the client invoker will use the `java.net.HttpURLConnection` methods `setConnectTimeout()` and `setReadTimeout()` methods. Note that in this

case the timeout value will be allowed twice, once to create the connection and once to read the invocation result.

If an earlier jdk is being used, the client invoker will simulate a timeout by making the connection and executing the invocation in a separate thread, which it waits on for the specified timeout. The threads are drawn from a thread pool, which is configurable. A custom thread pool may be set by calling the `HTTPClientInvoker` method

```
public void setTimeoutThreadPool(org.jboss.util.threadpool.ThreadPool pool);
```

where the `ThreadPool` interface is available from the anonomous JBoss svn repository at http://anonsvn.jboss.org/repos/common/common-core/trunk/src/main/java/ [http://anonsvn.jboss.org/repos/common/common-core/trunk/src/main/java]. If a thread pool is not set, it will default to an instance of `org.jboss.util.threadpool.BasicThreadPool`, which may be configured with the following parameters, defined as constants in `org.jboss.remoting.transport.http.HTTPClientInvoker`:

MAX_NUM_TIMEOUT_THREADS (actual value "maxNumTimeoutThreads"): the number of threads in the threadpool. The default value is 10.

MAX_TIMEOUT_QUEUE_SIZE (actual value "maxTimeoutQueueSize"): the size of the thread pool queue, which holds execution requests when all of the threads are in use. The default value is 1024.

5.8.3.3. Quick client disconnect

`org.jboss.remoting.Client` applies per invocation timeouts in its `removeListener()` and `disconnect()` methods to create a "quick disconnect" facility. If, for example, an `org.jboss.remoting.ConnectionValidator` (see Network Connection Monitoring) reports its suspicion that a connection is failing, the application might want to restrict, or even eliminate, the time spent trying to access the network while unregistering callback handlers and disconnecting. The quick disconnect facility is invoked by calling the `Client` method

```
public void setDisconnectTimeout(int disconnectTimeout);
```

to set the disconnect timeout value to a nonnegative value. If the disconnect timeout value is set, it will be applied as the per invocation timeout value for all network activity in the methods `removeListener()` and `disconnect()`. As a special case, if the disconnect timeout value is set to 0, `Client` will simply skip any network i/o in these two methods.

5.9. Configuration by properties

This section covers configuration properties by constant values and bean properties for individual classes. This will duplicate some of the configuration properties already covered and is just another view to some of the same information.

org.jboss.remoting.InvokerLocator

SERVER_BIND_ADDRESS (actual value is 'jboss.bind.address') - indicates the system property key for bind address that should be used.

BIND_BY_HOST (actual value is 'remoting.bind_by_host') - indicates the system property key for if the local bind address should be by host name (e.g. InetAddress.getLocalHost().getHostName()) or if should be by IP (e.g. InetAddress.getLocalHost().getHostAddress()). The default is 'True', meaning will will use local host name. This configuration only applies when the initial bind address is 0.0.0.0 (or InvokerLocator.ANY).

DATATYPE (actual value is 'datatype') - indicates the marshalling datatype that should be used for a particular invoker. Each invoker has its own default marshaller and unmarshaller based on default datatype. For examle, the socket transport has a default datatype of 'serializable', which is automatically registered with the MarshalFactory and associated by default with org.jboss.remoting.marshal.serializable.SerializableMarshaller and org.jboss.remoting.marshal.serializable.SerializableUnMarshaller. The marshaller and unmarshaller used by an invoker can be overriden by setting the 'datatype' parameter within the LocatorInvoker. For example, could use a locator url of:

```
socket://myhost:6500/?datatype=test
```

which would cause the socket invoker to use the marshaller and unmarshaller registered with the MarshalFactory under the datatype 'test'. Of course, this requires that the marshaller and unmarshaller implementations to be used have already been registered with the MarshalFactory (otherwise will get an exception).

SERIALIZATIONTYPE (actual value is 'serializationtype') - indicates the serialization implementation to use. Currently, the only possible values are 'java' and 'jboss'. Java serialization is the default. Setting to 'jboss' will cause JBoss Serialization to be used. In implementation, this equates to the parameter that will be passed to the SerializationStreamFactory.getManagerInstance() method. This configuration can be set as an invoker locator url parameter (e.g. socket://myhost:5400/?serializationtype=jboss) or as an entry to the configuration Map passed when constructing a remoting client or server.

MARSHALLER (actual value is 'marshaller') - used to indicate which marshaller implementation should be used by the invoker. This is an override for whatever the invoker's default implementation is. This can be set as a parameter of the invoker locator url (e.g. socket://myhost:6500/?marshaller=org.jboss.test.remoting.marshall.dynamic.remote.http.TestMarshaller). Using this configuration requires that the value be the fully qualified classname of the marshaller implementation to use (which must be on the classpath, have a void constructor, and implement the org.jboss.remoting.marshal.Marshaller interface).

UNMARSHALLER (actual value is 'unmarshaller') - used to indicate which unmarshaller implementation should be used by the invoker. This is an override for whatever the invoker's default implementation is. This can be set as a parameter of the invoker locator url (e.g. socket://myhost:6500/?unmarshaller=org.jboss.test.remoting.marshall.dynamic.remote.http.TestUnMarshaller). Using this configuration requires that the value be the fully qualified classname of the unmarshaller implementation to use (which must be on the classpath, have a void constructor, and implement the org.jboss.remotng.marshal.UnMarshaller interface).

LOADER_PORT (actual value is 'loaderport') - indicates the port number where the class loader server resides. This can be used when is possible that a client may not have particular classes locally and would want to load them from the server dynamically. This property can be set as a parameter to the invoker locator url. A clasic example of when this might be used would be in conjunction with using custom marshalling. For example, if have configured a server to use custom marshaller and unmarshaller that the client will not have access to, could create a invoker locator such as:

```
socket://myhost:6500/?datatype=test&loaderport=6501&
```

```
marshaller=org.jboss.test.remoting.marshall.dynamic.remote.http.TestMarshaller&
unmarshaller=org.jboss.test.remoting.marshall.dynamic.remote.http.TestUnMarshaller
```

When the client invoker begins to make an invocation, will try to look up marshaller and unmarshaller based on type ('test' in this case) and when can not find a registry entry for it, will try to load the TestMarshaller and TestUnMarshaller from the classpath. When the classes can not be found locally, will make a call to the loader server (on port 6501) to load the classes locally. Once they are retrieved from the server, will be registered locally, so is a one time only event (as next time will be found in the registry).

This can work for loading any remote server classes, but requires the loaderport be included in the invoker locator url.

BYVALUE (actual value is 'byvalue') - indicates if when making local invocations (meaning client and server invoker exists within same jvm), the marshalling will be done by value, instead of the default, by reference. Using this configuration, the marshalling will actually perform a clone of the object instance (see org.jboss.remoting.serialization.SerializationManager.createMarshalledValueForClone()). Value for this property should be of type String and be either 'true' or 'false'. In releases prior to 2.0.0, using this configuration setting would have forced invokers to be remote, which can now be done via FORCE_REMOTE config (see below).

FORCE_REMOTE (actual value is 'force_remote') - indicates if when making local invocations (meaning client and server invoker exists within same jvm), the remote invokers should be used instead of local invoker. Is equivalent to making invocations as though client and server were in different jvms). Value for this property should be of type String and be either 'true' or 'false'.

CLIENT_LEASE (actual value is 'leasing') - indicates if client should try to automatically establish a lease with the server. Is false by default. Value for this property should be of type String and be either 'true' or 'false'.

CLIENT_LEASE_PERIOD (actual value is 'lease_period') - defines what the client lease period should be in the case that server side leasing is turned on. Value for this parameter key should be the number of milliseconds to wait before each client lease renewal and must be greater than zero in order to be recognized. If this property is not set (and CLIENT_LEASE is), will use the lease period as specified by the server.

org.jboss.remoting.Client

RAW (actual value is 'rawPayload') - key to use for the metadata Map passed when making an invoke() call and wish for the invocation payload to be sent as is and not wrapped within a remoting invocation request object. This should be used when want to make direct calls on systems outside of remoting (e.g. making an http POST request to a web service).

ENABLE_LEASE (actual value is 'enableLease') - key for the configuration map passed to the Client constructor to indicate that client should make initial request to establish lease with server. The value for this should be either a String that java.lang.Boolean can evaluate or a java.lang.Boolean. By default, leasing is turned off, so this property would be used to turn on leasing for the client.

HANDSHAKE_COMPLETED_LISTENER (actual value is 'handshakeCompletedListener') - key for the configuration map passed to the Client constructor providing a ssl javax.net.ssl.HandshakeCompletedListener implementation, which will be called on when ssl handshake completed with server.

USE_ALL_PARAMS (actual value is 'useAllParameters') - used by `org.jboss.remoting.ConnectionValidator`

and `org.jboss.remoting.callback.CallbackPoller` to determine if configuration parameters should be retrieved from the `InvokerLocator` or the `Client`'s configuration map.

The following three configuration properties are only useful when using one of the following Client methods:

```
public void addListener(InvokerCallbackHandler callbackhandler, Map metadata, Object callbackHandlerObjec
public void addListener(InvokerCallbackHandler callbackhandler, Map metadata, Object callbackHandlerObjec
```

CALLBACK_SERVER_PROTOCOL (actual value is 'callbackServerProtocol') - key for the configuration when adding a callback handler and internal callback server connector is created. The value should be the transport protocol to be used. By default will use the same protocol as being used by this client (e.g. http, socket, rmi, multiplex, etc.).

CALLBACK_SERVER_HOST (actual value is 'callbackServerHost') - key for the configuration when adding a callback handler and internal callback server connector is created. The value should be the host name to be used. By default will use the result of calling InetAddress.getLocalHost().getHostAddress().

CALLBACK_SERVER_PORT (actual value is 'callbackServerPort') - key for the configuration when adding a callback handler and internal callback server connector is created. The value should be the port to be used. By default will find a random unused port.

Bean properties (meaning have getter/setter):

SessionId - session id used when making invocations on server invokers. There is a default unique id automatically generated for each Client instance, so unless you have a good reason to set this, do not set this.

Subsystem - the subsystem being used when routing invocation requests on the server side. Specifing a subsystem is only needed when server has multiple handlers registered (which will each have their own associated subsystem). Best if specified using Client constructor.

MaxNumberOfThreads - the maximum number of threads to use within client pool for one way invocations on the client side (meaning oneway invocation is handled by thread in this pool and user's call returns immediately) Default value is MAX_NUM_ONEWAY_THREADS (whose value is 10).

OnewayThreadPool - the thread pool being used for making one way invocations on the client side. If one has not been specifically set via configuration or call to set it, will always return instance of org.jboss.util.threadpool.BasicThreadPool.

SocketFactory - instance of javax.net.SocketFactory, which can only be set on the Client before the connect() method has been called. Otherwise, a runtime exception will be thrown.

Marshaller - the marshaller implementation that should be used by the client invoker (transport). This overrides the client's default marshaller (or any set within configuration).

UnMarshaller - the unmarshaller implementation that should be used by the client invoker (transport). This overrides the client's default unmarshaller (or any set within configuration).

org.jboss.remoting.Remoting

CUSTOM_SERVER_SOCKET_FACTORY (actual value is 'customServerSocketFactory') - key for the configuration map passed to a Connector to indicate the server socket factory to be used. This will override the creation of any other socket factory. Value must be an instance of javax.net.ServerSocketFactory.

CUSTOM_SOCKET_FACTORY (actual value is 'customSocketFactory') - key for the configuration map passed to a Client to indicate the socket factory to be used. Value must be instance of javax.net.SocketFactory.

SOCKET_FACTORY_CLASS_NAME (actual value is 'socketFactoryClassName') - key for the configuration map passed to a Client to indicate the classname of the socket factory to be used. Value should be fully qualified classname of class that is an instance of javax.net.SocketFactory and has a void constructor. This property will not be used if CUSTOM_SOCKET_FACTORY is also set.

USE_CLIENT_CONNECTION_IDENTITY (actual value "useClientConnectionIdentity") - tells Remoting to adhere to the new "connection identity" semantics described in Chapter Network Connection Monitoring.

PASS_CONFIG_MAP_TO_MARSHAL_FACTORY (actual value "passConfigMapToMarshalFactory") - if set to "true", tells clients and servers to take parameters for configuring marshallers and unmarshallers from the configuration map passed to `Clients` and `Connectors` in addition to the parameters in the `InvokerLocator`.

org.jboss.remoting.ServerInvoker

MAX_NUM_ONEWAY_THREADS_KEY (actual value is 'maxNumThreadsOneway') - key for the maximum number of threads to be used in the thread pool for one way invocations (server side). This property is only used when the default oneway thread pool is used.

ONEWAY_THREAD_POOL_CLASS_KEY (actual value is 'onewayThreadPool') - key for setting the setting the oneway thread pool to use. The value used with this key will first be checked to see if is a JMX ObjectName and if so, try to look up associated mbean for the ObjectName given and cast to type org.jboss.util.threadpool.ThreadPoolMBean (via MBeanServerInvocationHandler.newProxyInstance()). If the value is not a JMX ObjectName, will assume is a fully qualified classname and load the coresponding class and create a new instance of it (which will require it to have a void constructor). The newly created instance will then be cast to type of org.jboss.util.threadpool.ThreadPool.

SERVER_BIND_ADDRESS_KEY (actual value is 'serverBindAddress') - key for setting the address the server invoker should bind to. The value can be either host name or IP.

CLIENT_CONNECT_ADDRESS_KEY (actual value is 'clientConnectAddress') - key for setting the address the client invoker should connecto to. This should be used when client will be connecting to server from outside the server's network and the external address is different from that of the internal address the server invoker will bind to (e.g. using NAT to expose different external address). This will mostly be useful when client uses remoting detection to discover remoting servers. The value can be either host name or IP.

SERVER_BIND_PORT_KEY (actual value is 'serverBindPort') - key for setting the port the server invoker should bind to. If the value supplied is less than or equal to zero, the server invoker will randomly choose a free port to use.

CLIENT_CONNECT_PORT_KEY (actual value is 'clientConnectPort') - key for setting the port the client invoker should connect to. This should be used when client will be connecting to server from outside the server's network and the external port is different from that of the internal port the server invoker will bind to (e.g. using NAT

to expose different port routing). This will be mostly useful when client uses remoting detection to discover remoting servers.

CLIENT_LEASE_PERIOD (actual value is 'clientLeasePeriod') - key used for setting the amount of time (in milliseconds) that a client should renew its lease. If this value is not set, the default of five seconds (see DEFAULT_CLIENT_LEASE_PERIOD), will be used. This value will also be what is given to the client when it initially queries server for leasing information.

TIMEOUT (actual value is 'timeout') - key for setting the timeout value (in milliseconds) for socket connections.

SERVER_SOCKET_FACTORY (actual value is 'serverSocketFactory') - key for setting the value for the server socket factory to be used by the server invoker. The value can be either a JMX Object name, in which case will lookup the mbean and create a proxy to it with type of org.jboss.remoting.security.ServerSocketFactoryMBean (via MBeanServerInvocationHandler.newProxyInstance()), or, if not a JMX ObjectName, will assume is the fully qualified classname to the implementation to be used and will load the class and create a new instance of it (which requires it to have a void constructor). The instance will then be cast to type javax.net.ServerSocketFactory.

BLOCKING_MODE (actual value is "blockingMode"): if set to `ServerInvoker.BLOCKING` (actual value "blocking"), `org.jboss.remoting.Client.getCallbacks()` will do blocking pull callbacks and `CallbackPoller` will do blocking polled callbacks; if set to `ServerInvoker.NONBLOCKING` (actual value "nonblocking"), `Client.getCallbacks()` will do non-blocking pull callbacks and `CallbackPoller` will do non-blocking polled callbacks.

BLOCKING_TIMEOUT (actual value is "blockingTimeout"): the timeout value used for blocking callback.

REGISTER_CALLBACK_LISTENER (actual value is "registerCallbackListener"): determines if `org.jboss.remoting.callback.ServerInvokerCallbackHandlers` should register as `org.jboss.remoting.ConnectionListeners` with leases. The default value is "true".

Bean properties (meaning have getter/setter):

ServerSocketFactory - implementation of javax.net.ServerSocketFactory to be used by the server invoker. This takes precedence over any other configuration for the server socket factory.

Timeout - timeout (in milliseconds) for socket connection. If set after create() method called, this value will override value set by TIMEOUT key.

LeasePeriod - the amount of time (in milliseconds) that a client should renew its lease. If this value is not set, the default of five seconds (see DEFAULT_CLIENT_LEASE_PERIOD), will be used. This value will also be what is given to the client when it initially queries server for leasing information. If set after create() method called, this value will override value set by CLIENT_LEASE_PERIOD key.

MaxNumberOfOnewayThreads - the maximum number of threads to be used in the thread pool for one way invocations (server side). This property is only used when the default oneway thread pool is used. If set after create() method called, this value will override value set by MAX_NUM_ONEWAY_THREADS_KEY key.

OnewayThreadPool - the oneway thread pool to use.

org.jboss.remoting.callback.CallbackPoller

CALLBACK_POLL_PERIOD (actual value is 'callbackPollPeriod') - key for setting the frequency (in milliseconds) in which Client's internal callback poller should poll server for waiting callbacks. The default value is five seconds.

CALLBACK_SCHEDULE_MODE (actual value is "scheduleMode"): may be set to either `CallbackPoller.SCHEDULE_FIXED_RATE` (actual value "scheduleFixedRate") or `CallbackPoller.SCHEDULE_FIXED_DELAY` (actual value "scheduleFixedDelay"). In either case, polling will take place at approximately regular intervals, but in the former case the scheduler will attempt to perform each poll CALLBACK_POLL_PERIOD milliseconds after the previous attempt, and in the latter case the scheduler will attempt to schedule polling so that the *average* interval will be approximately CALLBACK_POLL_PERIOD milliseconds. `CallbackPoller.SCHEDULE_FIXED_RATE` is the default.

REPORT_STATISTICS (actual value is "reportStatistics"): The presence of this key in `metadata`, regardless of its value, will cause the `CallbackPoller` to print statistics that might be useful for configuring the other parameters..

`CallbackPoller` configuration is only necessary when using one of the following Client methods:

```
public void addListener(InvokerCallbackHandler callbackhandler, Map metadata, Object callbackHandlerObjec
public void addListener(InvokerCallbackHandler callbackhandler, Map metadata, Object callbackHandlerObjec
```

The keys should be among the entries in the metadata Map passed. This will also only apply when the underlying transport is uni-directional (e.g. socket, http, rmi). Bi-directional transports will not need to poll.

org.jboss.remoting.callback.CallbackStore

FILE_PATH_KEY (actual value is 'StoreFilePath') - key for setting the directory in which to write the callback objects. The default value is the property value of 'jboss.server.data.dir' and if this is not set, then will be 'data'. Will then append 'remoting' and the callback client's session id. An example would be 'data\remoting\5c4o05l-9jijyx-e5b6xyph-1-e5b6xyph-2'.

FILE_SUFFIX_KEY (actual value is 'StoreFileSuffix') - key for setting the file suffix to use for the callback objects written to disk. The default value is 'ser'.

org.jboss.remoting.callback.DefaultCallbackErrorHandler

CALLBACK_ERRORS_ALLOWED (actual value is 'callbackErrorsAllowed') - key for setting the number of callback exceptions that will be allowed when calling on org.jboss.remoting.callback.InvokerCallbackHandler.handleCallback(Callback callback) before cleaning up the callback listener. This only applies to push callback. The default if this property is not set is five.

org.jboss.remoting.callback.ServerInvokerCallbackHandler

CALLBACK_STORE_KEY (actual value is 'callbackStore') - key for specifing the callback store to be used. The value can be either a JMX ObjectName or a fully qualified class name; either way, must implement org.jboss.remoting.SerializableStore. If using class name, the callback store implementation must have a void con-

structor. The default is to use the org.jboss.remoting.callback.NullCallbackStore.

CALLBACK_ERROR_HANDLER_KEY (actual value is 'callbackErrorHandler') - key for specifing the callback exception handler to be used. The value can be either a JMX ObjectName or a fully qualified class name, either way, must implement org.jboss.remoting.callback.CallbackErrorHandler. If using class name, the callback exception handler implementation must have a void constructor. The default is to use org.jboss.remoting.callback.DefaultCallbackErrorHandler.

CALLBACK_MEM_CEILING (actual value is 'callbackMemCeiling') - key for specifying the percentage of free memory available before callbacks will be persisted. If the memory heap allocated has reached its maximum value and the percent of free memory available is less than the callbackMemCeiling, this will trigger persisting of the callback message. The default value is 20.

org.jboss.remoting.detection.jndi.JNDIDetector

Bean properties (meaning have getter/setter):

SubContextName - sub context name under which detection messages will be bound and looked up.

org.jboss.remoting.transport.bisocket.Bisocket

IS_CALLBACK_SERVER (actual value is "isCallbackServer"): when a bisocket server invoker receives this parameter with a value of true, it avoids the creation of a `ServerSocket`. Therefore, IS_CALLBACK_SERVER should be used on the client side for the creation of a callback server. The default value is false.

PING_FREQUENCY (actual value is "pingFrequency"): The server side uses this value to determine the interval, in milliseconds, between pings that it will send on the control connection. The client side uses this value to calculate the window in which it must receive pings on the control connection. In particular, the window is ping frequency * ping window factor. See also the definition of PING_WINDOW_FACTOR. The default value is 5000.

PING_WINDOW_FACTOR (actual value is "pingWindowFactor"): The client side uses this value to calculate the window in which it must receive pings on the control connection. In particular, the window is ping frequency * ping window factor. See also the definition of PING_FREQUENCY. The default value is 2.

MAX_RETRIES (actual value is "maxRetries"): This parameter is relevant only on the client side, where the `BisocketClientInvoker` uses it to govern the number of attempts it should make to get the address and port of the secondary `ServerSocket`, and the `BisocketServerInvoker` uses it to govern the number of attempts it should make to create both ordinary and control sockets. The default value is 10.

MAX_CONTROL_CONNECTION_RESTARTS (actual value is "maxControlConnectionRestarts"): The client side uses this value to limit the number of times it will request a new control connection after a ping timeout. The default value is 10.

SECONDARY_BIND_PORT (actual value is "secondaryBindPort"): The server side uses this parameter to determine the bind port for the secondary `ServerSocket`.

SECONDARY_CONNECT_PORT (actual value is "secondaryConnectPort"): The server side uses this parameter to determine the connect port used by the client side to connect to the secondary `ServerSocket`.

org.jboss.remoting.transport.http.HTTPMetadataConstants

The following are keys to use to get corresponding values from the Map returned from call to org.jboss.remoting.InvocationRequest.getRequestPayload() within a org.jboss.remoting.ServerInvocationHandler implementation. For example:

```
   public Object invoke(InvocationRequest invocation) throws Throwable
   {
      Map headers = invocation.getRequestPayload();
```

where variable 'headers' will contain entries for the following keys.

METHODTYPE (actual value is 'MethodType') - key for getting the method type used by client in http request. This will be populated within the Map returned from call to org.jboss.remoting.InvocationRequest.getRequestPayload() within a org.jboss.remoting.ServerInvocationHandler implementation. For example:

```
   public Object invoke(InvocationRequest invocation) throws Throwable
   {
      Map headers = invocation.getRequestPayload();
      String methodType = (String) headers.get(HTTPMetadataConstants.METHODTYPE);
      if(methodType != null)
      {
         if(methodType.equals("GET"))
         ...
```

PATH (actual value is 'Path') - key for getting the path from the url request from the calling client. This will be populated within the Map returned from call to org.jboss.remoting.InvocationRequest.getRequestPayload() within a org.jboss.remoting.ServerInvocationHandler implementation. For example:

```
   public Object invoke(InvocationRequest invocation) throws Throwable
   {
      Map headers = invocation.getRequestPayload();
      String path = (String) headers.get(HTTPMetadataConstants.PATH);
      ...
```

HTTPVERSION (actual value is 'HttpVersion') - key for getting the HTTP version from the calling client request (e.g. HTTP/1.1).

RESPONSE_CODE (actual value is 'ResponseCode') - key for getting and setting the HTTP response code. Will be used as key to get the response code from metadata Map passed to the Client's invoke() method after the invocation has been made. For example:

```
Map metadata = new HashMap();
Object response = remotingClient.invoke(myPayloadObject, metadata);
Integer responseCode = (Integer) metadata.get(HTTPMetadataConstants.RESPONSE_CODE);
```

Will be used as key to put the response code in the return payload map from invocation handler. For example:

```
   public Object invoke(InvocationRequest invocation) throws Throwable
   {
      Map responseHeaders = invocation.getReturnPayload();
      responseHeaders.put(HTTPMetadataConstants.RESPONSE_CODE,  new Integer(202));
```

RESPONSE_CODE_MESSAGE (actual value is 'ResponseCodeMessage') - key for getting and setting the HTTP response code message. Will be used as the key to get the response code message from metadata Map passed to the Client's invoke() method after the invocation has been made. For example:

```
Map metadata = new HashMap();
Object response = remotingClient.invoke(myPayloadObject, metadata);
String responseCodeMessage = (String) metadata.get(HTTPMetadataConstants.RESPONSE_CODE_MESSAGE);
```

Will be used as key to put the response code message in the return payload map from invocation handler. For example:

```
   public Object invoke(InvocationRequest invocation) throws Throwable
   {
      Map responseHeaders = invocation.getReturnPayload();
      responseHeaders.put(HTTPMetadataConstants.RESPONSE_CODE_MESSAGE, "Custom response code and message
```

RESPONSE_HEADERS (actual value is 'ResponseHeaders') - key for returning the value of `java.net.URLConnection.getHeaderFields()`. In other words, a map containing all of the response headers is stored in the metadata map. For example,

```
Object payload = ... ;
HashMap metadata = new HashMap();
client.invoke(payload, metadata);
Map responseHeaders = (Map) metadata.get(HTTPMetadataConstants.RESPONSE_HEADERS);
```

NO_THROW_ON_ERROR (actual value is 'NoThrowOnError') - key indicating if http client invoker (for transports http, https, servlet, and sslservlet) should throw an exception if the server response code is equal to or greater than 400. Unless set to true, the client invoker will by default throw either the exception that originated on the server (if using remoting server) or throw a org.jboss.remoting.transport.http.WebServerError, whose message will be the error html returned from the web server.

RETURN_EXCEPTION (actual value is 'return-exception') - key indicating if `org.jboss.remoting.transport.servlet.ServletServerInvoker` should throw an exception instead of the orginal error handling behavior of returning an error message.

USE_REMOTING_CONTENT_TYPE (actual value is 'useRemotingContentType') - key indicating if `org.jboss.remoting.marshal.http.HTTPUnMarshaller` should determine that it is reading a String by using the "content-type" return header (if set to "false") or the "remotingContentType" header if set to "true". The default value is "false", to avoid introducing any inconsistent behavior between versions, but, if the `org.jboss.remoting.ServerInvocationHandler` sets the content-type return header, then this parameter should be set to "true".

For every http client request made from remoting client, a remoting version and remoting specific user agent will be set as a request property. The request property key for the remoting version will be 'JBoss-Remoting-Version' and the value will be set based on return from call to Version.getDefaultVersion(). The 'User-Agent' request property value will be set to the evaluation of '"JBossRemoting - " + Version.VERSION'.

org.jboss.remoting.transport.http.HTTPClientInvoker

UNMARSHAL_NULL_STREAM (actual value is 'unmarshalNullStream') - key indicating if

`org.jboss.remoting.transport.http.HTTPClientInvoker` should make the call to `UnMarshaller.read()` when the `InputStream` is null. The default value is "true".

org.jboss.remoting.transport.http.ssl.HTTPSClientInvoker

IGNORE_HTTPS_HOST (actual value is 'org.jboss.security.ignoreHttpsHost') - key indicating if the http client invoker (for transports https and sslservlet) should ignore host name verification (meaning will not check for URL's hostname and server's identification hostname mismatch during handshaking). By default, if this not set to true, standard hostname verification will be performed.

HOSTNAME_VERIFIER (actual value is 'hostnameVerifier') - key indicating the hostname verifier that should be used by the http client invoker. The value should be the fully qualified classname of class that implements javax.net.ssl.HostnameVerifier and has a void constructor.

org.jboss.remoting.transport.rmi.RMIServerInvoker

REGISTRY_PORT_KEY (actual value is 'registryPort') - the port on which to create the RMI registry. The default is 3455. This also needs to have the isParam attribute set to true.

org.jboss.remoting.transport.socket.MicroSocketClientInvoker

TCP_NODELAY_FLAG (actual value is 'enableTcpNoDelay') - can be either true or false and will indicate if client socket should have TCP_NODELAY turned on or off. TCP_NODELAY is for a specific purpose; to disable the Nagle buffering algorithm. It should only be set for applications that send frequent small bursts of information without getting an immediate response; where timely delivery of data is required (the canonical example is mouse movements). The default is false.

MAX_POOL_SIZE_FLAG (actual value is 'clientMaxPoolSize') - the client side maximum number of threads. The default is 50.

CLIENT_SOCKET_CLASS_FLAG (actual value is 'clientSocketClass') - specifies the fully qualified class name for the custom SocketWrapper implementation to use on the client. Note, will need to make sure this is marked as a client parameter (using the 'isParam' attribute). Making this change will not affect the marshaller/unmarshaller that is used, which may also be a requirement.

org.jboss.remoting.transport.socket.ServerThread

CONTINUE_AFTER_TIMEOUT (actual value "continueAfterTimeout") - indicates what a server thread should do after experiencing a `java.net.SocketTimeoutException`. If set to "true", or if JBossSerialization is being used, the server thread will continue to wait for an invocation; otherwise, it will return itself to the thread pool.

org.jboss.remoting.transport.socket.SocketServerInvoker

CHECK_CONNECTION_KEY (actual value is 'socket.check_connection') - key for indicating if socket invoker should continue to keep socket connection between client and server open after invocations by sending a ping on

the connection before being re-used. The default for this is false.

SERVER_SOCKET_CLASS_FLAG (actual value is 'serverSocketClass') - specifies the fully qualified class name for the custom SocketWrapper implementation to use on the server.

org.jboss.remoting.transport.socket.SocketWrapper

WRITE_TIMEOUT (actual value is 'writeTimeout') - key for configuring the "write timeout" facility. If set to a value greater than 0, then a write operation that does not complete within the configured number of milliseconds will be terminated and the socket will be closed. The default value is -1.

6

Sending streams

Remoting supports the sending of InputStreams. It is important to note that this feature DOES NOT copy the stream data directly from the client to the server, but is a true on demand stream. Although this is obviously slower than reading from a stream on the server that has been copied locally, it does allow for true streaming on the server. It also allows for better memory control by the user (versus the framework trying to copy a 3 Gig file into memory and getting out of memory errors).

Use of this new feature is simple. From the client side, there is a method in org.jboss.remoting.Client with the signature:

```
public Object invoke(InputStream inputStream, Object param) throws Throwable
```

So from the client side, would just call invoke as done in the past, and pass the InputStream and the payload as the parameters. An example of the code from the client side would be (this is taken directly from org.jboss.test.remoting.stream.StreamingTestClient):

```
        String param = "foobar";
        File testFile = new File(fileURL.getFile());
        ...
        Object ret = remotingClient.invoke(fileInput, param);
```

From the server side, will need to implement `org.jboss.remoting.stream.StreamInvocationHandler` instead of `org.jboss.remoting.ServerInvocationHandler` . StreamInvocationHandler extends ServerInvocationHandler, with the addition of one new method:

```
public Object handleStream(InputStream stream, Object param)
```

The stream passed to this method can be called on just as any regular local stream. Under the covers, the InputStream passed is really proxy to the real input stream that exists in the client's VM. Subsequent calls to the passed stream will actually be converted to calls on the real stream on the client via this proxy. If the client makes an invocation on the server passing an InputStream as the parameter and the server handler does not implement StreamInvocationhandler, an exception will be thrown to the client caller.

If want to have more control over the stream server being created to send the stream data back to the caller, instead of letting remoting create it internally, can do this by creating a Connector to act as stream server and pass it when making Client invocation.

```
public Object invoke(InputStream inputStream, Object param, Connector streamConnector) throws Throwable
```

Note, the Connector passed must already have been started (else an exception will be thrown). The stream handler

will then be added to the connector with the subystem 'stream'. The Connector passed will NOT be stopped when the stream is closed by the server's stream proxy (which happens automatically when remoting creates the stream server internally).

Can also call `invoke()` method on client and pass the invoker locator would like to use and allow remoting to create the stream server using the specified locator.

```
public Object invoke(InputStream inputStream, Object param, InvokerLocator streamServerLocator) throws Th
```

In this case, the Connector created internally by remoting will be stopped when the stream is closed by the server's stream proxy.

It is VERY IMPORTANT that the StreamInvocationHandler implementation close the InputStream when it finishes reading, as will close the real stream that lives within the client VM.

6.1. Configuration

By default, the stream server which runs within the client JVM uses the following values for its locator uri:

transport - socket

host - tries to first get local host name and if that fails, the local ip (if that fails, localhost).

port - 5405

Currently, the only way to override these settings is to set the following system properties (either via JVM arguments or via `System.setProperty()` method):

remoting.stream.transport - sets the transport type (rmi, http, socket, etc.)

remoting.stream.host - host name or ip address to use

remoting.stream.port - the port to listen on

These properties are important because currently the only way for a target server to get the stream data from the stream server (running within the client JVM) is to have the server invoker make the invocation on a new connection back to the client (see issues below).

6.2. Issues

This is a first pass at the implementation and needs some work in regards to optimizations and configuration. In particular, there is a remoting server that is started to service requests from the stream proxy on the target server for data from the original stream. This raises an issue with the current transports, since the client will have to accept calls for the original stream on a different socket. This may be difficult when control over the client's environment (including firewalls) may not be available. A bi-directional transport, called multiplex, is being introduced as of 1.4.0 release which will allow calls from the server to go over the same socket connection established by the client to the server (JBREM-91). This will make communications back to client much simpler from this standpoint.

7

Remote classloading facility

When a Remoting unmarshaller reads a serialized request or response from the network, it needs to have access to the classes of the objects contained in the serialized object. Frequently, these will be a simple combination of Remoting and Java objects. For example, if an application returns a `java.lang.String`, it will be wrapped in an `org.jboss.remoting.InvocationResponse`, and both classes will be known to the unmarshaller. But what if an application returns an unknown implementation of a known interface? Remoting has a remote classloading facility to support the latter scenario.

7.1. Classloading in client invokers

An instance of `org.jboss.remoting.marshal.serializable.SerializableUnMarshaller`, which is the default unmarshaller, or parent of the default unmarshaller, for the socket, bisocket, and http transports (marshalling and unmarshalling in the rmi transport is handled by the RMI runtime), will look for classes in the following order:

1. try system classloader, which loads from locations specified in the classpath;
2. try the Remoting classloader, that is, the classloader that loaded the Remoting classes (which, depending on the context, may or may not be the system classloader);
3. try to load from Remoting's remote classloading facility;
4. try the current thread's context classloader.

The current thread's context classloader can be moved to the front of the list to enable the alternative behavior:

1. try the current thread's context classloader;
2. try system classloader, which loads from locations specified in the classpath;
3. try the Remoting classloader, that is, the classloader that loaded the Remoting classes (which, depending on the context, may or may not be the system classloader);
4. try to load from Remoting's remote classloading facility.

This alternative behavior may be enabled either by setting the `org.jboss.remoting.Remoting.CLASSLOADING_PARENT_FIRST_DELEGATION` parameter (actual value "classloadingParentFirstDelegation") to "false" in the `InvokerLocator` or in the client invoker's configuration map (see Configuration for more on configuration options) or by setting the system property Remoting.CLASSLOADING_PARENT_FIRST_DELEGATION_PROP (actual value "org.jboss.remoting.classloadingParentFirstDelegation") to "false".

Note that the default behavior, in the absence of any explicit action to the contrary, is for a thread's context classloader to be set to the same classloader that loaded the application. However, somewhere outside of Remoting, the context classloader may be set otherwise. For example, a Remoting client invoker might be running inside an EJB3 container that maintains a classloader associated with the EJB3's EAR file and sets the thread context classloader to the EAR classloader whenever it passes control into code supplied in the EAR. This situation would arise when one EJB3 calls another EJB3, where the invocation would ultimately be made by a Remoting client invoker. **Note**, by the way, the implication that this discussion about classloading in client invokers applies not only to clients in the ordinary client server sense, but also to clients running inside the server.

Note. Prior to Remoting release 2.2.3, a bug in the implementation of the classloading behavior caused the reference to the thread context classloader to be, in fact, to the context classloader of the thread in which the unmarshaller was originally created. A subsequent invocation in a different thread would ignore the current thread context classloader. To get the correct behavior, it is necessary to set the parameter `org.jboss.remoting.Remoting.USE_CURRENT_THREAD_CLASS_LOADER` (actual value "useCurrentThreadClassLoader") to "true".

7.2. Server side support for remote classloading

Remoting implements an optional remote classloading facility that makes it possible for a client invoker unmarshaller to request copies of classes that it encounters during the deserialization process. This facility is provided by a special `org.jboss.remoting.transport.Connector` that runs an `org.jboss.remoting.ServerInvocationHandler` designed to locate and return requested classes. This facility is enabled by configuring a `Connector` with the parameter `org.jboss.remoting.InvokerLocator.LOADER_PORT` (actual value "loaderport") set to an available port number. (See Configuration for more on configuration options.) Using the "loaderport" parameter will result in the creation of a second `Connector` which responds to requests to download classes.

Prior to Remoting release 2.2.2.SP8, the classloading search implemented by the classloading `SerrverInvocationHandler` was the following:

1. try system classloader, which loads from locations specified in the classpath;
2. try the Remoting classloader, that is, the classloader that loaded the Remoting classes (which, depending on the context, may or may not be the system classloader).

This original behavior is too restrictive in the context of the rich classloading options of the JBoss Application Server. As of release 2.2.2.SP8 (with corrections in release 2.2.2.SP11), it is possible to configure the classloading facility with a list of `org.jboss.mx.loading.HeirarchicalLoaderRepository3` class repositories used in the JBoss Application Server. (For more information about class repositories and scoped classloading, see Section 3.2.2.4. "Inside the JBoss Class Loading Architecture" of the Application Server Guide at http://www.jboss.org/file-access/default/members/jbossas/freezone/docs/Server_Configuration_Guide/4/html/index.html [http://www.jboss.org/file-access/default/members/jbossas/freezone/docs/Server_Configuration_Guide/4/html/index.html].)

For example, to allow an EJB3 client to get scoped classes in an EAR where the jboss-app.xml file is

```
<!DOCTYPE jboss-app PUBLIC "-//JBoss//DTD J2EE Application 1.4//EN" "http://www.jboss.org/j2ee/dtd/jbo
```

```
<jboss-app>
    <loader-repository>jboss.remoting:loader=titan.ear</loader-repository>
</jboss-app>
```

the EJB3 Connector should look something like the following, where the new "repositories" attribute is used:

```
<mbean code="org.jboss.remoting.transport.Connector"
       name="jboss.remoting:type=Connector,name=DefaultEjb3Connector,handler=ejb3">
   <depends>jboss.aop:service=AspectDeployer</depends>
   <attribute name="Configuration">
      <config>
         <invoker transport="socket">
            <attribute name="serverBindAddress">${jboss.bind.address}</attribute>
            <attribute name="serverBindPort">3873</attribute>
            <attribute name="loaderport" isParam="true">3883</attribute>
         </invoker>

         <repositories>
            <repository>jboss.remoting:loader=titan.ear</repository>
         </repositories>

         <handlers>
            <handler subsystem="AOP">org.jboss.aspects.remoting.AOPRemotingInvocationHandler</handler>
         </handlers>
      </config>
   </attribute>
</mbean>
```

That is, the classloader Connector is informed about the HeirarchicalLoaderRepository3 with JMX ObjectName "jboss.remoting:loader=titan.ear".

8

Serialization

Serialization - how it works within remoting: In general, remoting will rely on a factory to provide the serialization implementation, or `org.jboss.remoting.serialization.SerializationManager`, to be used when doing object serialization. This factory is `org.jboss.remoting.serialization.SerializationStreamFactory` and is a (as defined by its javadoc):

> factory is for defining the Object stream implemenations to be used along with creating those implemenations for use. The main function will be to return instance of ObjectOutput and ObjectInput. By default, the implementations will be java.io.ObjectOutputStream and java.io.ObjectInputStream.

Currently there are only two different types of serialization implementations; 'java' and 'jboss'. The 'java' type uses `org.jboss.remoting.serialization.impl.java.JavaSerializationManager` as the SerializationManager implementation and is backed by standard Java serialization provide by the JVM, which is the default. The 'jboss' type uses `org.jboss.remoting.serialization.impl.jboss.JBossSerializationManager` as the SerializationManager implementation and is backed by JBoss Serialization.

JBoss Serialization is a new project under development to provide a more performant implementation of object serialization. It complies with java serialization standard with three exceptions:

- SerialUID not needed

- java.io.Serializable is not required

- different protocol

JBoss Serialization requires JDK 1.5

It is possible to override the default SerializationManger implementation to be used by setting the system property 'SERIALIZATION' to the fully qualified name of the class to use (which will need to provide a void constructor).

9

Network Connection Monitoring

Remoting has two mechanisms for monitoring the health of estabilished connections, which inform listeners on the client and server sides when a possible connection failure has been detected.

9.1. Client side monitoring

On the client side, an `org.jboss.remoting.ConnectionValidator` periodically sends a PING message to the server and reports a failure if the response does not arrive within a specified timeout period. The PING is sent on one thread, and another thread determines if the response arrives in time. Separating these two activities allows Remoting to detect a failure regardless of the cause of the failure.

The creation of the `ConnectionValidator` is the responsibility of the `org.jboss.remoting.Client` class. All the application code needs to do is to register an implementation of the `org.jboss.remoting.ConnectionListener` interface, which has only one method:

```
public void handleConnectionException(Throwable throwable, Client client);
```

What actions the `ConnectionListener` chooses to take are up to the application, but disconnecting the `Client` might be a reasonable strategy.

The `Client` class has three methods for registering a `ConnectionListener`:

```
      public void addConnectionListener(ConnectionListener listener);
      public void addConnectionListener(ConnectionListener listener, int pingPeriod);
      public void addConnectionListener(ConnectionListener listener, Map metadata);
```

The second method supports configuring the frequency of PING messages, and the third method supports more general configuration of the `ConnectionValidator`. Note that a given `Client` maintains a single `ConnectionValidator`, so the parameters in the metadata map are applied only on the first call to `Client.addConnectionListener()`. The following parameters are supported by `ConnectionValidator`, which is where the parameter names are defined:

VALIDATOR_PING_PERIOD (actual value "validatorPingPeriod") - specifies the time, in milliseconds, that elapses between the sending of PING messages to the server. The default value is 2000.

VALIDATOR_PING_TIMEOUT (actual value "validatorPingTimeout") - specifies the time, in milliseconds, allowed for arrival of a response to a PING message. The default value is 1000.

FAILURE_DISCONNECT_TIMEOUT (actual value "failureDisconnectTimeout") - if the parameter "stopLeaseOnFailure" (see Interactions between client side and server side connection monitoring) is set to "true",

then "failureDisconnectTimeout" determines the disconnect timeout value to be used by `org.jboss.remoting.LeasePinger` in shutting down. In particular, if "failureDisconnectTimeout" is set to "0", then `LeasePinger` will avoid any network i/o.

Normally, the values for these parameters are obtained either from the `Client`'s configuration map or the metadata map passed to `addConnectionListener()`, with values in the metadata map taking precedence. However, another relevant parameter is defined in `org.jboss.remoting.Client`:

USE_ALL_PARAMS (actual value "useAllParams") - this parameter is searched for in the `InvokerLocator`, in the configuration map passed to the `Client`, and in the metadata map (in that order). If the last occurrence found is set to "true", then parameter values are first obtained from the `InvokerLocator`, followed by the `Client`'s configuration map and the metadata map.

Note that `ConnectionValidator` creates a client invoker to sends the PING messages, and it passes the metadata map to configure the client invoker.

NOTE. The default values of VALIDATOR_PING_PERIOD and VALIDATOR_PING_TIMEOUT have often been found in practice to be too small, increasing the likelihood of spurious connection failures.

NOTE. It is important to set VALIDATOR_PING_PERIOD to a value greater than the value of VALIDATOR_PING_TIMEOUT. Doing so gives the `ConnectionValidator` a chance to notify all `ConnectionListeners`, which might result in shutting down the connection, before the next PING is sent.

For more configuration parameters, see Interactions between client side and server side connection monitoring.

9.2. Server side monitoring

A remoting server also has the capability to detect when a client is no longer available. This is done by estabilishing a lease with the remoting clients that connect to a server. On the client side, an `org.jboss.remoting.LeasePinger` periodically sends PING messages to the server, and on the server side an `org.jboss.remoting.Lease` informs registered listeners if the PING doesn't arrive withing the specified timeout period.

Server side activation. To turn on server side connection failure detection of remoting clients, it is necessary to satisfy two criteria. The first is that the client lease period is set and is a value greater than 0. The value is represented in milliseconds. The client lease period can be set by either the 'clientLeasePeriod' attribute within the Connector configuration or by calling the `Connector` method

```
public void setLeasePeriod(long leasePeriodValue);
```

The second criterion is that an implementation of the `org.jboss.remoting.ConnectionListener` interface is added as a connection listener to the Connector, either via the method

```
public void addConnectionListener(ConnectionListener listener)
```

or through the use of the `ServerInvoker.CONNECTION_LISTENER` parameter (actual value "connectionListener") in the `Connector`'s configuration map or XML configuration file. Once both criteria are met, the remoting server will turn on client leasing.

The ConnectionListener will be notified of both client failures and client disconnects via the handleConnectionEx-

ception() method. If the client failed, meaning its lease was not renewed within configured time period, the first parameter to the handleConnectionException() method will be null. If the client disconnected in a regular manner, the first parameter to the handleConnectionException() method will be of type ClientDisconnectedException (which indicates a normal termination). Note, the client's lease will be renewed on the server with any and every invocation made on the server from the client, whether it be a normal invocation or a ping from the client internally.

The actual lease window established on the server side is dynamic based the rate at which the client updates its lease. In particular, the lease window will always be set to lease period * 2 for any lease that does not have a lease update duration that is longer than 75% of the lease window (meaning if set lease period to 10 seconds and always update that lease in less then 7.5 seconds, the lease period will always remain 10 seconds). If the update duration is greater than 75% of the lease window, the lease window will be reset to the lease duration X 2 (meaning if set lease period to 10 seconds and update that lease in 8 seconds, the new lease window will be set to 16 seconds). Also, the lease will not immediately expire on the first lease timeout (meaning did not get an update within the lease window). It takes two consecutive timeouts before a lease will expire and a notification for client connection failure is fired. This essentially means that the time it will take before a connection listener is notified of a client connection failure will be at least 4 X lease period (no exceptions).

Client side activation. By default, the client is not configured to do client leasing. To allow a client to do leasing, either set the parameter "leasing" to "true" in the `InvokerLocator` or set the parameter `Client.ENABLE_LEASE` (actual value "enableLease") to true in the `InvokerLocator` or in the `Client` configuration map. [The use of `Client.ENABLE_LEASE` is recommended.] This does not mean that client will lease for sure, but will indicate the client should call on the server to see if the server has activated leasing and get the leasing period suggested by the server. It is possible to override the suggested lease period by setting the parameter `org.jboss.remoting.InvokerLocator.CLIENT_LEASE_PERIOD` (actual value "lease_period") to a value greater than 0 and less than the value suggested by the server. **Note.** If the client and server are local, meaning running within the JVM, leasing (and thus connection notification) will not be activated, even if is configured to do so.

If leasing is turned on within the client side, there is no API or configuration changes needed, unless want to override as mentioned previously. When the client initially connects to the server, it will check to see if client leasing is turned on by the server. If it is, it will internally start pinging periodically to the server to maintain the lease. When the client disconnects, it will internally send message to the server to stop monitoring lease for this client. Therefore, it is **IMPORTANT** that disconnect is called on the client when done using it. Otherwise, the client will continue to make its ping call on the server to keep its lease current.

The client can also provide extra metadata that will be communicated to the connection listener in case of failure by supplying a metadata Map to the Client constructor. This map will be included in the Client instance passed to the connection listener (via the handleConnectionException() method) via the Client's getConfiguration() method.

From the server side, there are two ways in which to disable leasing (i.e. turn leasing off). The first is to call:

```
public void removeConnectionListener(ConnectionListener listener)
```

and remove all the registered ConnectionListeners. Once the last one has been removed, leasing will be disabled and all the current leasing sessions will be terminated. The other way is to call:

```
public void setLeasePeriod(long leasePeriodValue)
```

and pass a value less than zero. This will disable leasing, preventing any new leases to be established but will allow current leasing sessions to continue.

The following parameter is relevant to leasing configuration on the server side:

`org.jboss.remoting.ServerInvoker.CLIENT_LEASE_PERIOD` (actual value "clientLeasePeriod") - specifies the timeout period used by the server to determine if a PING is late. The default value is "5000", which indicates that leasing will be activated if an `org.jboss.remoting.ConnectionListener` is registered with the server. This is also the suggested lease period returned by the server when the client inquires if leasing is activated.

The following parameters are relevant to leasing configuration on the client side:

`org.jboss.remoting.Client.ENABLE_LEASE` (actual value "enableLease") - if set to "true", will lead `org.jboss.remoting.Client` to attempt to set up a lease with the server, if leasing is activated on the server.

`org.jboss.remoting.InvokerLocator.CLIENT_LEASE` (actual value "leasing") - if set to "true" in the `InvokerLocator`, will lead `org.jboss.remoting.Client` to attempt to set up a lease with the server, if leasing is activated on the server. It is suggested that this parameter be avoided, in favor of `Client.ENABLE_LEASE`.

`org.jboss.remoting.InvokerLocator.CLIENT_LEASE_PERIOD` (actual value "lease_period") - if set to a value greater than 0 and less than the suggested lease period returned by the server, will be used to determine the time between PING messages sent by `LeasePinger`.

For examples of how to use server side connection listeners, reference org.jboss.test.remoting.lease.LeaseTestServer and org.jboss.test.remoting.lease.LeaseTestClient.

9.3. Interactions between client side and server side connection monitoring

As of Remoting release 2.2.2.SP7, the client side and server side connection monitoring mechanisms can be, and by default are, more closely related, in two ways.

1. If the parameter `org.jboss.remoting.ConnectionValidator.TIE_TO_LEASE` (actual value "tieToLease") is set to true, then, when the server receives a PING message from an `org.jboss.remoting.ConnectionValidator`, it will return a boolean value that indicates whether a lease currently exists for the connection being monitored. If leasing is activated on the client and server side, then a value of "false" indicates that the lease has failed, and the `ConnectionValidator` will treat a returned value of "false" the same as a timeout; that is, it will notifiy listeners of a connection failure. The default value of this parameter is "true". **Note.** If leasing is not activated on the client side, then this parameter has no effect.

2. If the parameter `org.jboss.remoting.ConnectionValidator.STOP_LEASE_ON_FAILURE` (actual value "stopLeaseOnFailure") is set to true, then, upon detecting a connection failure, `ConnectionValidator` will stop the `LeasePinger`, if any, pinging a lease on the same connection. The default value is "true".

Note. As of release 2.2.3, an important concept related to connection monitoring, *connection identity*, is available. Suppose that leasing is enabled and that a client invoker stops and is replaced by a new client invoker. If the replacement occurs quickly, the server side `Lease` may never miss a ping, in which there is no evidence that anything changed on the client side. That is, the connection is still alive, as far as the server is concerned. That semantics might be perfectly acceptable for some applications, but other applications might interpret the same events as a connection failure followed by a new connection. Remoting can be configured to treat a connection as being defined by a client/server pair, which supports the second category of applications.

More specifically, when configured to do so by setting the parameter `org.jboss.remoting.Remoting.USE_CLIENT_CONNECTION_IDENTITY` (actual value "useClientConnectionIdentity") to "true", Remoting identifies a connection with a `LeasePinger`/`Lease` pair. A `Client` participates in a connection when it is connected by way of the new method

```
public void connect(ConnectionListener listener, Map metadata) throws Exception;
```

This method serves to connect the `Client` to the server by way of a new or existing client invoker, and it also (1) registers the `ConnectionListener` with the `Client`'s new or exiting `ConnectionValidator` and (2) registers the `ConnectionValidator` with the client invoker's `LeasePinger`. Subsequently, if any `ConnectionValidator` registered with that `LeasePinger` detects a connection failure, it will (if "stopLeaseOnFailure" is "true") stop the `LeasePinger`, and the `LeasePinger` will cause each registered `ConnectionValidators` to notify each of its registered `ConnectionListeners` of the connection failure. Once the `LeasePinger` has been shut down and all of the notifications have been made, the connection anchored by the `LeasePinger` is defunct, and the associated `Client`'s should be disconnected by a call to `Client.disconnect()`. If such a `Client` is reconnected by a call to `Client.connect()`, it will be associated with a new `LeasePinger` and, therefore, a new connection.

Relevant parameters:

TIE_TO_LEASE (actual value "tieToLease") - specifies whether `ConnectionValidator` should treat the failure of a related lease on the server side as a connection failure. The default value is "true".

STOP_LEASE_ON_FAILURE (actual value "stopLeaseOnFailure") - specifies whether, when a `ConnectionValidator` detects a connection failure, it should stop the associated `org.jboss.remoting.LeasePinger`, if any. The default value is "true".

org.jboss.remoting.Remoting.USE_CLIENT_CONNECTION_IDENTITY (actual value "useClientConnectionIdentity") - tells Remoting to adhere to the new "connection identity" semantics.

10

Transporters - beaming POJOs

There are many ways in which to expose a remote interface to a java object. Some require a complex framework API based on a standard specification and some require new technologies like annotations and AOP. Each of these have their own benefits. JBoss Remoting transporters provide the same behavior via a simple API without the need for any of the newer technologies.

When boiled down, transporters take a plain old java object (POJO) and expose a remote proxy to it via JBoss Remoting. Dynamic proxies and reflection are used to make the typed method calls on that target POJO. Since JBoss Remoting is used, can select from a number of different network transports (i.e. rmi, http, socket, multiplex, etc.), including support for SSL. Even clustering features can be included. See the transporter samples in the next chapter for detailed examples of how to set up use of a transporter.

11

How to use it - sample code

Sample code demonstrating different remoting features can be found in the examples directory. They can be compiled and run manually via your IDE or via an ant build file found in the examples directory. There are many sets of sample code, each with their own package. Within most of these packages, there will be a server and a client class that will need to be executed

11.1. Simple invocation

The simple invocation sample (found in the org.jboss.remoting.samples.simple package), has two classes; SimpleClient and SimpleServer. It demonstrates making a simple invocation from a remoting client to a remoting server. The SimpleClient class will create an InvokerLocator object from a simple url-like string that identifies the remoting server to call upon (which will be socket://localhost:5400 by default). Then the SimpleClient will create a remoting Client class, passing the newly created InvokerLocator. Next the Client will be called to make an invocation on the remoting server, passing the request payload object (which is a String with the value of "Do something"). The server will return a response from this call which is printed to standard output.

Within the SimpleServer, a remoting server is created and started. This is done by first creating an InvokerLocator, just like was done in the SimpleClient. Then constructing a Connector, passing the InvokerLocator. Next, need to call create() on the Connector to initialize all the resources, such as the remoting server invoker. Once created, need to create the invocation handler. The invocation handler is the class that the remoting server will pass client requests on to. The invocation handler in this sample simply returns the simple String "This is the return to SampleInvocationHandler invocation". Once created, the handler is added to the Connector. Finally, the Connector is started and will start listening for incoming client requests.

To run this example, can compile both the SimpleClient and SimpleServer class, then first run the SimpleServer and then the SimpleClient. Or can go to the examples directory and run the ant target 'run-simple-server' and then in another console window run the ant target 'run-simple-client'. For example:

```
ant run-simple-server
```

ant then:

```
ant run-simple-client
```

The output when running the SimpleClient should look like:

```
Calling remoting server with locator uri of: socket://localhost:5400
Invoking server with request of 'Do something'
Invocation response: This is the return to SampleInvocationHandler invocation
```

The output when running the SimpleServer should look like:

```
Starting remoting server with locator uri of: socket://localhost:5400
Invocation request is: Do something
Returning response of: This is the return to SampleInvocationHandler invocation
```

Note: will have to manually shut down the SimpleServer once started.

11.2. HTTP invocation

This http invocation sample (found in the org.jboss.remoting.samples.http package), demonstrates how the http invoker can be used for a variety of http based invocations. This time, will start with the server side. The SimpleServer class is much like the one from the previous simple invocation example, except that instead of using the 'socket' transport, will be using the 'http' transport. Also, instead of using the SampleInvocationHandler class as the handler, will be using the WebInvocationHandler (code shown below).

```
public class WebInvocationHandler implements ServerInvocationHandler
{
   // Pre-defined returns to be sent back to client based on type of request.
   public static final String RESPONSE_VALUE = "This is the return to simple text based http invocation.'
   public static final ComplexObject OBJECT_RESPONSE_VALUE = new ComplexObject(5, "dub", false);
   public static final String HTML_PAGE_RESPONSE = "<html><head><title>Test HTML page</title></head><body
                                                   "<h1>HTTP/Servlet Test HTML page</h1><p>This is a simp
                                                   "<p>Should show up in browser or via invoker client</b

   // Different request types that client may make
   public static final String NULL_RETURN_PARAM = "return_null";
   public static final String OBJECT_RETURN_PARAM = "return_object";
   public static final String STRING_RETURN_PARAM = "return_string";


   /**
    * called to handle a specific invocation
    *
    * @param invocation
    * @return
    * @throws Throwable
    */
   public Object invoke(InvocationRequest invocation) throws Throwable
   {
      // Print out the invocation request
      System.out.println("Invocation request from client is: " + invocation.getParameter());
      if(NULL_RETURN_PARAM.equals(invocation.getParameter()))
      {
         return null;
      }
      else if(invocation.getParameter() instanceof ComplexObject)
      {
            return OBJECT_RESPONSE_VALUE;
      }
      else if(STRING_RETURN_PARAM.equals(invocation.getParameter()))
      {
         Map responseMetadata = invocation.getReturnPayload();
         responseMetadata.put(HTTPMetadataConstants.RESPONSE_CODE,  new Integer(207));
         responseMetadata.put(HTTPMetadataConstants.RESPONSE_CODE_MESSAGE, "Custom response code and mess
         // Just going to return static string as this is just simple example code.
         return RESPONSE_VALUE;
      }
      else
      {
         return HTML_PAGE_RESPONSE;
      }
   }
```

The most interesting part of the WebInvocationHandler is its invoke() method implementation. First it will check to see what the request parameter was from the InvocationRequest and based on what the value is, will return different responses. The first check is to see if the client passed a request to return a null value. The second will check to see if the request parameter from the client was of type ComplexObject. If so, return the pre-built ComplexObject that was created as a static variable.

After that, will check to see if the request parameter was for returning a simple String. Notice in this block, will set the desired response code and message to be returned to the client. In this case, are setting the response code to be returned to 207 and the response message to "Custom response code and message from remoting server". These are non-standard code and message, but can be anything desired.

Last, if have not found a matching invocation request parameter, will just return some simple html.

Now onto the client side for making the calls to this handler, which can be found in SimpleClient (code shown below).

```
public class SimpleClient
{
   // Default locator values
   private static String transport = "http";
   private static String host = "localhost";
   private static int port = 5400;

   public void makeInvocation(String locatorURI) throws Throwable
   {
      // create InvokerLocator with the url type string
      // indicating the target remoting server to call upon.
      InvokerLocator locator = new InvokerLocator(locatorURI);
      System.out.println("Calling remoting server with locator uri of: " + locatorURI);

      Client remotingClient = new Client(locator);

      // make invocation on remoting server and send complex data object
      // by default, the remoting http client invoker will use method type of POST,
      // which is needed when ever sending objects to the server.  So no metadata map needs
      // to be passed to the invoke() method.
      Object response = remotingClient.invoke(new ComplexObject(2, "foo", true), null);

      System.out.println("\nResponse from remoting http server when making http POST request and sending


      Map metadata = new HashMap();
      // set the metadata so remoting client knows to use http GET method type
      metadata.put("TYPE", "GET");
      // not actually sending any data to the remoting server, just want to get its response
      response = remotingClient.invoke((Object) null, metadata);

      System.out.println("\nResponse from remoting http server when making GET request:\n" + response);

      // now set type back to POST and send a plain text based request
      metadata.put("TYPE", "POST");
      response = remotingClient.invoke(WebInvocationHandler.STRING_RETURN_PARAM, metadata);

      System.out.println("\nResponse from remoting http server when making http POST request and sending

      // notice are getting custom response code and message set by web invocation handler
      Integer responseCode = (Integer) metadata.get(HTTPMetadataConstants.RESPONSE_CODE);
      String responseMessage = (String) metadata.get(HTTPMetadataConstants.RESPONSE_CODE_MESSAGE);
      System.out.println("Response code from server: " + responseCode);
      System.out.println("Response message from server: " + responseMessage);
```

```
   }
```

This SimpleClient, like the one before in the simple invocation example, starts off by creating an InvokerLocator and remoting Client instance, except is using http transport instead of socket. The first invocation made is to send a newly constructed ComplexObject. If remember from the WebInvocationHandler above, will expect this invocation to return a different ComplexObject, which can be seen in the following system output line.

The next invocation to be made is a simple http GET request. To do this, must first let the remoting client know that the method type needs to be changed from the default, which is POST, to be GET. Then make the invocation with a null payload (since not wanting to send any data, just get data in response) and the metadata map just populated with the GET type. This invocation request will return a response of html.

Then, will change back to being a POST type request and will pass a simple String as the payload to the invocation request. This will return a simple String as the response from the WebInvocationHandler. Afterward, will see the specific response code and message printed to standard output, as well as the exception itself.

To run this example, can compile all the classes in the package, then first run the SimpleServer and then the SimpleClient. Or can go to the examples directory and run the ant target 'run-http-server' and then in another console window run the ant target 'run-http-client'. For example:

```
ant run-http-server
```

and then:

```
ant run-http-client
```

The output when running the SimpleClient should look like:

```
Response from remoting http server when making http POST request and sending a complex data object:
ComplexObject (i = 5, s = dub, b = false, bytes.length = 0)

Response from remoting http server when making GET request:
<html><head><title>Test HTML page</title></head><body><h1>HTTP/Servlet Test HTML page</h1><p>This is a si

Response from remoting http server when making http POST request and sending a text based request:
This is the return to simple text based http invocation.
Response code from server: 207
Response message from server: Custom response code and message from remoting server
```

Notice that the first response is the ComplexObject from the static variable returned within WebInvocationHandler. The next response is html and then simple text from the WebInvocationHandler. Can see the specific response code and message set in the WebInvocationHandler.

The output from the SimpleServer should look like:

```
Starting remoting server with locator uri of: http://localhost:5400
Jan 26, 2006 11:39:53 PM org.apache.coyote.http11.Http11BaseProtocol init
INFO: Initializing Coyote HTTP/1.1 on http-127.0.0.1-5400
Jan 26, 2006 11:39:53 PM org.apache.coyote.http11.Http11BaseProtocol start
INFO: Starting Coyote HTTP/1.1 on http-127.0.0.1-5400
Invocation request from client is: ComplexObject (i = 2, s = foo, b = true, bytes.length = 0)
Invocation request from client is: null
Invocation request from client is: return_string
```

First the information for the http server invoker is written, which includes the locator uri used to start the server and

the output from starting the Tomcat connector. Then will see the invocation parameter passed for each client request.

Since the SimpleServer should still be running, can open a web browser and enter the locator uri, http://localhost:5400. This should cause the browser to render the html returned from the WebInvocationHandler.

11.3. Oneway invocation

The oneway invocation sample (found in the org.jboss.remoting.samples.oneway package) is very similar to the simple invocation example, except in this sample, the client will make asynchronous invocations on the server.

The OnewayClient class sets up the remoting client as in the simple invocation sample, but instead of using the invoke() method, it uses the invokeOneway() method on the Client class. There are two basic modes when making a oneway invocation in remoting. The first is to have the calling thread to be the one that makes the actual call to the server. This allows the caller to ensure that the invocation request at least made it to the server. Once the server receives the invocation request, the call will return (and the request will be processed by a separate worker thread on the server). The other mode, which is demonstrated in the second call to invokeOneway, allows for the calling thread to return immediately and a worker thread on the client side will make the actual invocation on the server. This is faster of the two modes, but if there is a problem making the request on the server, the original caller will be unaware.

NOTE. In the interest of performance, the behavior of the various transports is not required to conform to the preceding description of the first, "server side", mode, in which the invocation is made on the calling thread. In particular, the socket and bisocket transports return immediately after writing the invocation, without waiting for a response from the server.

The OnewayServer is exactly the same as the SimpleServer from the previous example, with the exception that invocation handler returns null (since even if did return a response, would not be delivered to the original caller).

To run this example, can compile both the OnewayClient and OnewayServer class, then run the OnewayServer and then the OnewayClient. Or can go to the examples directory and run the ant target 'run-oneway-server' and then in another console window run the ant target 'run-oneway-client'. For example:

```
ant run-oneway-server
```

and then:

```
ant run-oneway-client
```

The output when running the OnewayClient should look like:

```
Calling remoting server with locator uri of: socket://localhost:5400
Making oneway invocation with payload of 'Oneway call 1.'
Making oneway invocation with payload of 'Oneway call 2.'
```

The output when running the OnewayServer should look like:

```
Starting remoting server with locator uri of: socket://localhost:5400
Invocation request is: Oneway call 1.
Invocation request is: Oneway call 2.
```

Note: will have to manually shut down the OnewayServer once started.

Although this example only demonstrates making one way invocations, could include this with callbacks (see further down) to have asynchronous invocations with callbacks to verify was processed.

11.4. Discovery and invocation

The discovery sample (found in the org.jboss.remoting.samples.detection package) is similar to the simple invocation example in that it makes a simple invocation from the client to the server. However, in this example, instead of explicitly specifying the invoker locator to use for the target remoting server, it is discovered dynamically during runtime. This example is composed of two classes; SimpleDetectorClient and SimpleDetectorServer.

The SimpleDetectorClient starts off by setting up the remoting detector. Detection on the client side requires a few components; a JMX MBeanServer, one or more Detectors, and a NetworkRegistry. The Detectors will listen for detection messages from remoting servers and then add the information for the detected servers to the NetworkRegistry. They use JMX to lookup and call on the NetworkRegistry. The NetworkRegistry uses JMX Notifications to emit changes in network topology (remoting servers being added or removed).

In this particular example, the SimpleDetectorClient is registered with the NetworkRegistry as a notification listener. When it receives notifications from the NetworkRegistry (via the handleNotification() method), it will check to see if the notification is for adding or removing a remoting server. If it is for adding a remoting server, the SimpleDetectorClient will get the array of InvokerLocators from the NetworkNotification and make a remote call for each. If the notification is for removing a remoting server, the SimpleDetectorClient will simply print out a message saying which server has been removed.

The biggest change between the SimpleDetectorServer and the SimpleServer from the first sample is that have added a method, setupDetector(), to create and start a remoting Detector. On the server side, only two components are needed for detection; the Detector and a JMX MBeanServer. As for the setup of the Connector, it is exactly the same as before. Notice that even though we have added a Detector on the server side, the Connector is not directly aware of either Detector or the MBeanServer, so no code changes for the Connector setup is required.

To run this example, can compile both the SimpleDetectorClient and SimpleDetectorServer class, then run the SimpleDetectorServer and then the SimpleDetectorClient. Or can go to the examples directory and run the ant target 'run-detector-server' and then in another window run the ant target 'run-detector-client'. For example:

```
ant run-detector-server
```

and then:

```
ant run-detector-client
```

The initial output when running the SimpleDetectorClient should look like:

```
ri Jan 13 09:36:50 EST 2006: [CLIENT]: Starting JBoss/Remoting client... to stop this client, kill it mar
Fri Jan 13 09:36:50 EST 2006: [CLIENT]: NetworkRegistry has been created
Fri Jan 13 09:36:50 EST 2006: [CLIENT]: NetworkRegistry has added the client as a listener
Fri Jan 13 09:36:50 EST 2006: [CLIENT]: MulticastDetector has been created and is listening for new Netwo
Fri Jan 13 09:36:50 EST 2006: [CLIENT]: GOT A NETWORK-REGISTRY NOTIFICATION: jboss.network.server.added
Fri Jan 13 09:36:50 EST 2006: [CLIENT]: New server(s) have been detected - getting locators and sending w
Fri Jan 13 09:36:50 EST 2006: [CLIENT]: Sending welcome message to remoting server with locator uri of: s
Fri Jan 13 09:36:50 EST 2006: [CLIENT]: The newly discovered server sent this response to our welcome mes
```

The output when running the SimpleDetectorServer should look like:

```
Fri Jan 13 09:36:46 EST 2006: [SERVER]: Starting JBoss/Remoting server... to stop this server, kill it ma
Fri Jan 13 09:36:46 EST 2006: [SERVER]: This server's endpoint will be: socket://localhost:5400
Fri Jan 13 09:36:46 EST 2006: [SERVER]: MulticastDetector has been created and is listening for new Netwo
Fri Jan 13 09:36:46 EST 2006: [SERVER]: Starting remoting server with locator uri of: socket://localhost:
Fri Jan 13 09:36:46 EST 2006: [SERVER]: Added our invocation handler; we are now ready to begin accepting
Fri Jan 13 09:36:50 EST 2006: [SERVER]: RECEIVED A CLIENT MESSAGE: Welcome Aboard!
Fri Jan 13 09:36:50 EST 2006: [SERVER]: Returning the following message back to the client: Received your
```

At this point, try stopping the SimpleDetectorServer (notice that the SimpleDetectorClient should still be running). After a few seconds, the client detector should detect that the server is no longer available and will see something like the following appended in the SimpleDetectorClient console window:

```
Fri Jan 13 09:37:04 EST 2006: [CLIENT]: GOT A NETWORK-REGISTRY NOTIFICATION: jboss.network.server.removed
Fri Jan 13 09:37:04 EST 2006: [CLIENT]: It has been detected that a server has gone down with a locator o
```

11.5. Callbacks

The callback sample (found in the org.jboss.remoting.samples.callback package) illustrates how to perform callbacks from a remoting server to a remoting client. This example is composed of two classes; CallbackClient and CallbackServer.

Within remoting, there are two approaches in which a callback can be received. The first is to actively ask for callback messages from the remoting server, which is called a pull callback (since are pulling the callbacks from the server). The second is to have the server send the callbacks to the client as they are generated, which is called a push callback. This sample demonstrates how to do both pull and push callbacks.

Looking at the CallbackClient class, will see that the first thing done is to create a remoting Client, which is done in the same manner as previous examples. Next, we'll perform a pull callback, which requires the creation of a CallbackHandler. The CallbackHandler, which implements the InvokerCallbackHandler interface, is what is called upon with a Callback object when a callback is received. The Callback object contains information such as the callback message (in Object form), the server locator from where the callback originally came from, and a handle object which can help to identify callback context (similar to the handle object within a JMX Notification). Once created, the CallbackHandler is then registered as a listener within the Client. This will cause the client to make a call to the server to notify the server it has a callback listener (more on this below in the server section). Although the CallbackHandler is not called upon directly when doing pull callbacks, it is needed as an identifier for the callbacks.

Then the client will wait a few seconds, make a simple invocation on the server, and then call on the remoting Client instance to get any callbacks that may be available for our CallbackHandler. This will return a list of callbacks, if any exist. The list will be iterated and each callback will be printed to standard output. Finally, the callback handler will be removed as a listener from the remoting Client (which in turns removes it from the remoting server).

After performing a pull callback, will perform a push callback. This is a little more involved as requires creating a callback server to which the remoting target server can callback on when it generates a callback message. To do this, will need to create a remoting Connector, just as have seen in previous examples. For this particular example, we use the same locator url as our target remoting server, but increment the port to listen on by one. Will also notice that use the SampleInvocationHandler hander from the CallbackServer (more in this in a minute). After creating our callback server, a CallbackHandler and callback handle object is created. Next, remoting Client is called to

add our callback listener. Here we pass not only the CallbackHandler, but the InvokerLocator for the callback server (so the target server will know where to deliver callback messages to), and the callback handle object (which will be included in all the callback messages delivered for this particular callback listener).

Then the client will wait a few seconds, to allow the target server time to generate and deliver callback messages. After that, we remove the callback listener and clean up our callback server.

The CallbackServer is pretty much the same as the previous samples in setting up the remoting server, via the Connector. The biggest change resides in the ServerInvocationHandler implementation, SampleInvocationHandler (which is an inner class to CallbackServer). The first thing to notice is now have a variable called listeners, which is a List to hold any callback listeners that get registered. Also, in the constructor of the SampleInvocationHandler, we set up a new thread to run in the background. This thread, executing the run() method in SampleInvocationHandler, will continually loop looking to see if the shouldGenerateCallbacks has been set. If it has been, will create a Callback object and loop through its list of listeners and tell each listener to handle the newly created callback. Have also added implementation to the addListener() and removeListener() methods where will either add or remove specified callback listener from the internal callback listener list and set the shouldGenerateCallbacks flag accordingly. The invoke() method remains the same as in previous samples.

To run this example, can compile both the CallbackClient and CallbackServer class, then run the CallbackServer and then the CallbackClient. Or can go to the examples directory and run the ant target 'run-callback-server' and then in another window run the ant target 'run-callback-client. For example:

```
ant run-callback-server
```

and then:

```
ant run-callback-client
```

The output in the CallbackClient console window should look like:

```
Calling remoting server with locator uri of: socket://localhost:5400
Invocation response: This is the return to SampleInvocationHandler invocation
Pull Callback value = Callback 1: This is the payload of callback invocation.
Pull Callback value = Callback 2: This is the payload of callback invocation.
Starting remoting server with locator uri of: InvokerLocator [socket://127.0.0.1:5401/]
Received push callback.
Received callback value of: Callback 3: This is the payload of callback invocation.
Received callback handle object of: myCallbackHandleObject
Received callback server invoker of: InvokerLocator [socket://127.0.0.1:5400/]
Received push callback.
Received callback value of: Callback 4: This is the payload of callback invocation.
Received callback handle object of: myCallbackHandleObject
Received callback server invoker of: InvokerLocator [socket://127.0.0.1:5400/]
```

This output shows that client first pulled two callbacks generated from the server. Then, after creating and registering our second callback handler and a callback server, two callbacks were received from the target server.

The output in the CallbackServer console window should look like:

```
Starting remoting server with locator uri of: socket://localhost:5400
Adding callback listener.
Invocation request is: Do something
Removing callback listener.
Adding callback listener.
Removing callback listener.
```

This output shows two distinct callback handlers being added and removed (with an invocation request being received after the first was added).

There are a few important points to mention about this example. First, notice that in the client, the same callback handle object in the push callbacks was received as was registered with the callback listener. However, there was no special code required to facilitate this within the SampleInvocationHandler. This is handled within remoting automatically. Also notice when the callback server was created within the client, no special coding was required to register the callback handler with it, both were simply passed to the remoting Client instance when registering the callback listener and was handled internally.

11.6. Streaming

The streaning sample (found in the org.jboss.remoting.samples.stream package) illustrates how a java.io.InputStream can be sent from a client and read on demand from a server. This example is composed of two classes: StreamingClient and StreamingServer.

Unlike the previous examples that sent plain old java objects as the payload, this example will be sending a java.io.FileInputStream as the payload to the server. This is a special case because streams can not be serialized. One approach to this might be to write out the contents of a stream to a byte buffer and send the whole data content to the server. However, this approach can be dangerous because if the data content of the stream is large, such as an 800MB file, would run the risk of causing an out of memory error (since are loading all 800MB into memory). Another approach, which is used by JBossRemoting, is to create a proxy to the original stream. This proxy can then be called upon for reading, same as the original stream. When this happens, the proxy will call back the original stream for the requested data.

Looking at the StreamingClient, the remoting Client is created as in previous samples. Next, will create a java.io.FileInputStream to the sample.txt file on disk (which is in the same directory as the test classes). Finally, will call the remoting Client to do its invocation, passing the new FileInputStream and the name of the file. The second parameter could be of any Object type and is meant to supply some meaningful context to the server in regards to the stream being passed, such as the file name to use when writing to disk on the server side. The response from the server, in this example, is the size of the file it wrote to disk.

The StreamingServer sets up the remoting server as was done in previous examples. However, instead of using an implementation of the ServerInvocationHandler class as the server handler, an implementation of the StreamInvocationHandler (which extends the ServerInvocationHandler) is used. The StreamInvocationHandler includes an extra method called handleStream() especially for processing requests with a stream as the payload. In this example, the class implementing the StreamInvocationHandler is the TestStreamInvocationHandler class, which is an inner class to the StreamingServer. The handleStream() method within the TestStreamInvocationHandler will use the stream passed to it to write out its contents to a file on disk, as specified by the second parameter passed to the handleStream() method. Upon writing out the file to disk, the handleStream() method will return to the client caller the size of the file.

To run this example, can compile both the StreamingClient and StreamingServer class, then run the StreamingServer and then the StreamingClient. Or can go to the examples directory and run the ant target 'run-stream-server' and then in another window run the ant target 'run-stream-client'. For example:

```
ant run-stream-server
```

and then:

```
ant run-stream-client
```

The output in the StreamingClient console window should look like:

```
Calling on remoting server with locator uri of: socket://localhost:5400
Sending input stream for file sample.txt to server.
Size of file sample.txt is 987
Server returned 987 as the size of the file read.
```

The output in the StreamingServer console window should look like:

```
Starting remoting server with locator uri of: socket://localhost:5400
Received input stream from client to write out to file server_sample.txt
Read stream of size 987.  Now writing to server_sample.txt
New file server_sample.txt has been written out to C:\tmp\JBossRemoting_1_4_0_final\examples\server_sampl
```

After running this example, there should be a newly created server_sample.txt file in the root examples directory. The contents of the file should look exactly like the contents of the sample.txt file located in the examples\org\jboss\remoting\samples\stream directory.

11.7. JBoss Serialization

The serialization sample (found in the org.jboss.remoting.samples.serialization package) illustrates how JBoss Serialization can be used in place of the standard java serialization to allow for sending of invocation payload objects that do not implement the java.io.Serializable interface. This example is composed of three classes: SerializationClient, SerializationServer, and NonSerializablePayload.

This example is exactly like the one from the simple example with two differences. The first difference is the use of JBoss Serialization to convert object instances to binary data format for wire transfer. This is accomplished by adding an extra parameter (serializationtype) to the locator url with a value of 'jboss'. Is important to note that use of JBoss Serialization requires JDK 1.5, so this example will need to be run using JDK 1.5. The second difference is instead of sending and receiving a simple String type for the remote invocation payload, will be sending and receiving an instance of the NonSerializablePayload class.

There are a few important points to notice with the NonSerializablePayload class. The first is that it does NOT implement the java.io.Serializable interface. The second is that it has a void parameter constructor. This is a requirement of JBoss Serialization for object instances that do not implement the Serializable interface. However, this void parameter constructor can be private, as in the case of NonSerializablePayload, as to not change the external API of the class.

To run this example, can compile both the SerializationClient and SerializationServer class, then run the SerializationServer and then the SerializationClient. Or can go to the examples directory and run the ant target 'run-serialization-server' and then in another window run the ant target 'run-serialization-client'. For example:

```
ant run-serialization-server
```

and then:

```
ant run-serialization-client
```

The output in the SerializationClient console window should look like:

```
Calling remoting server with locator uri of: socket://localhost:5400/?serializationtype=jboss
Invoking server with request of 'NonSerializablePayload - name: foo, id: 1'
Invocation response: NonSerializablePayload - name: bar, id: 2
```

The output in the SerializationServer console window should look like:

```
Starting remoting server with locator uri of: socket://localhost:5400/?serializationtype=jboss
Invocation request is: NonSerializablePayload - name: foo, id: 1
Returning response of: NonSerializablePayload - name: bar, id: 2
```

Note: will have to manually shut down the SerializationServer once started.

11.8. Transporters

11.8.1. Transporters - beaming POJOs

There are many ways in which to expose a remote interface to a java object. Some require a complex framework API based on a standard specification and some require new technologies like annotations and AOP. Each of these have their own benefits. JBoss Remoting transporters provide the same behavior via a simple API without the need for any of the newer technologies.

When boiled down, transporters take a plain old java object (POJO) and expose a remote proxy to it via JBoss Remoting. Dynamic proxies and reflection are used to make the typed method calls on that target POJO. Since JBoss Remoting is used, can select from a number of different network transports (i.e. rmi, http, socket, multiplex, etc.), including support for SSL. Even clustering features can be included.

How it works

In this section will discuss how remoting transporters can be used, some requirments for usage, and a little detail on the implementation. For greater breath on usage, please review the transporter samples as most use cases are covered there.

To start, will need to have a plain old java object that implements one or more interfaces that want to expose for remote method invocation. Then will need to create a `org.jboss.remoting.transporter.TransporterServer` to wrap around it, so that can be exposed remotely. This can be done in one of two basic ways. The first is to use a static `createTransporterServer()` method of the TransporterServer class. There are many of these create methods, but all basically do that same thing in that they take a remoting locator and target pojo and will return a TransporterServer instance that has been started and ready to receive remote invocations (see javadoc for TransporterServer for all the different static createTransporterServer() methods). The other way to create a TransporterServer for the target pojo is to construct an instance of it. This provides a little more flexibility as are able to control more aspects of the TransporterServer, such as when it will be started.

When a TransporterServer is created, it will create a remoting Connector using the locator provided. It will generate a server invocation handler that wraps the target pojo provided and use reflection to make the calls on it based on the invocations it receives from clients. By default, the subsystem underwhich the server invocation handler is registered is the interface class name for which the target pojo is exposing. If the target implements multiple inter-

faces, and a specific one to use is not specified, all the interfaces will be registered as subsystems for the same server invocation handler. Whenever no long want the target pojo to receive remote method invocations, will need to call the `stop()` method on the TransporterServer for the target pojo (this is very important, as otherwise will never be released from memory and will continue to consume network and memory resources).

On the client side, in order to be able to call on the target pojo remotely, will need to use the `org.jboss.remoting.transporter.TransporterClient`. Unlike the TransporterServer, can only use the static create methods of the TransporterClient (this is because the return to the static create method is a typed dynamic proxy). The static method to call on the TransportClient is `createTransporterClient()`, where will pass the locator to find the target pojo (same as one used when creating the TransporterServer) and the interface for the target pojo that want to make remote method invocations on. The return from this create call will be a dynamic proxy which you can cast to to same interface type supplied. At that point, can make typed method invocations on the returned object, which will then make the remote invocations under the covers. Note that can have multiple transporter clients to the same target pojo, each using different interface types for making calls.

When no longer need to make invocations on the target pojo, the resources associated with the remoting client will need to be cleaned up. This is done by calling the `destroyTransporterClient()` method of the TransporterClient. This is important to remember to do, as will otherwise leave network resources active even though not in use.

One of the features of using remoting transporters is location transparency. By this mean that client proxies returned by the TransporterClient can be passed over the network. For example, can have a target pojo that returns from a method call a client proxy (that it created using the TransporterClient) in which the client can call on directly as well. See the transporter proxy sample code to see how this can be done.

Another nice feature when using transporters is the ability to cluster. To be more specific, can create multiple target pojos using the TransporterServer in clustered mode and then use the TransporterClient in clustered mode to create a client proxy that will discover the location of the target pojos are wanting to call on. Will also provide automatic, seemless failover of remote method invocations in the case that a particular target pojo instance fails. However, note that only provide invocation failover and does not take into account state transfer between target pojos (would need addition of JBoss Cache or some other state synchronization tool).

The transporter sample spans several examples showing different ways to use the transporter. Each specific example is within its own package under the org.jboss.remoting.samples.transporter package. Since each of the transporter examples includes common objects, as well as client and server classes, the common objects will be found under the main transporter sub-package and the client and server classes in their respective sub-packages (named client and server).

11.8.2. Transporters sample - simple

The simple transporter example (found in org.jboss.remoting.samples.transporter.simple package) demonstrates a very simple example of how to use the transporters to expose a plain old java object for remote method invocations.

In this simple transporter example, will be taking a class that formats a java.util.Date into a simple String representation and exposing it so can call on the remotely. The target object in this case, org.jboss.remoting.samples.transporter.simple.DateProcessorImpl, implements the org.jboss.remoting.samples.transporter.simple.DateProcessor interfaces (as shown below):

```
public interface DateProcessor
{
```

```
   public String formatDate(Date dateToConvert);
}


public class DateProcessorImpl implements DateProcessor
{
   public String formatDate(Date dateToConvert)
   {
      DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.MEDIUM);
      return dateFormat.format(dateToConvert);
   }
}
```

This is then exposed using the TransporterServer by the org.jboss.remoting.samples.transporter.simple.Server class.

```
public class Server
{
   public static void main(String[] args) throws Exception
   {
      TransporterServer server = TransporterServer.createTransporterServer("socket://localhost:5400", new
      Thread.sleep(10000);
      server.stop();
   }
}
```

The Server class simply creates a TransporterServer by indicating the locator url would like to use for the remoting server, a newly created instance of DataProcessorImpl, and the interface type would like to expose remotely. The TransporterServer returned from the createTransporterServer call is live and ready to receive incoming method invocation requests. Will then wait 10 seconds for a request, then stop the server.

Next need to have client to make the remote invocation. This can be found within org.jboss.remoting.samples.transporter.simple.Client.

```
public class Client
{
   public static void main(String[] args) throws Exception
   {
      DateProcessor dateProcessor = (DateProcessor) TransporterClient.createTransporterClient("socket://l
      String formattedDate = dateProcessor.formatDate(new Date());
      System.out.println("Current date: " + formattedDate);
   }
}
```

In the Client class, create a TransporterClient which can be cast to the desired type, which is DataProcessor in this case. In calling the createTransporterClient, need to specify the locator ulr (same as was used for the TransporterServer), and the interface type will be calling on for the target pojo. Once have the DateProcessor variable, will make the call to formatDate() and pass a newly created Date object. The return will be a formated String of the date passed.

To run this example, can run the Server and then the Client. Or can go to the examples directory and run the ant target 'run-transporter-simple-server' and then in another window run the ant target 'run-transporter-simple-client'. For example:

```
ant run-transporter-simple-server
```

and then:

```
ant run-transporter-simple-client
```

The output from the client window should look similar to:

```
Current date: Jul 31, 2006
```

11.8.3. Transporter sample - basic

The basic transporter example (found in org.jboss.remoting.samples.transporter.basic package) illustrates how to build a simple transporter for making remote invocations on plain old java objects.

In this basic transporter example, will be using a few domain objects; `Customer` and Address, which are just data objects.

```
public class Customer implements Serializable
{
   private String firstName = null;
   private String lastName = null;
   private Address addr = null;
   private int customerId = -1;

   public String getFirstName()
   {
      return firstName;
   }

   public void setFirstName(String firstName)
   {
      this.firstName = firstName;
   }

   public String getLastName()
   {
      return lastName;
   }

   public void setLastName(String lastName)
   {
      this.lastName = lastName;
   }

   public Address getAddr()
   {
      return addr;
   }

   public void setAddr(Address addr)
   {
      this.addr = addr;
   }

   public int getCustomerId()
   {
      return customerId;
   }

   public void setCustomerId(int customerId)
   {
      this.customerId = customerId;
   }
```

```
   public String toString()
   {
      StringBuffer buffer = new StringBuffer();
      buffer.append("\nCustomer:\n");
      buffer.append("customer id: " + customerId + "\n");
      buffer.append("first name: " + firstName + "\n");
      buffer.append("last name: " + lastName + "\n");
      buffer.append("street: " + addr.getStreet() + "\n");
      buffer.append("city: " + addr.getCity() + "\n");
      buffer.append("state: " + addr.getState() + "\n");
      buffer.append("zip: " + addr.getZip() + "\n");

      return buffer.toString();
   }
}
```

```
public class Address implements Serializable
{
   private String street = null;
   private String city = null;
   private String state = null;
   private int zip = -1;

   public String getStreet()
   {
      return street;
   }

   public void setStreet(String street)
   {
      this.street = street;
   }

   public String getCity()
   {
      return city;
   }

   public void setCity(String city)
   {
      this.city = city;
   }

   public String getState()
   {
      return state;
   }

   public void setState(String state)
   {
      this.state = state;
   }

   public int getZip()
   {
      return zip;
   }

   public void setZip(int zip)
   {
      this.zip = zip;
   }
}
```

Next comes the POJO that we want to expose a remote proxy for, which is `CustomerProcessorImpl` class. This im-

plementation has one method to process a `Customer` object. It also implements the `CustomerProcessor` interface.

```
public class CustomerProcessorImpl implements CustomerProcessor
{
   /**
    * Takes the customer passed, and if not null and customer id
    * is less than 0, will create a new random id and set it.
    * The customer object returned will be the modified customer
    * object passed.
    *
    * @param customer
    * @return
    */
   public Customer processCustomer(Customer customer)
   {
      if(customer != null && customer.getCustomerId() < 0)
      {
         customer.setCustomerId(new Random().nextInt(1000));
      }
      System.out.println("processed customer with new id of " + customer.getCustomerId());
      return customer;
   }
}
```

```
public interface CustomerProcessor
{
   /**
    * Process a customer object.  Implementors
    * should ensure that the customer object
    * passed as parameter should have its internal
    * state changed somehow and returned.
    *
    * @param customer
    * @return
    */
   public Customer processCustomer(Customer customer);
}
```

So far, nothing special, just plain old java objects. Next need to create the server component that will listen for remote request to invoke on the target POJO. This is where the transporter comes in.

```
public class Server
{
   private String locatorURI = "socket://localhost:5400";
   private TransporterServer server = null;

   public void start() throws Exception
   {
      server = TransporterServer.createTransporterServer(locatorURI, new CustomerProcessorImpl());
   }

   public void stop()
   {
      if(server != null)
      {
         server.stop();
      }
   }

   public static void main(String[] args)
   {
      Server server = new Server();
      try
      {
```

```
         server.start();

         Thread.currentThread().sleep(60000);

      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
      finally
      {
         server.stop();
      }
   }
}
```

The `Server` class is a pretty simple one. It calls the `TransporterServer` factory method to create the server component for the `CustomerProcessorImpl` instance using the specified remoting locator information.

The `TransporterServer` returned from the `createTransporterServer()` call will be a running instance of a remoting server using the `socket` transport that is bound to `localhost` and listening for remote requests on port `5400`. The requests that come in will be forwarded to the remoting handler which will convert them into direct method calls on the target POJO, `CustomerProcessorImpl` in this case, using reflection.

The `TransporterServer` has a `start()` and `stop()` method exposed to control when to start and stop the running of the remoting server. The `start()` method is called automatically within the `createTransporterServer()` method, so is ready to receive requests upon the return of this method. The `stop()` method, however, needs to be called explicitly when no longer wish to receive remote calls on the target POJO.

Next up is the client side. This is represented by the `Client` class.

```
public class Client
{
   private String locatorURI = "socket://localhost:5400";

   public void makeClientCall() throws Exception
   {
      Customer customer = createCustomer();

      CustomerProcessor customerProcessor = (CustomerProcessor) TransporterClient.createTransporterClient

      System.out.println("Customer to be processed: " + customer);
      Customer processedCustomer = customerProcessor.processCustomer(customer);
      System.out.println("Customer is now: " + processedCustomer);

      TransporterClient.destroyTransporterClient(customerProcessor);
   }

   private Customer createCustomer()
   {
      Customer cust = new Customer();
      cust.setFirstName("Bob");
      cust.setLastName("Smith");
      Address addr = new Address();
      addr.setStreet("101 Oak Street");
      addr.setCity("Atlanata");
      addr.setState("GA");
      addr.setZip(30249);
      cust.setAddr(addr);

      return cust;
   }
```

```
   public static void main(String[] args)
   {
      Client client = new Client();
      try
      {
         client.makeClientCall();
      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
   }
}
```

The `Client` class is also pretty simple. It creates a new `Customer` object instance, creates the remote proxy to the `CustomerProcessor`, and then calls on the `CustomerProcessor` to process its new `Customer` instance.

To get the remote proxy for the `CustomerProcessor`, all that is required is to call the `TransporterClient`'s method `createTransporterClient()` method and pass the locator uri and the type of the remote proxy (and explicitly cast the return to that type). This will create a dynamic proxy for the specified type, `CustomerProcessor` in this case, which is backed by a remoting client which in turn makes the calls to the remote POJO's remoting server. Once the call to `createTransportClient()` has returned, the remoting client has already made its connection to the remoting server and is ready to make calls (will throw an exception if it could not connect to the specified remoting server).

When finished making calls on the remote POJO proxy, will need to explicitly destroy the client by calling `destroyTransporterClient()` and pass the remote proxy instance. This allows the remoting client to disconnect from the POJO's remoting server and clean up any network resources previously used.

To run this example, can run the Server and then the Client. Or can go to the examples directory and run the ant target 'run-transporter-basic-server' and then in another window run the ant target 'run-transporter-basic-client'. For example:

```
ant run-transporter-basic-server
```

and then:

```
ant run-transporter-basic-client
```

The output from the Client console should be similar to:

```
Customer to be processed:
Customer:
customer id: -1
first name: Bob
last name: Smith
street: 101 Oak Street
city: Atlanata
state: GA
zip: 30249

Customer is now:
Customer:
customer id: 204
first name: Bob
last name: Smith
street: 101 Oak Street
```

```
city: Atlanata
state: GA
zip: 30249
```

and the output from the Server class should be similar to:

```
processed customer with new id of 204
```

The output shows that the `Customer` instance created on the client was sent to the server where it was processed (by setting the customer id to 204) and returned to the client (and printed out showing that the customer id was set to 204).

11.8.4. Transporter sample - JBoss serialization

The transporter serialization example (found in org.jboss.remoting.samples.transporter.serialization package) is very similar to the previous basic example, except in this one, the domain objects being sent over the wire will NOT be Serializable. This is accomplished via the use of JBoss Serialization. This can be useful when don't know which domain objects you may be using in remote calls or if adding ability for remote calls on legacy code.

To start, there are a few more domain objects: `Order`, `OrderProcessor`, and `OrderProcessorImpl`. These will use some of the domain objects from the previous example as well, such as `Customer`.

```
public class Order
{
   private int orderId = -1;
   private boolean isProcessed = false;
   private Customer customer = null;
   private List items = null;


   public int getOrderId()
   {
      return orderId;
   }

   public void setOrderId(int orderId)
   {
      this.orderId = orderId;
   }

   public boolean isProcessed()
   {
      return isProcessed;
   }

   public void setProcessed(boolean processed)
   {
      isProcessed = processed;
   }

   public Customer getCustomer()
   {
      return customer;
   }

   public void setCustomer(Customer customer)
   {
      this.customer = customer;
```

```
   }

   public List getItems()
   {
      return items;
   }

   public void setItems(List items)
   {
      this.items = items;
   }

   public String toString()
   {
      StringBuffer buffer = new StringBuffer();
      buffer.append("\nOrder:\n");
      buffer.append("\nIs processed: " + isProcessed);
      buffer.append("\nOrder id: " + orderId);
      buffer.append(customer.toString());

      buffer.append("\nItems ordered:");
      Iterator itr = items.iterator();
      while(itr.hasNext())
      {
         buffer.append("\n" + itr.next().toString());
      }

      return buffer.toString();
   }
}
```

```
public class OrderProcessorImpl implements OrderProcessor
{
   private CustomerProcessor customerProcessor = null;

   public OrderProcessorImpl()
   {
      customerProcessor = new CustomerProcessorImpl();
   }

   public Order processOrder(Order order)
   {
      System.out.println("Incoming order to process from customer.\n" + order.getCustomer());

      // has this customer been processed?
      if(order.getCustomer().getCustomerId() < 0)
      {
         order.setCustomer(customerProcessor.processCustomer(order.getCustomer()));
      }

      List items = order.getItems();
      System.out.println("Items ordered:");
      Iterator itr = items.iterator();
      while(itr.hasNext())
      {
         System.out.println(itr.next());
      }

      order.setOrderId(new Random().nextInt(1000));
      order.setProcessed(true);

      System.out.println("Order processed.  Order id now: " + order.getOrderId());
      return order;
   }
}
```

```
public interface OrderProcessor
{
   public Order processOrder(Order order);
}
```

The `OrderProcessorImpl` will take orders, via the `processOrder()` method, check that the customer for the order has been processed, and if not have the customer processor process the new customer. Then will place the order, which means will just set the order id and processed attribute to true.

The most important point to this example is that the `Order` class does NOT implement `java.io.Serializable`.

Now onto the `Server` class. This is just like the previous `Server` class in the basic example with one main difference: the `locatorURI` value.

```
public class Server
{
   private String locatorURI = "socket://localhost:5400/?serializationtype=jboss";
   private TransporterServer server = null;

   public void start() throws Exception
   {
      server = TransporterServer.createTransporterServer(locatorURI, new OrderProcessorImpl());
   }

   public void stop()
   {
      if(server != null)
      {
         server.stop();
      }
   }

   public static void main(String[] args)
   {
      Server server = new Server();
      try
      {
         server.start();

         Thread.currentThread().sleep(60000);

      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
      finally
      {
         server.stop();
      }
   }
}
```

The addition of `serializationtype=jboss` tells the remoting framework to use JBoss Serialization in place of the standard java serialization.

On the client side, there is the `Client` class, just as in the previous basic example.

```
public class Client
{
   private String locatorURI = "socket://localhost:5400/?serializationtype=jboss";
```

```
   public void makeClientCall() throws Exception
   {
      Order order = createOrder();

      OrderProcessor orderProcessor = (OrderProcessor) TransporterClient.createTransporterClient(locatorU

      System.out.println("Order to be processed: " + order);
      Order changedOrder = orderProcessor.processOrder(order);
      System.out.println("Order now processed " + changedOrder);

      TransporterClient.destroyTransporterClient(orderProcessor);

   }

   private Order createOrder()
   {
      Order order = new Order();
      Customer customer = createCustomer();
      order.setCustomer(customer);

      List items = new ArrayList();
      items.add("Xbox 360");
      items.add("Wireless controller");
      items.add("Ghost Recon 3");

      order.setItems(items);

      return order;
   }

   private Customer createCustomer()
   {
      Customer cust = new Customer();
      cust.setFirstName("Bob");
      cust.setLastName("Smith");
      Address addr = new Address();
      addr.setStreet("101 Oak Street");
      addr.setCity("Atlanata");
      addr.setState("GA");
      addr.setZip(30249);
      cust.setAddr(addr);

      return cust;
   }

   public static void main(String[] args)
   {
      Client client = new Client();
      try
      {
         client.makeClientCall();
      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
   }
}
```

Again, the biggest difference to note is that have added `serializationtype=jboss` to the locator uri.

Note: Running this example requires JDK 1.5.

To run this example, can run the Server and then the Client. Or can go to the examples directory and run the ant tar-

get 'ant run-transporter-serialization-server' and then in another window run the ant target 'ant run-transporter-serialization-client'. For example:

```
ant run-transporter-serialization-server
```

and then:

```
ant run-transporter-serialization-client
```

When the server and client are run the output for the `Client` class is:

```
Order to be processed:
Order:

Is processed: false
Order id: -1
Customer:
customer id: -1
first name: Bob
last name: Smith
street: 101 Oak Street
city: Atlanata
state: GA
zip: 30249

Items ordered:
Xbox 360
Wireless controller
Ghost Recon 3
Order now processed
Order:

Is processed: true
Order id: 221
Customer:
customer id: 861
first name: Bob
last name: Smith
street: 101 Oak Street
city: Atlanata
state: GA
zip: 30249

Items ordered:
Xbox 360
Wireless controller
Ghost Recon 3
```

The client output shows the printout of the newly created order before calling the `OrderProcessor` and then the processed order afterwards. Noticed that the processed order has its customer's id set, its order id set and the processed attribute is set to true.

And the output from the `Server` is:

```
Incoming order to process from customer.

Customer:
customer id: -1
first name: Bob
last name: Smith
street: 101 Oak Street
```

```
city: Atlanata
state: GA
zip: 30249

processed customer with new id of 861
Items ordered:
Xbox 360
Wireless controller
Ghost Recon 3
Order processed.  Order id now: 221
```

The server output shows the printout of the customer before being processed and then the order while being processed.

11.8.5. Transporter sample - clustered

In the previous examples, there has been one and only one target POJO to make calls upon. If that target POJO was not available, the client call would fail. In the transporter clustered example (found in org.jboss.remoting.samples.transporter.clustered package), will show how to use the transporter in clustered mode so that if one target POJO becomes unavailable, the client call can be seamlessly failed over to another available target POJO on the network, regardless of network transport type.

This example uses the domain objects from the first, basic example, so only need to cover the client and server code. For this example, there are three different server classes. The first class is the `SocketServer` class, which is the exact same as the `Server` class in the basic example, except for the call to the `TransportServer`'s `createTransportServer()` method.

```
public class SocketServer
{
   public static String locatorURI = "socket://localhost:5400";
   private TransporterServer server = null;

   public void start() throws Exception
   {
      server = TransporterServer.createTransporterServer(getLocatorURI(), new CustomerProcessorImpl(),
                                                          CustomerProcessor.class.getName(), true);
   }

   protected String getLocatorURI()
   {
      return locatorURI;
   }

   public void stop()
   {
      if(server != null)
      {
         server.stop();
      }
   }

   public static void main(String[] args)
   {
      SocketServer server = new SocketServer();
      try
      {
         server.start();

         Thread.currentThread().sleep(60000);
```

```
      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
      finally
      {
         server.stop();
      }
   }
}
```

Notice that are now calling on the `TransportServer` to create a server with the locator uri and target POJO (`CustomerProcessorImpl`) as before, but have also added the interface type of the target POJO (`CustomerProcessor`) and that want clustering turned on (via the last `true` parameter).

The interface type of the target POJO is needed because this will be used as the subsystem within the remoting server for the target POJO. The subsystem value will be what the client uses to determine if discovered remoting server is for the target POJO they are looking for.

The transporter uses the MulticastDetector from JBoss Remoting for automatic discovery when in clustered mode. The actual detection of remote servers that come online can take up to a few seconds once started. There is a JNDI based detector provided within JBoss Remoting, but has not been integrated within the transporters yet.

The second server class is the `RMIServer` class. The `RMIServer` class extends the `SocketServer` class and uses a different locator uri to specify `rmi` as the transport protocol and a different port (`5500`).

```
public class RMIServer extends SocketServer
{
   private String localLocatorURI = "rmi://localhost:5500";

   protected String getLocatorURI()
   {
      return localLocatorURI;
   }

   public static void main(String[] args)
   {
      SocketServer server = new RMIServer();
      try
      {
         server.start();

         Thread.currentThread().sleep(60000);

      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
      finally
      {
         server.stop();
      }
   }
}
```

The last server class is the `HTTPServer` class. The `HTTPServer` class also extends the `SocketServer` class and specifies `http` as the transport protocol and `5600` as the port to listen for requests on.

```
public class HTTPServer extends SocketServer
{
   private String localLocatorURI = "http://localhost:5600";

   protected String getLocatorURI()
   {
      return localLocatorURI;
   }

   public static void main(String[] args)
   {
      SocketServer server = new HTTPServer();
      try
      {
         server.start();

         Thread.currentThread().sleep(60000);

      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
      finally
      {
         server.stop();
      }
   }
}
```

On the client side, there is only the `Client` class. This class is very similar to the one from the basic example. The main exceptions are (1) the addition of a `TransporterClient` call to create a transporter client and (2) the fact that it continually loops, making calls on its `customerProcessor` variable to process customers. This is done so that when we run the client, we can kill the different servers and see that the client continues to loop making its calls without any exceptions or errors.

```
public class Client
{
   private String locatorURI = SocketServer.locatorURI;

   private CustomerProcessor customerProcessor = null;

   public void makeClientCall() throws Exception
   {
      Customer customer = createCustomer();

      System.out.println("Customer to be processed: " + customer);
      Customer processedCustomer = customerProcessor.processCustomer(customer);
      System.out.println("Customer is now: " + processedCustomer);

      //TransporterClient.destroyTransporterClient(customerProcessor);
   }

   public void getCustomerProcessor() throws Exception
   {
      customerProcessor = (CustomerProcessor) TransporterClient.createTransporterClient(locatorURI, Custo
   }

   private Customer createCustomer()
   {
```

```
      Customer cust = new Customer();
      cust.setFirstName("Bob");
      cust.setLastName("Smith");
      Address addr = new Address();
      addr.setStreet("101 Oak Street");
      addr.setCity("Atlanata");
      addr.setState("GA");
      addr.setZip(30249);
      cust.setAddr(addr);

      return cust;
   }

   public static void main(String[] args)
   {
      Client client = new Client();
      try
      {
         client.getCustomerProcessor();
         while(true)
         {
            try
            {
               client.makeClientCall();
               Thread.currentThread().sleep(5000);
            }
            catch(Exception e)
            {
               e.printStackTrace();
            }
         }
      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
   }
}
```

The first item of note is that the locator uri from the `SocketServer` class is being used. Technically, this is not required as once the clustered `TransporterClient` is started, it will start to discover the remoting servers that exist on the network. However, this process can take several seconds to occur, so unless it is known that no calls will be made on the remote proxy right away, it is best to bootstrap with a known target server.

Can also see that in the `main()` method, the first call on the Client instance is to `getCustomerProcessor()`. This method will call the `TransporterClient`'s `createTransporterClient()` method and passes the locator uri for the target POJO server, the type of POJO's remote proxy, and that clustering should be enabled.

After getting the customer processor remote proxy, will continually loop making calls using the remote proxy (via the `processCustomer()` method on the `customerProcessor` variable).

To run this example, all the servers need to be started (by running the `SocketServer`, `RMIServer`, and `HTTPServer` classes). Then run the Client class. This can be done via ant targets as well. So for example, could open four console windows and enter the ant targets as follows:

```
ant run-transporter-clustered-socket-server
```

```
ant run-transporter-clustered-http-server
```

```
ant run-transporter-clustered-rmi-server
```

```
ant run-transporter-clustered-client
```

Once the client starts running, should start to see output logged to the `SocketServer`, since this is the one used to bootstrap. This output would look like:

```
processed customer with new id of 378
processed customer with new id of 487
processed customer with new id of 980
```

Once the `SocketServer` instance has received a few calls, kill this instance. The next time the client makes a call on its remote proxy, which happens every five seconds, it should fail over to another one of the servers (and will see similar output on that server instance). After that server has received a few calls, kill it and should see it fail over once again to the last server instance that is still running. Then, if kill that server instance, will see a CannotConnectException and stack trace similar to the following:

```
...
org.jboss.remoting.CannotConnectException: Can not connect http client invoker.
 at org.jboss.remoting.transport.http.HTTPClientInvoker.useHttpURLConnection(HTTPClientInvoker.java:147)
 at org.jboss.remoting.transport.http.HTTPClientInvoker.transport(HTTPClientInvoker.java:56)
 at org.jboss.remoting.RemoteClientInvoker.invoke(RemoteClientInvoker.java:112)
 at org.jboss.remoting.Client.invoke(Client.java:226)
 at org.jboss.remoting.Client.invoke(Client.java:189)
 at org.jboss.remoting.Client.invoke(Client.java:174)
 at org.jboss.remoting.transporter.TransporterClient.invoke(TransporterClient.java:219)
 at $Proxy0.processCustomer(Unknown Source)
 at org.jboss.remoting.samples.transporter3.client.Client.makeClientCall(Client.java:29)
 at org.jboss.remoting.samples.transporter3.client.Client.main(Client.java:64)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
 at java.lang.reflect.Method.invoke(Method.java:585)
 at com.intellij.rt.execution.application.AppMain.main(AppMain.java:86)
Caused by: java.net.ConnectException: Connection refused: connect
 at java.net.PlainSocketImpl.socketConnect(Native Method)
 at java.net.PlainSocketImpl.doConnect(PlainSocketImpl.java:333)
 at java.net.PlainSocketImpl.connectToAddress(PlainSocketImpl.java:195)
 at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:182)
 at java.net.Socket.connect(Socket.java:507)
 at java.net.Socket.connect(Socket.java:457)
 at sun.net.NetworkClient.doConnect(NetworkClient.java:157)
 at sun.net.www.http.HttpClient.openServer(HttpClient.java:365)
 at sun.net.www.http.HttpClient.openServer(HttpClient.java:477)
 at sun.net.www.http.HttpClient.<init>(HttpClient.java:214)
 at sun.net.www.http.HttpClient.New(HttpClient.java:287)
 at sun.net.www.http.HttpClient.New(HttpClient.java:299)
 at sun.net.www.protocol.http.HttpURLConnection.getNewHttpClient(HttpURLConnection.java:792)
 at sun.net.www.protocol.http.HttpURLConnection.plainConnect(HttpURLConnection.java:744)
 at sun.net.www.protocol.http.HttpURLConnection.connect(HttpURLConnection.java:669)
 at sun.net.www.protocol.http.HttpURLConnection.getOutputStream(HttpURLConnection.java:836)
 at org.jboss.remoting.transport.http.HTTPClientInvoker.useHttpURLConnection(HTTPClientInvoker.java:117)
 ... 14 more
```

since there are no target servers left to make calls on. Notice that earlier in the client output there were no errors while was failing over to the different servers as they were being killed.

Because the CannotConnectException is being caught within the while loop, the client will continue to try calling the remote proxy and getting this exception. Now re-run any of the previously killed servers and will see that the client will discover that server instance and begin to successfully call on that server. The output should look something like:

```
...
 at sun.net.www.protocol.http.HttpURLConnection.connect(HttpURLConnection.java:669)
 at sun.net.www.protocol.http.HttpURLConnection.getOutputStream(HttpURLConnection.java:836)
 at org.jboss.remoting.transport.http.HTTPClientInvoker.useHttpURLConnection(HTTPClientInvoker.java:117)
 ... 14 more

Customer to be processed:
Customer:
customer id: -1
first name: Bob
last name: Smith
street: 101 Oak Stree
city: Atlanata
state: null
zip: 30249

Customer is now:
Customer:
customer id: 633
first name: Bob
last name: Smith
street: 101 Oak Stree
city: Atlanata
state: null
zip: 30249

...
```

As demonstrated in this example, fail over can occur across any of the JBoss Remoting transports. Clustered transporters is also supported using JBoss Serialization, which was introduced in the previous example.

It is important to understand that in the context of transporters, clustering means invocation fail over. The JBoss Remoting transporters themselves do not handle any form of state replication. If this feature were needed, could use JBoss Cache to store the target POJO instances so that when their state changed, that change would be replicated to the other target POJO instances running in other processes.

11.8.6. Transporters sample - multiple

The multiple transporter example (found in org.jboss.remoting.samples.transporter.multiple package) shows how can have a multiple target pojos exposed via the same TransporterServer. In this example, will be two pojos being exposed, CustomerProcessorImpl and AccountProcessorImpl. Since the domain objects for this example is similar to the others discussed in previous examples, will just focus on the server and client code. On the server side, need to create the TransporterServer so that will included both of the target pojos.

```
public class Server
{
   private String locatorURI = "socket://localhost:5400";
   private TransporterServer server = null;

   public void start() throws Exception
   {
      server = TransporterServer.createTransporterServer(locatorURI, new CustomerProcessorImpl(), Custome
      server.addHandler(new AccountProcessorImpl(), AccountProcessor.class.getName());
   }

   public void stop()
```

```
   {
      if(server != null)
      {
         server.stop();
      }
   }

   public static void main(String[] args)
   {
      Server server = new Server();
      try
      {
         server.start();

         Thread.currentThread().sleep(60000);

      }
      catch(Exception e)
      {
         e.printStackTrace();
      }
      finally
      {
         server.stop();
      }
   }
}
```

The TransporterServer is created with the CustomerProcessorImpl as the inital target pojo. Now that have a live TransporterServer, can add other pojos as targets. This is done using the addHandler() method where the target pojo instance is passed and then the interface type to be exposed as.

Next have the Client that makes the call to both pojos.

```
public class Client
{
   private String locatorURI = "socket://localhost:5400";

   public void makeClientCall() throws Exception
   {
      Customer customer = createCustomer();

      CustomerProcessor customerProcessor = (CustomerProcessor) TransporterClient.createTransporterClient

      System.out.println("Customer to be processed: " + customer);
      Customer processedCustomer = customerProcessor.processCustomer(customer);
      System.out.println("Customer is now: " + processedCustomer);

      AccountProcessor accountProcessor = (AccountProcessor) TransporterClient.createTransporterClient(lo

      System.out.println("Asking for a new account to be created for customer.");
      Account account = accountProcessor.createAccount(processedCustomer);
      System.out.println("New account: " + account);

      TransporterClient.destroyTransporterClient(customerProcessor);
      TransporterClient.destroyTransporterClient(accountProcessor);

   }

   private Customer createCustomer()
   {
      Customer cust = new Customer();
      cust.setFirstName("Bob");
      cust.setLastName("Smith");
```

```
      Address addr = new Address();
      addr.setStreet("101 Oak Street");
      addr.setCity("Atlanta");
      addr.setState("GA");
      addr.setZip(30249);
      cust.setAddr(addr);

      return cust;
   }

   public static void main(String[] args)
   {
      org.jboss.remoting.samples.transporter.multiple.client.Client client = new org.jboss.remoting.sampl
      try
      {
         client.makeClientCall();
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }


}
```

Notice that TransporterClients are created for each target pojo want to call upon, they just happen to share the same locator uri. These are independant instances so need to both be destroyed on their own when finished with them.

To run this example, run the Server class and then the Client class. This can be done via ant targets 'run-transporter-multiple-server' and then 'run-transporter-multiple-client'. For example:

```
ant run-transporter-multiple-server
```

and then:

```
ant run-transporter-multiple-client
```

The output for the server should look similar to:

```
processed customer with new id of 980
Created new account with account number: 1 and for customer:

Customer:
customer id: 980
first name: Bob
last name: Smith
street: 101 Oak Street
city: Atlanta
state: GA
zip: 30249
```

and the output from the client should look similar to:

```
Customer to be processed:
Customer:
customer id: -1
first name: Bob
last name: Smith
street: 101 Oak Street
```

```
city: Atlanta
state: GA
zip: 30249

Customer is now:
Customer:
customer id: 980
first name: Bob
last name: Smith
street: 101 Oak Street
city: Atlanta
state: GA
zip: 30249

Asking for a new account to be created for customer.
New account: Account - account number: 1
Customer:
Customer:
customer id: 980
first name: Bob
last name: Smith
street: 101 Oak Street
city: Atlanta
state: GA
zip: 30249
```

11.8.7. Transporters sample - proxy

The proxy transporter example (found in org.jboss.remoting.samples.transporter.proxy package) shows how can have a TransporterClient sent over the network and called upon. In this example, will have a target pojo, CustomerProcessorImpl which itself creates a TransporterClient to another target pojo, Customer, and return it as response to a method invocation.

To start, will look at the initial target pojo, CustomerProcessorImpl.

```
public class CustomerProcessorImpl implements CustomerProcessor
{
   private String locatorURI = "socket://localhost:5401";

   /**
    * Takes the customer passed, and if not null and customer id
    * is less than 0, will create a new random id and set it.
    * The customer object returned will be the modified customer
    * object passed.
    *
    * @param customer
    * @return
    */
   public ICustomer processCustomer(Customer customer)
   {
      if (customer != null && customer.getCustomerId() < 0)
      {
         customer.setCustomerId(new Random().nextInt(1000));
      }

      ICustomer customerProxy = null;
      try
      {
         TransporterServer server = TransporterServer.createTransporterServer(locatorURI, customer, ICust
         customerProxy = (ICustomer) TransporterClient.createTransporterClient(locatorURI, ICustomer.clas
      }
      catch (Exception e)
```

```
      {
         e.printStackTrace();
      }

      System.out.println("processed customer with new id of " + customerProxy.getCustomerId());
      return customerProxy;
   }

}
```

Notice that the processCustomer() method will take a Customer object and set customer id on it. Then it will create a TransporterServer for that customer instance and also create a TransporterClient for the same instance and return that TransporterClient proxy as the return to the processCustomer() method.

Next will look at the Customer class. It is a basic data object in that is really just stores the customer data.

```
public class Customer implements Serializable, ICustomer
{
   private String firstName = null;
   private String lastName = null;
   private Address addr = null;
   private int customerId = -1;

   public String getFirstName()
   {
      return firstName;
   }

   public void setFirstName(String firstName)
   {
      this.firstName = firstName;
   }

   public String getLastName()
   {
      return lastName;
   }

   public void setLastName(String lastName)
   {
      this.lastName = lastName;
   }

   public Address getAddr()
   {
      return addr;
   }

   public void setAddr(Address addr)
   {
      this.addr = addr;
   }

   public int getCustomerId()
   {
      return customerId;
   }

   public void setCustomerId(int customerId)
   {
      this.customerId = customerId;
   }

   public String toString()
```

```
   {
      System.out.println("Customer.toString() being called.");
      StringBuffer buffer = new StringBuffer();
      buffer.append("\nCustomer:\n");
      buffer.append("customer id: " + customerId + "\n");
      buffer.append("first name: " + firstName + "\n");
      buffer.append("last name: " + lastName + "\n");
      buffer.append("street: " + addr.getStreet() + "\n");
      buffer.append("city: " + addr.getCity() + "\n");
      buffer.append("state: " + addr.getState() + "\n");
      buffer.append("zip: " + addr.getZip() + "\n");

      return buffer.toString();
   }


}
```

Notice the toString() method and how it prints out to the standard out when being called. This will be important when the sample is run later.

Now if look at the Server class, will see is a standard setup like have seen in previous samples.

```
public class Server
{
   private String locatorURI = "socket://localhost:5400";
   private TransporterServer server = null;

   public void start() throws Exception
   {
      server = TransporterServer.createTransporterServer(locatorURI, new CustomerProcessorImpl(), Custome
   }

   public void stop()
   {
      if (server != null)
      {
         server.stop();
      }
   }

   public static void main(String[] args)
   {
      Server server = new Server();
      try
      {
         server.start();

         Thread.currentThread().sleep(60000);

      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
      finally
      {
         server.stop();
      }
   }
}
```

It is creating a TransporterServer for the CustomerProcessImpl upon being started and will wait 60 seconds for in-

vocations.

Next is the Client class.

```
public class Client
{
   private String locatorURI = "socket://localhost:5400";

   public void makeClientCall() throws Exception
   {
      Customer customer = createCustomer();

      CustomerProcessor customerProcessor = (CustomerProcessor) TransporterClient.createTransporterClient

      System.out.println("Customer to be processed: " + customer);
      ICustomer processedCustomer = customerProcessor.processCustomer(customer);
      // processedCustomer returned is actually a proxy to the Customer instnace
      // that lives on the server.  So when print it out below, will actually
      // be calling back to the server to get the string (vi toString() call).
      // Notice the output of 'Customer.toString() being called.' on the server side.
      System.out.println("Customer is now: " + processedCustomer);

      TransporterClient.destroyTransporterClient(customerProcessor);


   }

   private Customer createCustomer()
   {
      Customer cust = new Customer();
      cust.setFirstName("Bob");
      cust.setLastName("Smith");
      Address addr = new Address();
      addr.setStreet("101 Oak Street");
      addr.setCity("Atlanta");
      addr.setState("GA");
      addr.setZip(30249);
      cust.setAddr(addr);

      return cust;
   }

   public static void main(String[] args)
   {
      Client client = new Client();
      try
      {
         client.makeClientCall();
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }


}
```

The client class looks similar to the other example seen in that it creates a TransporterClient for the CustomerProcessor and calls on it to process the customer. Will then call on the ICustomer instance returned from the processCustomer() method call and call toString() on it (in the system out call).

To run this example, run the Server class and then the Client class. This can be done via ant targets 'run-

transporter-proxy-server' and then 'run-transporter-proxy-client'. For example:

```
ant run-transporter-proxy-server
```

ant then:

```
ant run-transporter-proxy-client
```

The output for the client should look similar to:

```
Customer.toString() being called.
Customer to be processed:
Customer:
customer id: -1
first name: Bob
last name: Smith
street: 101 Oak Street
city: Atlanta
state: GA
zip: 30249

Customer is now:
Customer:
customer id: 418
first name: Bob
last name: Smith
street: 101 Oak Street
city: Atlanta
state: GA
zip: 30249
```

The first line is the print out from calling the Customer's toString() method that was created to be passed to the CustomerProcessor's processCustomer() method. Then the contents of the Customer object before being processed. Then have the print out of the customer after has been processed. Notice that when the ICustomer object instance is printed out the second time, do not see the 'Customer.toString() being called'. This is because that code is no longer being executed in the client vm, but instead is a remote call to the customer instance living on the server (remember, the processCustomer() method returned a TransporterClient proxy to the customer living on the server side).

Now, if look at output from the server will look similar to:

```
processed customer with new id of 418
Customer.toString() being called.
```

Notice that the 'Customer.toString() being called.' printed out at the end. This is the result of the client's call to print out the contents of the customer object returned from the processCustomer() method, which actually lives within the server vm.

This example has shown how can pass around TransporterClient proxies to target pojos. However, when doing this, is important to understand where the code is actually being executed as there are consequences to being remote verse local, which need to be understood.

11.8.8. Transporter sample -complex

The complex transporter example (found in org.jboss.remoting.samples.transporter.complex package) is based off a test case a user, Milt Grinberg, provided (thanks Milt). The example is similar to the previous examples, except in this case involves matching Doctors and Patients using the ProviderInterface and provides a more complex sample in which to demonstrate how to use transporters.

This example requires JDK 1.5 to run, since is using JBoss Serialization (and non-serialized data objects). To run this example, run the Server class and then the Client class. This can be done via ant targets 'run-transporter-complex-server' and then 'run-transporter-complex-client' as well. For example:

```
ant run-transporter-complex-server
```

and then:

```
ant run-transporter-complex-client
```

The output for the client should look similar to:

```
*** Have a new patient that needs a doctor.  The patient is:

Patient:
   Name: Bill Gates
   Ailment - Type: financial, Description: Money coming out the wazoo.

*** Looking for doctor that can help our patient...

*** Found doctor for our patient.  Doctor found is:
Doctor:
   Name: Andy Jones
   Specialty: financial
   Patients:

Patient:
   Name: Larry Ellison
   Ailment - Type: null, Description: null
   Doctor - Name: Andy Jones

Patient:
   Name: Steve Jobs
   Ailment - Type: null, Description: null
   Doctor - Name: Andy Jones

Patient:
   Name: Bill Gates
   Ailment - Type: financial, Description: Money coming out the wazoo.

*** Set doctor as patient's doctor.  Patient info is now:

Patient:
   Name: Bill Gates
   Ailment - Type: financial, Description: Money coming out the wazoo.
   Doctor - Name: Andy Jones

*** Have a new patient that we need to find a doctor for (remember, the previous one retired and there ar
*** Could not find doctor for patient.  This is an expected exception when there are not doctors availabl
org.jboss.remoting.samples.transporter.complex.NoDoctorAvailableException: No doctor available for ailmen
 at org.jboss.remoting.RemoteClientInvoker.invoke(RemoteClientInvoker.java:183)
 at org.jboss.remoting.Client.invoke(Client.java:325)
 at org.jboss.remoting.Client.invoke(Client.java:288)
 at org.jboss.remoting.Client.invoke(Client.java:273)
 at org.jboss.remoting.transporter.TransporterClient.invoke(TransporterClient.java:237)
 at $Proxy0.findDoctor(Unknown Source)
```

```
at org.jboss.remoting.samples.transporter.complex.client.Client.makeClientCall(Client.java:72)
at org.jboss.remoting.samples.transporter.complex.client.Client.main(Client.java:90)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:585)
at com.intellij.rt.execution.application.AppMain.main(AppMain.java:86)
```

From the output see the creation of a new patient, Bill Gates, and the attempt to find a doctor that specializes in his ailment. For Mr. Gates, we were able to find a doctor, Andy Jones, and can see that he has been added to the list of Dr. Jones' patients. Then we have Dr. Jones retire. Then we create a new patient and try to find an available doctor for the same ailment. Since Dr. Jones has retired, and there are no other doctors that specialize in that particular ailment, an exception is thrown. This is as expected.

11.9. Multiplex invokers

This section illustrates the construction of multiplex invoker groups described in the section Multiplex Invoker. The directory

```
examples/org/jboss/remoting/samples/multiplex/invoker
```

contains a server class, `MultiplexInvokerServer`, which is suitable for use with any of the client classes described below. It may be run in an IDE or from the command line using ant target `run-multiplex-server` from the `build.xml` file found in the `examples` directory. The server will stay alive, processing invocation requests as they are presented, until it has sent two push callbacks to however many listeners are registered, at which time it will shut itself down.

The sample clients are as follows. Each sample client <*client*> may be run in an IDE or by using the ant target `run-`<*client*> (e.g., `run-Client2Server1`).

- `Client2Server1`: A `MultiplexClientInvoker` starts according to client rule 2, after which a `MultiplexServerInvoker` is started according to server rule 1. Note that the `Client` and `Connector` are passed matching *clientMultiplexId* and *serverMultiplexId* parameters, respectively.
- `Client2Server2`: A `MultiplexClientInvoker` starts according to client rule 2, after which a `MultiplexServerInvoker` is started according to server rule 2. Note that no *clientMultiplexId* is passed to the `Client` and no *serverMultiplexId* parameter is passed to the `Connector` in this example.
- `Client3Server1`: A `MultiplexClientInvoker` is created, and, lacking binding information, finds itself governed by client rule 3. Subsequently, a `MultiplexServerInvoker` is started according to server rule 1, providing the binding information which allows the `MultiplexClientInvoker` to start. Note that the `Client` and `Connector` are passed matching *clientMultiplexId* and *serverMultiplexId* parameters, respectively.
- `Server2Client1`: A `MultiplexServerInvoker` starts according to server rule 2, after which a `MultiplexClientInvoker` is started according to client rule 1. Note that the `Connector` and `Client` are passed matching *serverMultiplexId* and *clientMultiplexId* parameters, respectively.
- `Server2Client2`: A `MultiplexServerInvoker` starts according to server rule 2, after which a `MultiplexCli-`

entInvoker is started according to client rule 2. Note that no *serverMultiplexId* is passed to the Connector and no *clientMultiplexId* parameter is passed to the Client in this example.

- Server3Client1: A MultiplexServerInvoker is created, and, lacking connect information, finds itself governed by server rule 3. Subsequently, a MultiplexClientInvoker is started according to client rule 1, providing the connect information which allows the MultiplexServerInvoker to start. Note that the Connector and Client are passed matching *serverMultiplexId* and *clientMultiplexId* parameters, respectively.

For variety, the examples in which the client invoker starts first use the configuration Map to pass invoker group parameters, and the examples in which the server invoker starts first pass parameters in the InvokerLocator.

12

Client programming model

The approach taken for the programming model on the client side is one based on a session based model. This means that it is expected that once a Client is created for a particular target server, it will be used exclusively to make calls on that server. This expectation dictates some of the behavior of the remoting client.

For example, if create a Client on the client side to make server invocations, including adding callback listeners, will have to use that same instance of Client to remove the callback listeners. This is because the Client creates a unique session id that it passes within the calls to the server. This id is used as part of the key for registering callback listeners on the server. If create a new Client instance and attempt to remove the callback listeners, a new session id will be passed to the server invoker, who will not recognize the callback listener to be removed.

See test case `org.jboss.test.remoting.callback.push.MultipleCallbackServersTestCase` .

13

Compatibility and versioning

As of JBossRemoting 2.0.0 versioning has been added to guarantee compatibility between different versions. This is accomplished by changing serialization formats for certain classes and by using wire versioning. By wire versioning, mean that the version used by a client and server will be sent on the wire so that the other side will be able to adjust accordingly. This will be automatic for JBossRemoting 2.0.0 and later versions. However, since versioning was not introduced until the 2.0.0 release, if need to have a 1.4.x version of remoting communicate to a later version, will need to set a system property on the 2.0.0 version so that knows to use the older wire protocol version. The system property to set is 'jboss.remoting.pre_2_0_compatible' and should be set to true. There are a few minor features that will not be fully compatible between 1.4.x release and 2.0.0, which are listed in the release notes.

14

Getting the JBossRemoting source and building

The JBossRemoting source code resides in the JBoss CVS repository under the CVS module JBossRemoting. To check out the source using the anonymous account, use the following command:

```
cvs -d:pserver:anonymous@anoncvs.forge.jboss.com:/cvsroot/jboss checkout JBossRemoting
```

To check out the source using a committer user id, use the following:

```
cvs -d:ext:username@cvs.forge.jboss.com:/cvsroot/jboss checkout JBossRemoting
```

This should checkout the entire remoting project, including doc, tests, libs, etc.

See http://www.jboss.org/wiki/Wiki.jsp?page=CVSRepository [http://www.jboss.org/wiki/Wiki.jsp?page=CVSRepository] for more information on how to access the JBoss CVS repository.

The build process for JBossRemoting is based on a standard ant build file (build.xml). The version of ant that is supported is ant 1.6.2, but should work with earlier versions as there are no special ant features being used.

The main ant build targets are as follows:

compile - compiles all the core JBossRemoting classes.

jars - creates the jboss-remoting.jar file from the compiled classes

dist.jars - creates the subsystem jar files (jboss-remoting-core.jar, jboss-remoting-socket.jar, etc.) from the compiled classes

javadoc - creates the javadoc html files for JBossRemoting

tests.compile - compiles the JBossRemoting test files

tests.jars - creates the jboss-remoting-tests.jar and jboss-remoting-loading-tests.jar files.

tests.quick - runs the functional unit tests for JBossRemoting.

tests - runs all the tests for JBossRemoting, including functional and performance tests for all the different transports.

clean - removes all the build artifacts and directories.

most - calls clean then jars targets.

dist - builds the full JBossRemoting distribution including running the full test suite.

dist.quick - builds the full JBossRemoting distribution, but does not run the test suite.

The root directory for all build output is the output directory. Under this directory will be:

`classes` - compiled core classes

`etc` - deployment and JMX XMBean xml files

`lib` - all the jars and war file produced by the build

`tests` - contains the compiled test classes and test results

For most development, the most target can be used. Please run the tests.quick target before checking anything in to ensure that code changes did not break any previously functioning test.

15

Known issues

All of the known issues and road map can be found on our bug tracking system, Jira, at http://jira.jboss.com/jira/secure/BrowseProject.jspa?id=10031
[http://jira.jboss.com/jira/secure/BrowseProject.jspa?id=10031] (require member plus registration, which is free). If you find more, please post them to Jira. If you have questions post them to the JBoss Remoting users forum (http://www.jboss.com/index.html?module=bb&op=viewforum&f=222
[http://www.jboss.com/index.html?module=bb&op=viewforum&f=222]).

16

Future plans

Full road map for JBossRemoting can be found at http://jira.jboss.com/jira/browse/JBREM?report=com.atlassian.jira.plugin.system.project:roadmap-panel [http://jira.jboss.com/jira/browse/JBREM?report=com.atlassian.jira.plugin.system.project:roadmap-panel].

If you have questions, comments, bugs, fixes, contributions, or flames, please post them to the JBoss Remoting users forum (http://www.jboss.com/index.html?module=bb&op=viewforum&f=222 [http://www.jboss.com/index.html?module=bb&op=viewforum&f=222]). You can also find more information about JBoss Remoting on our wiki (http://www.jboss.org/wiki/Wiki.jsp?page=Remoting [http://www.jboss.org/wiki/Wiki.jsp?page=Remoting]). The wiki will usually contain the latest updates to doc and features that did not make into previous release.

17

Release Notes

Important changes and differences in 2.2.0 release (from 2.0.0 release)

- Asynchronous method for handling callbacks (JBREM-640)

- Bidirectional transport (JBREM-650)

- Local transport (JBREM-660)

- Marshallers/Unmarshallers construct their preferred streams (JBREM-692)

==========

Release Notes - JBoss Remoting - Version 2.2.3.SP1

Bug

* [JBREM-1077] - Fix problem in CompressingMarshaller

* [JBREM-1101] - Low: Missing header validation in org.jboss.remoting.transport.servlet.ServletServerInvoker

* [JBREM-1105] - DefaultLoadBalancer doesn't use all available server (if there are x servers available it uses x-1 of these)

* [JBREM-1107] - SocketServerInvoker.IdleTimeoutTask interrupts ServerThreads during long invocation

* [JBREM-1143] - CLONE [JBREM-877] - New Socket Connection is being Created for Every Client Request to the Server

* [JBREM-1145] - HTTPUnMarshaller shouldn't use the value of content-type to determine the type of an object

* [JBREM-1147] - BisocketClientInvoker.createSocket() in callback mode should check for replaced control socket

* [JBREM-1148] - Remove log.info() calls from ServletServerInvoker

* [JBREM-1150] - Lease should update client list if PING invocation has same time as previous PING)

* [JBREM-1152] - If MicroSocketClientInvoker gets a SocketException in getConnection(), it should retry

* [JBREM-1154] - Client.connect() should throw an exception when Lease creation fails

Feature Request

* [JBREM-1120] - Add a socket write timeout facility

* [JBREM-1139] - Modify PortUtil to allow a configurable range for MIN_UNPRIVILEGED_PORT and MAX_LEGAL_PORT.

* [JBREM-1146] - Treat IOException("Connection reset by peer") as a retriable exception

Release

* [JBREM-1157] - Release 2.2.3.SP1

Task

* [JBREM-1142] - Clarify use of "clientConnectAddress" in RemotingGuide

* [JBREM-1151] - Correct dependency discussion in Chapter 4 of Remoting Guide

* [JBREM-1156] - Assure version compatibility with earlier versions of Remoting2

==========

Release Notes - JBoss Remoting - Version 2.2.3

Bug

* [JBREM-1081] - Fix NPE in ServerInvokerCallbackHandler

* [JBREM-1088] - MicroSocketClientInvoker(InvokerLocator locator, Map configuration) ctor - not propagating exceptions (only message is wrapped)

* [JBREM-1099] - Make MulticastDetector detection message send buffer size configurable

* [JBREM-1109] - Eliminate race in MicroRemoteClientInvoker.getDataType()

* [JBREM-1111] - CLONE [JBREM-851] - In LeasePinger replace Timer if it has shut down

* [JBREM-1112] - Potential race between ConnectionValidator and ConnectionListener upon connection failure

* [JBREM-1119] - CLONE [JBREM-1113] - ServerInvokerCallbackHandlers leak when client doesn't shut down

* [JBREM-1121] - Client SocketFactory should be configurable by InvokerLocator

* [JBREM-1125] - Test for IllegalStateException when calling Timer.schedule()

* [JBREM-1127] - ClassCastExceptions arising from Unmarsharller/Classloader caching in MicroRemoteClientInvoker

* [JBREM-1129] - Eliminate nondeterminism in Lease updates

Feature Request

* [JBREM-1082] - Allow ConnectionValidator to access InvokerLocator parameters

* [JBREM-1084] - Allow CallbackPoller to access Client and InvokerLocator parameters

* [JBREM-1100] - Link ServerInvokerServlet instances to Connectors via MBean names rather than locator URLs

* [JBREM-1102] - Make configuration map available to MarshalFactory

* [JBREM-1128] - Introduce connection identity concept

Release

* [JBREM-1131] - Release 2.2.3

Task

* [JBREM-139] - need automated test for servlet server invoker

* [JBREM-1078] - Remove reference to "numberOfRetries" parameter from Remoting Guide

* [JBREM-1079] - Port fixes to ServletServerInvoker from branch 2.x to branch 2.2

* [JBREM-1085] - Reduce log level of ServerSocketWrapper.close() log messages

* [JBREM-1103] - Correct javadoc for Client.invokeOneway()

* [JBREM-1104] - Identity.get() should create a more meaningful RuntimeException message

* [JBREM-1108] - Warn against making ConnectionValidator.validatorPingPeriod shorter than ConnectionValidator.validatorPingTimeout

* [JBREM-1110] - InvokerLocator.getParameters() should not return null

* [JBREM-1130] - Assure version compatibility with earlier versions of Remoting

==========

Release Notes - JBoss Remoting - Version 2.2.2.SP11

Bug

* [JBREM-1041] - Remote classloading does not work with Isolated EARs, Part 2

* [JBREM-1042] - CLONE [JBREM-1019] - RemotingClassLoader needs option to delegate to user class loader before parent

* [JBREM-1045] - ServerInvokerCallbackHandler can suffer deadlock when used with BlockingCallbackStore

* [JBREM-1050] - HTTPClientInvoker does not support BASIC authentication for proxies when use of proxy is configured by system properties

* [JBREM-1052] - CLONE [JBREM-1046] - HTTPClientInvoker throws NullPointerException when HttpURLConnection.getInputStream() returns null

* [JBREM-1055] - ConnectionValidator.run() should have a sanity test to prevent calls from application code

* [JBREM-1056] - Fix race condition in InvokerRegistry

* [JBREM-1069] - Make ConnectorValidator configure ping period correctly

* [JBREM-1070] - Fix deadlock in ConnectionValidator

* [JBREM-1071] - IllegalStateException in ConnectorValidator.run()

* [JBREM-1072] - Synchronize access to static maps in MarshalFactory

* [JBREM-1073] - Allow invocations to stopped servers to throw CannotConnectExceptions

Release

* [JBREM-1039] - Release 2.2.2.SP11

Task

* [JBREM-1075] - Assure version compatibility with earlier versions of Remoting

==========

Release Notes - JBoss Remoting - Version 2.2.2.SP10

Bug

* [JBREM-925] - SSLSocketBuilder config attribute names out of sync with docs

* [JBREM-1021] - JavaSerializationManager should not log exceptions at ERROR level

* [JBREM-1022] - Fix misleading javadoc in InvokerLocator

* [JBREM-1027] - CoyoteInvoker should pass URL query in InvocationRequest

* [JBREM-1037] - CLONE [JBREM-1006] - SOA MTOM bug points to bug in Remoting codebase

Release

* [JBREM-1020] - Release 2.2.2.SP10

Task

* [JBREM-1038] - Assure version compatibility with earlier versions of Remoting

==========

Release Notes - JBoss Remoting - Version 2.2.2.SP9

Bug

* [JBREM-1005] - Prevent build up of cancelled TimerTasks in bisocket transport

* [JBREM-1014] - Support injection of socket factory class name into AbstractInvoker

Feature Request

* [JBREM-1010] - Add feature to declaratively turn on leasing for 2.2.2

Release

* [JBREM-1011] - Release 2.2.2.SP9

Task

* [JBREM-1013] - Assure version compatibility with earlier versions of Remoting

* [JBREM-1017] - Improve socket timeout log message

==========

Release Notes - JBoss Remoting - Version 2.2.2.SP8

Bug

* [JBREM-949] - CLONE [JBREM-947] - ConnectionValidator hangs when server dies

* [JBREM-954] - InterruptedException should not be rethrown as CannotConnectionException

* [JBREM-960] - Remoting configured with Servlet invoker can return misleading Exceptions when Servlet path is incorrect

* [JBREM-962] - Remote classloading does not work with Isolated EARs

* [JBREM-965] - Fix PortUtil.getRandomStartingPort()

* [JBREM-981] - CLONE [JBREM-980] - ServerInvokerServlet should retrieve ServletServerInvoker based on updated InvokerLocator

* [JBREM-1003] - Verify IPv6 addresses are handled correctly, part 2

Feature Request

* [JBREM-972] - CLONE [JBREM-971] - Enhance client-side connection error handling so certain (potentially revealing) socket-related exceptins are not discarded

* [JBREM-973] - CLONE [JBREM-970] - Enhance client-side error reporting so a misspelled truststore file name required by SSL can be easily spotted

Release

* [JBREM-948] - Release 2.2.2.SP8

Task

* [JBREM-950] - Assure version compatibility with earlier versions of Remoting

* [JBREM-995] - Apply unit test timing fixes

* [JBREM-1001] - Update Remoting Guide

* [JBREM-1002] - Allow ServerThread to keep running after SocketTImeoutException, part 2

* [JBREM-1004] - Run manual servlet unit tests

==
========================

Release Notes - JBoss Remoting - Version 2.2.2.SP7

Bug

* [JBREM-942] - A deadlock encountered on ConnectionValidator

* [JBREM-944] - Fix race in ConnectionNotifier

Release

* [JBREM-943] - Release 2.2.0.SP7

Task

* [JBREM-945] - Allow ServerThread to keep running after SocketTImeoutException

* [JBREM-946] - Assure version compatibility with earlier versions of Remoting

==
========================

Release Notes - JBoss Remoting - Version 2.2.2.SP6

Bug

* [JBREM-915] - NullPointerException in InvokerLocator

* [JBREM-937] - Callback BisocketServerInvoker should reuse available ServerThreads

Release

* [JBREM-939] - Release 2.2.2.SP6

Task

* [JBREM-940] - Assure version compatibility with earlier versions of Remoting

==
========================

Release Notes - JBoss Remoting - Version 2.2.2.SP5

Bug

* [JBREM-892] - CLONE -Client side connection exception is not thrown on the client side when the lease times out [JBREM-888]

* [JBREM-910] - CLONE -Connector.stop() cannot find invoker MBean when bind address is 0.0.0.0 [JBREM-909]

Release

* [JBREM-913] - Release 2.2.2.SP5

Task

* [JBREM-912] - Remove stacktrace when SSLSocketBuilder.createSSLSocketFactory() fails

==
========================

Release Notes - JBoss Remoting - Version 2.2.2.SP4

** Bug

* [JBREM-823] - ServerInvoker#getMBeanObjectName() returns invalid ObjectName if host value is IPv6

* [JBREM-845] - Infinite loop in BisocketClientInvoker.createSocket

* [JBREM-858] - MaxPoolSize value should be used in key to MicroSocketClientInvoker.connectionPools

* [JBREM-860] - Eliminate delay in MicroSocketClientInvoker.getConnection()

* [JBREM-871] - HTTP Client invoker doesn't throw exceptions when using the sslservlet protocol

** Feature Request

* [JBREM-852] - Verify IPv6 addresses are handled correctly

* [JBREM-855] - Update build.xml to allow jdk 1.5 compiler to target JVM version 1.4

* [JBREM-873] - Have ServerInvokerCallbackHandler register as connection listener

** Release

* [JBREM-872] - Release 2.2.0.SP4

** Task

* [JBREM-862] - Verify compatibility with earlier versions

==
========================

N.B. Release 2.2.2.SP4 replaces 2.2.2.SP3.

==
========================

Release Notes - JBoss Remoting - Version 2.2.2.SP2

Bug

* [JBREM-811] - Privileged Block to create Class Loader

* [JBREM-813] - ServletServerInvoker should return an exception instead of just an error message

Release

* [JBREM-817] - Release 2.2.2.SP2

Task

* [JBREM-687] - allow binding to 0.0.0.0

==
========================

Release Notes - JBoss Remoting - Version 2.2.2.SP1

** Bug

* [JBREM-653] - allow user to set content-type for http responses

* [JBREM-750] - Logger in HTTPClientInvoker should be static.

** Release

* [JBREM-803] - Release 2.2.2.SP1

** Task

* [JBREM-805] - Verify Remoting 2.2.2.SP1 is compatible with earlier versions

==
========================

Release Notes - JBoss Remoting - Version 2.2.2.GA

** Bug

* [JBREM-731] - Address of secondary server socket should be acquired each time a control connection is created.

* [JBREM-743] - For polling callback handler, org.jboss.remoting.Client.addListener() should create only one CallbackPoller per InvokerCallbackHandler

* [JBREM-747] - org.jboss.remoting.transport.Connector should unregister server invoker from MBeanServer

* [JBREM-754] - Reset timeout on each use of HttpURLConnection

* [JBREM-761] - NPE in BisocketServerInvoker$ControlConnectionThread

* [JBREM-766] - Guard against "spurious wakeup" from Thread.sleep()

* [JBREM-771] - MicroSocketClientInvoker can experience socket leaks

* [JBREM-774] - BisocketClientInvoker.replaceControlSocket() and handleDisconnect() should close control socket

* [JBREM-775] - MicroSocketClientInvoker.initPool() should omit pool from log message

* [JBREM-778] - BisocketServerInvoker.start() creates a new static Timer each time

* [JBREM-779] - BisocketClientInvoker should guard agains scheduling on an expired Timer, part 2

* [JBREM-784] - Use separate maps for control sockets and ordinary sockets in BisocketClientInvoker

* [JBREM-785] - BisocketClientInvoker.transport() inadvertently uses listenerId member variable

* [JBREM-787] - Move network i/o in BisocketClientInvoker constructor to handleConnect()

* [JBREM-788] - Access to BisocketClientInvoker static maps should be synchronized in handleDisconnect()

* [JBREM-790] - NPE in BisocketClientInvoker$PingTimerTask

* [JBREM-793] - Lease should synchronize access to client map

* [JBREM-794] - LeasePinger.addClient() should not create a new LeaseTimerTask if none currently exists

* The following is the public version of support patch JBREM-791, under which the fix was applied. -RS

* [JBREM-806] - In HTTPClientInvoker remove newlines and carriage returns from Base64 encoded user names and passwords

** Feature Request

* [JBREM-749] - BisocketServerInvoker: Make configurable the address and port of secondary server socket

* [JBREM-755] - Make ConnectorValidator parameters configurable

* [JBREM-756] - CallbackPoller should shut down if too many errors occur.

* [JBREM-757] - Implement quick Client.removeListener() for polled callbacks.

* [JBREM-765] - Add a separate timeout parameter for callback clients

** Patch

* [JBREM-781] - Socket transport needs to provide to the client local address of a TCP/IP connection, as seen from the server

** Release

* [JBREM-789] - Release 2.2.2.GA

** Task

* [JBREM-641] - re-implement the callback polling for http transport to reduce latency

* [JBREM-767] - Avoid deadlock in callback BisocketClientInvoker when timeout == 0

* [JBREM-782] - Remove network i/o from synch block in ServerInvokerCallbackHandler.getCallbackHandler()

* [JBREM-783] - Remove network i/o from synch blocks that establish and terminate LeasePingers

* [JBREM-796] - Verify Remoting 2.2.2 is compatible with earlier versions

==========

Release Notes - JBoss Remoting - Version 2.2.1.GA

** Bug

* [JBREM-751] - Eliminate unnecessary "Unable to process control connection:" message from BisocketServerInvoker

** Release

* [JBREM-763] - Release 2.2.1.GA

==========

Release Notes - JBoss Remoting - Version 2.2.0.SP4

** Bug

* [JBREM-748] - BisocketClientInvoker should guard agains scheduling on an expired Timer

** Release

* [JBREM-744] - Release 2.2.0.SP4

** Task

* [JBREM-714] - Make sure 2.2.0 and 2.0.0 are compatible binary releases

* [JBREM-734] - BisocketClientInvoker constructor should get parameters from InvokerLocator as well as configuration map.

==========

Release Notes - JBoss Remoting - Version 2.2.0.SP3

** Task

* [JBREM-741] - Eliminate unnecessary log.warn() in BisocketServerInvoker

Release Notes - JBoss Remoting - Version 2.2.0.SP2

** Bug

* [JBREM-739] - Fix java serialization leak. [Note. This issue has been moved to 2.4.0.Beta1 pending the addition of unit tests, but the bug has been fixed.]

Release Notes - JBoss Remoting - Version 2.2.0.SP1

** Bug

* [JBREM-732] - When server terminates and has clients, when the server comes back up clients that survived, can't connect. Connection refused when trying to connect the control socket.

Release Notes - JBoss Remoting - Version 2.2.0.GA (Bluto)

** Bug
* [JBREM-721] - Fix memory leaks in bisocket transport and LeasePinger

* [JBREM-722] - BisocketClientInvoker should start pinging on control connection without waiting for call to createsocket()

* [JBREM-725] - NPE in BisocketServeInvoker::createControlConnection

* [JBREM-726] - BisocketServerInvoker control connection creation needs to be in loop

** Feature Request

* [JBREM-705] - Separate the http invoker and web container dependency

* [JBREM-727] - Make Client's implicitly created Connectors accessible

** Task * [JBREM-634] - update doc on callbacks

* [JBREM-724] - Update build.xml to create bisocket transport jars

Release Notes - JBoss Remoting - Version 2.2.0.Beta1 (Bluto)

** Bug
* [JBREM-581] - can not do connection validation with ssl transport (only impacts detection)

* [JBREM-600] - org.jboss.test.remoting.lease.multiplex.MultiplexLeaseTestCase fails

* [JBREM-623] - need reset() call added back to JavaSerializationManager.sendObject() method

* [JBREM-642] - Socket.setReuseAddress() in MicroSocketClientInvoker invocation is ignored

* [JBREM-648] - Client.disconnect without clearing ConnectionListeners will cause NPEs

* [JBREM-651] - Array class loading problem under jdk6

* [JBREM-654] - a NullPointerException occures and is not handled in SocketServerInvoker and MultiplexServer-

Invoker

* [JBREM-655] - rename server thread when new socket connection comes in

* [JBREM-656] - Creating a client inside a ConnectionListener might lead into Lease reference counting problems

* [JBREM-658] - bug in oneway thread pool under heavy load

* [JBREM-659] - Java 6 and ClassLoader.loadClass()

* [JBREM-670] - Remove equals() and hashCode() from org.jboss.remoting.transport.rmi.RemotingRMIClientSocketFactory.

* [JBREM-671] - serlvet invoker no longer supports leasing

* [JBREM-683] - ByValueInvocationTestCase is broken

* [JBREM-685] - A server needs redundant information to detect a one way invocation

* [JBREM-690] - Once the socket of a callback server timeouts, it starts to silently discard traffic

* [JBREM-697] - Horg.jboss.remoting.transport.rmi.RemotingRMIClientSocketFactory.ComparableHolder should use InetAddress for host.

* [JBREM-700] - NPE in AbstractDetector

* [JBREM-704] - BisocketServerInvoker inadvertently logs "got listener: null" as INFO

* [JBREM-708] - Correct org.jboss.remoting.Client.readExternal()

* [JBREM-711] - ChunkedTestCase and Chuncked2TestCase failing

* [JBREM-712] - HTTPInvokerProxyTestCase failing

* [JBREM-723] - BisocketClientInvoker.transport() needs to distinguish between push and pull callback connections

** Feature Request

* [JBREM-525] - Automatically set HostnameVerifier in HTTPSClientInvoker to allow all hosts if authorization is turned off.

* [JBREM-598] - add timeout config per client invocation

* [JBREM-618] - Support CallbackPoller configuration.

* [JBREM-640] - Implement an asynchronous method for handling callbacks.

* [JBREM-650] - Create bidirectional transport

* [JBREM-657] - Implement versions of Client.removeListener() and Client.disconnect() which do not write to a broken server.

* [JBREM-660] - create local transport

* [JBREM-664] - Fix misleading InvalidConfigurationException

* [JBREM-692] - Let marshallers/unmarshallers construct their preferred streams.

* [JBREM-720] - Need to expose create method for TransporterClient that passes load balancing policy

** Task

* [JBREM-274] - add callback methods to the Client API

* [JBREM-369] - For Connectors that support callbacks on SSL connections, there should be a unified means of configuring SSLServerSocket and callback Client SSLSocket.s.

* [JBREM-453] - Send the pre-release jar to the messaging team for testing

* [JBREM-614] - Client.invoke() should check isConnected().

* [JBREM-631] - Fix org.jboss.test.remoting.transport.socket.connection.SocketConnectionCheckTestCase and SocketConnectionTestCase failures.

* [JBREM-635] - Remove misleading error message from HTTPUnMarshaller.

* [JBREM-636] - Remove ServerInvokerCallbackHandler's dependence on initial InvocationRequest for listerner id.

* [JBREM-637] - add tomcat jar to component-info.xml for remoting release

* [JBREM-644] - Reduce unit test logging output.

* [JBREM-647] - Initialize Client configuration map to empty HashMap.

* [JBREM-663] - Put org.jboss.remoting.LeasePinger on separate thread.

* [JBREM-669] - Client.removeListener() should catch exception and continue if invocation to server fails.

* [JBREM-674] - add test case for client exiting correctly

* [JBREM-693] - Make sure "bisocket" can fully replace "socket" as Messaging's default transport

* [JBREM-695] - RemotingRMIClientSocketFactory.createSocket() should return a socket even if a RMIClientInvoker has not been registered.

* [JBREM-702] - http.basic.password should allow for empty passwords

* [JBREM-707] - Fix handling of OPTIONS invocations in CoyoteInvoker

* [JBREM-709] - Fix occasional failures of org.jboss.test.remoting.lease.socket.multiple.SocketLeaseTestCase

* [JBREM-719] - Fix spelling of ServerInvokerCallbackHandler.REMOTING_ACKNOWLEDGES_PUSH_CALLBACKS

Release Notes - JBoss Remoting - Version 2.2.0.Alpha6

** Bug
* [JBREM-662] - Failed ClientInvoker not cleaned up properly

* [JBREM-673] - Use of java.util.Timer recently added and not set to daemon, so applications not exiting

* [JBREM-683] - ByValueInvocationTestCase is broken

** Feature Request

* [JBREM-678] - Sending an one-way invocation into a server invoker that is not started should generate a warning in logs

* [JBREM-679] - Add the possibility to obtain ConnectionValidator's ping period from a Client

* [JBREM-680] - An invocation into a "broken" client should throw a subclass of IOException

** Task

* [JBREM-676] - TimerTasks run by TimerUtil should have a chance to clean up if TimerUtil.destroy() is called.

Release Notes - JBoss Remoting - Version 2.2.0.Alpha5

** Bug

* [JBREM-666] - Broken or malicious clients can lock up the remoting server

* [JBREM-667] - Worker thread names are confusing

** Feature Request

* [JBREM-668] - jrunit should allow TRACE level logging

Release Notes - JBoss Remoting - Version 2.2.0.Alpha4

** Bug

* [JBREM-649] - Concurrent exceptions on Lease when connecting/disconnecting new Clients

Release Notes - JBoss Remoting - Version 2.2.0.Alpha3 (Bluto)

** Bug

* [JBREM-594] - invoker not torn down upon connector startup error

* [JBREM-596] - Lease stops working if the First Client using the same Locator is closed

* [JBREM-602] - If LeasePeriod is not set and if enableLease==true leasePeriod assumes negative value

* [JBREM-610] - Prevent org.jboss.remoting.callback.CallbackPoller from delivering callbacks out of order.

* [JBREM-611] - Initializing Client.sessionId outside constructor leads to java.lang.NoClassDefFoundError in cer-

tain circumstances

* [JBREM-615] - If CallbackStore.add() is called twice quickly, System.currentTimeMillis() might not change, leading to duplicate file names.

* [JBREM-616] - Deletion of callback files in getNext() is not synchronized, allowing callbacks to be returned multiple times.

* [JBREM-619] - In SocketServerInvoker.run() and MultiplexServerInvoker().run, guarantee ServerSocketRefresh thread terminates.

* [JBREM-622] - InvokerLocator already exists for listener

* [JBREM-625] - MicroSocketClientInvoker should decrement count of used sockets when a socket is discarded.

* [JBREM-629] - NPE in sending notification of lost client

** Feature Request

* [JBREM-419] - Invokers Encryption

* [JBREM-429] - Create JBossSerialization MarshalledValue more optimized for RemoteCalls

* [JBREM-548] - Support one way invocations with no response

* [JBREM-597] - Allow access to underlying stream in marshaller with socket transport

* [JBREM-604] - allow socket server invoker to accept third party requests

* [JBREM-605] - Inform a server side listener that a callback has been delivered.

* [JBREM-607] - Add idle timeout setting for invoker threads

* [JBREM-609] - Support nonserializable callbacks in CallbackStore

** Task

* [JBREM-562] - publish performance benchmarks

* [JBREM-601] - Integrate http with messaging

* [JBREM-612] - Verify push callback connection with multiplex transport shares client to server connection.

* [JBREM-613] - ServerInvoker.InvalidStateException should be a static class.

* [JBREM-617] - CallbackPoller should have its own thread.

* [JBREM-620] - If HTTPClientInvoker receives an Exception in an InvocationRespose, it should throw it instead of creating a new Exception.

* [JBREM-621] - http transport should behave more like other transports.

* [JBREM-624] - Add JBoss EULA

* [JBREM-627] - Fix org.jboss.test.remoting.transport.multiplex.MultiplexInvokerShutdownTestCase failure.

* [JBREM-630] - Fix client/server race in org.jboss.test.remoting.transport.multiplex.LateClientShutdownTestCase.

* [JBREM-632] - Modify src/etc/log4j.xml to allow DEBUG level logging for org.jboss.remoting loggers in jrunit test cases.

Release Notes - JBoss Remoting - Version 2.0.0.GA (Boon)

** Bug

* [JBREM-568] - SSLSocketBuilderMBean does not have matching getter/setter attribute types

* [JBREM-569] - HTTP(S) proxy broken

* [JBREM-576] - deadlock with socket invoker

* [JBREM-579] - transporter does not handle reflection conversion for primitive types

* [JBREM-580] - detection can not be used with ssl based transports

* [JBREM-586] - socket client invoker connection pooling not bounded

* [JBREM-590] - SSL client socket invoker does not use configuration map for SSLSocketBuilder

** Feature Request

* [JBREM-564] - Default client socket factory configured by a system property

* [JBREM-575] - local client invoker should convert itself to remote client invoker when being serialized

** Task

* [JBREM-570] - Change log in ConnectionValidator to be debug instead of warn when not able to ping server

* [JBREM-571] - fix/cleanup doc

* [JBREM-574] - Write SSL info for virtual sockets and server sockets in toString()

* [JBREM-578] - add spring remoting to performance benchmark tests

* [JBREM-582] - remove System.out.println and printStackTrace calls

* [JBREM-583] - Fix ConcurrentModificationException in MultiplexingManager.notifySocketsOfException()

* [JBREM-584] - Get org.jboss.test.remoting.performance.spring.rmi.SpringRMIPerformanceTestCase to run with multiple clients and callback handlers

* [JBREM-587] - ClientConfigurationCallbackConnectorTestCase(jboss_serialization) failure.

* [JBREM-593] - Synchronize client and server in

org.jboss.test.remoting.transport.multiplex.LateClientShutdownTestCase

Release Notes - JBoss Remoting - Version 2.0.0.CR1 (Boon)

** Bug

* [JBREM-303] - org.jboss.test.remoting.transport.multiplex.BasicSocketTestCase(jboss_serialization) failure

* [JBREM-387] - classloading problem - using wrong classloader

* [JBREM-468] - No connection possible after an illegitimate attempt

* [JBREM-484] - AbstractDetector.checkInvokerServer() is probably broken

* [JBREM-494] - ClientDisconnectedException does not have serial version UID

* [JBREM-495] - classes that do not have serial version UID

* [JBREM-500] - ServerThread never dies

* [JBREM-502] - not getting REMOVED notification from registry for intra-VM detection

* [JBREM-503] - NPE in abstract detector

* [JBREM-506] - StreamHandler throws index out of bounds exception

* [JBREM-508] - serialization exception with mustang

* [JBREM-519] - StreamServer never shuts down the server

* [JBREM-526] - TimeUtil not using daemon thread

* [JBREM-528] - ConcurrentModificationException when checking for dead servers (AbstractDetector)

* [JBREM-530] - Detection heartbeat requires small timeout (for dead server detection)

* [JBREM-534] - multiplex client cannot re-connect to server after it has died and then been re-started

* [JBREM-537] - org.jboss.test.remoting.transport.rmi.ssl.handshake.RMIInvokerTestCase(java_serialization) - failing

* [JBREM-541] - null pointer when receiving detection message

* [JBREM-545] - setting of the bind address within MulticastDetector not working

* [JBREM-546] - InvokerLocator.equals is broken

* [JBREM-552] - cannot init cause of ClassCastException

* [JBREM-553] - deadlock when disconnecting

* [JBREM-556] - versioning tests failing

* [JBREM-561] - http chuncked test cases failing under jdk 1.5

** Feature Request

* [JBREM-427] - SSL Connection: load a new keystore at runtime

* [JBREM-430] - transporter needs to be customizable

* [JBREM-461] - Better documentation for sslmultiplex

* [JBREM-491] - need to implement using ssl client mode for push callbacks for all transports

* [JBREM-492] - would like an API to indicate if a transport requires SSL configuration

* [JBREM-499] - need indication if invoker is secured by ssl

* [JBREM-501] - give descriptive names to threads

* [JBREM-504] - some synch blocks in AbstractDetector could change

* [JBREM-520] - Organize configuring of ServerSocketFactory's and callback SocketFactory's.

* [JBREM-527] - Allow user to pass Connector to be used for stream server

* [JBREM-532] - need synchronous call from detector client to get all remoting servers on network

* [JBREM-539] - add sslservlet procotol

* [JBREM-544] - http client invoker (for http, https, servlet, and sslservlet) needs to handle exceptions in same manner as other transport implementations

** Task

* [JBREM-21] - Add stress tests

* [JBREM-218] - investigate why jrunit report on cruisecontrol inaccurate

* [JBREM-311] - need required library matrix

* [JBREM-320] - optimize pass by value within remoting

* [JBREM-321] - performance tuning

* [JBREM-368] - Configure SSLSockets and SSLServerSockets used in callbacks to be in server mode and client mode, respectively.

* [JBREM-383] - Document new versioning for remoting

* [JBREM-384] - correct manifest to comply with new standard

* [JBREM-390] - finish multiplex

* [JBREM-412] - Remoting Guide lacks left margin

* [JBREM-423] - document how remoting identity works and how to configure

* [JBREM-428] - add the samples/transporter/multiple/ to the distribution build (think may be there by default) and update the docs

* [JBREM-434] - fix configuration data within document (socketTimeout should be timeout)

* [JBREM-435] - break out remoting jars (serialization)

* [JBREM-442] - need full doc on how socket invoker works (connection pooling, etc.)

* [JBREM-447] - convert static transporter factory methods into constructor calls

* [JBREM-452] - Send the pre-release jar to the messaging team for testing

* [JBREM-454] - cache socket wrapper classes

* [JBREM-477] - remove Client.setInvoker() and Client.getInvoker() methods

* [JBREM-487] - Eliminate possible synchronization problem in InvokerRegistry

* [JBREM-490] - consolidate the remoting security related classes

* [JBREM-493] - Update version of jboss serialization being used

* [JBREM-496] - restructure service providers for remoting

* [JBREM-497] - change InvokerLocator to respect hostname over ip address

* [JBREM-498] - change logging on cleaning up failed detection

* [JBREM-507] - need to make configuration properties consistent

* [JBREM-509] - Fix call to super() in ServerInvoker's two argument constructor.

* [JBREM-511] - Allow HTTPSClientInvoker to create a HostnameVerifier from classname.

* [JBREM-513] - Create SSL version of RMI transport.

* [JBREM-514] - Fix potential NullPointerException in SSLSocketClientInvoker.createSocket().

* [JBREM-516] - add very simple transporter sample

* [JBREM-517] - HTTPServerInvoker needs to be deprecated

* [JBREM-523] - connection pool on socket client invoker needs to be bound

* [JBREM-524] - Clean up MicrosocketClientInvoker code

* [JBREM-529] - Need to be able to reuse socket connections after move to TIME_WAIT state

* [JBREM-533] - remove external http GET test

* [JBREM-535] - add config to force use of remote invoker instead of local

* [JBREM-536] - turn off host verification when doing push callback from server using same ssl config as server

* [JBREM-538] - update remoting dist build to break out transports into individual jars

* [JBREM-540] - need to make servlet-invoker.war part of remoting distribution

* [JBREM-542] - change how remoting servlet finds servlet invoker

* [JBREM-543] - fix servlet invoker error handling to be more like that of the http invoker

* [JBREM-547] - need test case for exposing multiple interfaces for transporter server target pojo

* [JBREM-551] - org.jboss.test.remoting.transport.multiplex.MultiplexInvokerTestCase(java_serialization) failure

* [JBREM-555] - fix connection validator to not require extra thread to execute ping every time

* [JBREM-558] - Break master.xml documentation into chapter files

* [JBREM-559] - update doc for 2.0.0.CR1 release

* [JBREM-560] - InvokerGroupTestCase(java_serialization) failure

* [JBREM-563] - Multiplex ClientConfigurationCallbackConnectorTestCase(jboss_serialization) failure

Release Notes - JBoss Remoting - Version 2.0.0.Beta2 (Boon)

** Bug

* [JBREM-304] - org.jboss.test.remoting.transport.multiplex.MultiplexInvokerTestCase(java_serialization) fails

* [JBREM-371] - HTTPClientInvoker does not pass an ObjectOutputStream to the marshaller

* [JBREM-405] - NPE when calling stop() twice on MulticastDetector

* [JBREM-406] - StringIndexOutOfBoundsException in InvokerLocator

* [JBREM-408] - client lease updates broken on server side

* [JBREM-409] - Invocations fail when the pool exhausts and under heavy load

* [JBREM-414] - JNDI detection failing

* [JBREM-418] - ObjectInputStreamWithClassLoader can't handle primitives

* [JBREM-426] - keyStorePath and keyStorePassword being printed to standard out

* [JBREM-432] - TransporterClient missing serialVersionUID

* [JBREM-440] - CallbackStore.getNext() won't necessarily get the oldest one

* [JBREM-441] - DefaultCallbackErrorHandler.setConfig needs to avoid NPE

* [JBREM-449] - Failure Information lost in RemotingSSLSocketFactory

* [JBREM-450] - ClassNotFoundException for class array type during deserialization

* [JBREM-464] - ssl socket invoker not using ssl server socket factory

* [JBREM-467] - NPE when calling Client.removeConnectionListener()

* [JBREM-470] - javax.net.ssl.SSLException: No available certificate corresponds to the SSL cipher suites

* [JBREM-472] - Misspelled serialization type generates obscure NPE

* [JBREM-479] - ClientConfigurationMapTestCase failure

* [JBREM-482] - client invoker configuration lost after first time invoker is created

** Feature Request

* [JBREM-312] - make TransporterClient so can be sent over network as dynamic proxy

* [JBREM-363] - make callbacks easier with richer API for registering for callbacks

* [JBREM-411] - Add chunked streaming support to the HTTP invoker

* [JBREM-413] - Transporter server should allow multiple pojo targets

* [JBREM-422] - Add plugable load balancing policy to transporter client

* [JBREM-425] - Add support for setting the HTTP invoker content encoding that is accepted

* [JBREM-431] - transporter server should automatically expose all interfaces implemented as subsystems

* [JBREM-439] - StreamInvocationHandler.handleStream should throw Throwable for consistency

* [JBREM-469] - Enable HTTP polling

* [JBREM-471] - need better InvokerLocator.equals() implementation

* [JBREM-481] - Changing StringUtilBuffer creation on JBossSerialization

** Task

* [JBREM-299] - MultiplexInvokerTestCase failure

* [JBREM-314] - need org.jboss.test.pooled.test.SSLSocketsUnitTestCase for remoting

* [JBREM-328] - change lease ping to be HEAD instead of POST for http transport

* [JBREM-362] - convert Connector to be standard mbean instead of xmbean

* [JBREM-365] - set default user agent header in http client invoker

* [JBREM-366] - clean up client invoker tracking within InvokerRegistry

* [JBREM-367] - set live server socket factory on Connector

* [JBREM-370] - add changes from 1.4.1 release to master.xml doc

* [JBREM-377] - need to convert ConnectionValidator to use TimerQueue

* [JBREM-379] - need to update jboss-serialization jar being used

* [JBREM-380] - change ConnectionValidator to only notify once of failure

* [JBREM-382] - disable lease ping for local invoker

* [JBREM-415] - sync bug fixes with pooled invoker and socket invoker

* [JBREM-420] - JNDI Detector should not need a connector when running in client mode

* [JBREM-421] - remote stream handler api inconsistent with regular handler

* [JBREM-436] - Extend MultiplexingInputStream with readInt() to avoid creating a MultiplexingDataInputStream in VirtualSocket.connect() and elsewhere.

* [JBREM-437] - Eliminate "verify connect" phase from virtual socket connection protocol.

* [JBREM-443] - add HandshakeCompletedListener support to ssl multiplex

* [JBREM-451] - Send the pre-release jar to the messaging team for testing

* [JBREM-455] - checking of socket connection is not really needed

* [JBREM-456] - block callback handling when callback store full

* [JBREM-460] - createSocket() in SSLSocketClientInvoker and SSLMultiplexClientInvoker should not assume SocketFactory has been created.

* [JBREM-465] - property setting on the client from locator parameters and config map

* [JBREM-476] - make externalization of Client match original instance state

* [JBREM-478] - fix local client invoker handling of disconnected server invokers

* [JBREM-483] - remove LocalLeaseTestCase

* [JBREM-485] - use the ClientInvokerHolder to contain the reference counting instead of having to use clientInvokerCounter

* [JBREM-486] - Fix ConcurrentModificationException in org.jboss.test.remoting.transport.mock.MockServerInvocationHandler

Release Notes - JBoss Remoting - Version 2.0.0.Beta1

** Bug

* [JBREM-372] - memory leak on server side leasing

* [JBREM-376] - problem versioning with not using connection checking

* [JBREM-378] - client connection checking not working

** Feature Request

* [JBREM-340] - Strong version compatibility guarantee

** Task

* [JBREM-374] - single thread the leasing timer

Release Notes - JBoss Remoting - Version 1.4.4.GA

** Bug

* [JBREM-426] - keyStorePath and keyStorePassword being printed to standard out

Release Notes - JBoss Remoting - Version 1.4.3.GA

** Bug

* [JBREM-418] - ObjectInputStreamWithClassLoader can't handle primitives

Release Notes - JBoss Remoting - Version 1.4.2 final

** Feature Request

* [JBREM-429] - Create JBossSerialization MarshalledValue more optimized for RemoteCalls

Release Notes - JBoss Remoting - Version 1.4.1 final

** Feature Request

* [JBREM-310] - Ability to turn connection checking off

* [JBREM-325] - move IMarshalledValue from jboss-commons to jboss-remoting.jar

** Bug

* [JBREM-313] - client lease does not work if client and server in same VM (using local invoker)

* [JBREM-317] - HTTPClientInvoker conect sends gratuitous POST

* [JBREM-341] - Client ping interval must be lease than lease period

* [JBREM-343] - Exceptions on connection closing

* [JBREM-345] - problem using client address and port

* [JBREM-346] - fix ConcurrentModificationException in cleanup of MultiplexServerInvoker

* [JBREM-350] - ConcurrentModificationException in InvokerRegistry

* [JBREM-361] - Race condition in invoking on Client

** Task

* [JBREM-2] - sample-bindings.xml does not have entry for remoting

* [JBREM-220] - clean up remoting wiki

* [JBREM-316] - Maintain tomcat originated code under the ASF license.

* [JBREM-319] - ability to inject socket factory by classname or instance in all remoting transports

* [JBREM-323] - client lease config changes

* [JBREM-329] - create global transport config for timeout

* [JBREM-330] - create socket server factory based off of configuration properties

* [JBREM-335] - Client.invoke() should pass configuration map to InvokerRegistry.createClientInvoker().

* [JBREM-336] - InvokerRegistry doesn't purge InvokerLocators from static Set registeredLocators.

* [JBREM-337] - PortUtil.findFreePort() should return ports only between 1024 and 65535.

* [JBREM-342] - Thread usage for timers and lease functionality

* [JBREM-354] - ServerInvokerCallbackHandler should make its subsystem accessible.

* [JBREM-356] - ServerInvoker should destroy its callback handlers.

* [JBREM-359] - MultiplexInvokerConfigTestCase should execute MultiplexInvokerConfigTestServer instead of MultiplexInvokerTestServer.

Release Notes - JBoss Remoting - Version 1.4.0 final

** Feature Request

* [JBREM-91] - UIL2 type transport (duplex calling of same socket)

* [JBREM-117] - clean up callback client after several failures delivering callbacks

* [JBREM-138] - HTTP/Servlet invokers require content length to be set

* [JBREM-229] - Remove dependency on ThreadLocal for SerializationManagers and pluggable serialization

* [JBREM-233] - Server side exception listeners for client connections

* [JBREM-257] - Append client stack trace to thrown remote exception

* [JBREM-261] - Integration with IMarshalledValue from JBossCommons

* [JBREM-278] - remoting detection needs ability to accept detection of server invoker running locally

* [JBREM-280] - no way to add path to invoker uri when using complex configuration

** Bug

* [JBREM-41] - problem using localhost/127.0.0.1

* [JBREM-115] - http server invoker does not wait to finish processing on stop

* [JBREM-223] - Broken Pipe if client don't do any calls before the timeout value

* [JBREM-224] - java.net.SocketTimeoutException when socket timeout on the keep alive

* [JBREM-231] - bug in invoker locator when there are no params (NPE)

* [JBREM-234] - StreamCorruptedException in DTM testcase

* [JBREM-240] - TestUtil does not always give free port for server

* [JBREM-243] - socket client invoker sharing pooled connections

* [JBREM-250] - InvokerLocator doesn't support URL in IPv6 format (ex: socket://3000::117:5400/)

* [JBREM-251] - transporter passes method signature based on concrete object and not the parameter type

* [JBREM-256] - NullPointer in MarshallerLoaderHandler.java:69

* [JBREM-259] - Unmarshalling of server response is not using caller's classloader

* [JBREM-271] - http client invoker needs to explicitly set the content type if not provided

* [JBREM-277] - error shutting down coyote invoker when using APR protocol

* [JBREM-281] - getting random port for connectors is not reliable

* [JBREM-282] - ServletServerInvoker not working with depployed for use as ejb invoker

* [JBREM-286] - Socket server does not clean up server threads on shutdown

* [JBREM-289] - PortUtil only checking for free ports on localhost

** Task

* [JBREM-7] - Add more tests for local invoker

* [JBREM-121] - improve connection failure callback

* [JBREM-126] - add tests for client vs. server address bindings

* [JBREM-195] - Performance optimization

* [JBREM-199] - remoting clients required to include servlet-api.jar

* [JBREM-207] - clean up build file

* [JBREM-214] - multiplex performance tests getting out of memory error

* [JBREM-215] - re-write http transport/handler documentation

* [JBREM-216] - Need to add new samples to example build in distro

* [JBREM-217] - create samples documentation

* [JBREM-219] - move remoting site to jboss labs

* [JBREM-226] - Release JBoss Remoting 1.4.0 final

* [JBREM-230] - create interface for marshallers to implement for swapping out serialization impl

* [JBREM-235] - add new header to source files

* [JBREM-239] - Update the LGPL headers

* [JBREM-242] - Subclass multiplex invoker from socket invoker.

* [JBREM-249] - http invoker (tomcat connector) documentation

* [JBREM-253] - Convert http server invoker implementation to use tomcat connector and protocols

* [JBREM-255] - HTTPClientInvoker not setting response code or message

* [JBREM-275] - fix package error in examle-service.xml

* [JBREM-276] - transporter does not throw original exception from server implementation

* [JBREM-279] - socket server invoker spits out error messages on shutdown when is not needed

* [JBREM-287] - need to complete javadoc for all user classes/interfaces

* [JBREM-288] - update example-service.xml with new configurations

** Reactor Event

* [JBREM-241] - Refactor SocketServerInvoker so that can be subclassed by MultiplexServerInvoker

Release Notes - JBoss Remoting - Version 1.4.0 beta

** Feature Request

* [JBREM-28] - Marshaller for non serializable objects

* [JBREM-40] - Compression marshaller/unmarshaller

* [JBREM-120] - config for using hostname in locator url instead of ip

* [JBREM-140] - can not set response headers from invocation handlers

* [JBREM-148] - support pluggable object serialization packages

* [JBREM-175] - Remove Dependencies to Server Classes from UnifiedInvoker

* [JBREM-180] - add plugable serialization

* [JBREM-187] - Better HTTP 1.1 stack support for HTTP invoker

* [JBREM-201] - Remove dependency from JBossSerialization

** Bug

* [JBREM-127] - RMI Invoker will not bind to specified address

* [JBREM-192] - distro contains samples in src and examples directory

* [JBREM-193] - HTTPClientInvoker doesn't call getErrorStream() on HttpURLConnection when an error response code is returned

* [JBREM-194] - multiplex performance tests hang

* [JBREM-202] - getUnmarshaller always calls Class.forName operation for creating Unmarshallers

* [JBREM-203] - rmi server invoker hangs if custom unmarshaller

* [JBREM-205] - Spurious java.net.SocketException: Connection reset error logging

* [JBREM-210] - InvokerLocator should be insensitive to parameter order

** Task

* [JBREM-9] - Fix performance tests

* [JBREM-33] - Add GET support within HTTP server invoker

* [JBREM-145] - convert user guide from MS word doc to docbook

* [JBREM-182] - Socket timeout too short (and better error message)

* [JBREM-183] - keep alive support for http invoker

* [JBREM-196] - reducde the number of retries for socket client invoker

* [JBREM-204] - create complex remoting example using dynamic proxy to endpoint

* [JBREM-212] - create transporter implementation

* [JBREM-213] - allow config of ignoring https host validation (ssl) via metadata

** Patch

* [JBREM-152] - NullPointerException in SocketServerInvoker.stop() at line 185.

* [JBREM-153] - LocalClientInvoker's outlive their useful lifetime, causing anomalous behavior

Release Notes - JBoss Remoting - Version 1.2.1 final

** Feature Request

* [JBREM-161] - Upgrade JRunit to Beta 2

** Bug

* [JBREM-147] - Invalid reuse of target location

* [JBREM-163] - NPE in Mutlicast Detector

* [JBREM-164] - HTTP Invoker unable to send large amounts of data

* [JBREM-176] - Correct inheritance structure for detectors

* [JBREM-177] - configuration attribute spelled incorrectly in ServerInvokerMBean

* [JBREM-178] - SocketServerInvoker hanging on Linux

* [JBREM-179] - socket timeout not being set properly

** Task

* [JBREM-156] - Better exception handling within socket server invoker

* [JBREM-158] - Clean up test cases

* [JBREM-162] - add version to the remoting jar

Release Notes - JBoss Remoting - Version 1.2.0 final

** Feature Request

* [JBREM-8] - Ability to stream files via remoting

* [JBREM-22] - Manipulation of the client proxy interceptor stack

* [JBREM-24] - Allow for specific network interface bindings

* [JBREM-27] - Support for HTTP/HTTPS proxy

* [JBREM-35] - Servlet Invoker - counterpart to HTTP Invoker (runs within web container)

* [JBREM-43] - custom socket factories

* [JBREM-46] - Connection failure callback

* [JBREM-87] - Add handler metadata to detection messages

* [JBREM-93] - Callback handler returning a generic Object

* [JBREM-94] - callback server specific implementation

* [JBREM-109] - Add support for JaasSecurityDomain within SSL support

* [JBREM-122] - need log4j.xml in examples

** Bug

* [JBREM-58] - Bug with multiple callback handler registered with same server

* [JBREM-64] - Need MarshalFactory to produce new instance per get request

* [JBREM-84] - Duplicate Connector shutdown using same server invoker

* [JBREM-92] - in-VM push callbacks don't work

* [JBREM-97] - Won't compile under JDK 1.5

* [JBREM-108] - can not set bind address and port for rmi and http(s)

* [JBREM-114] - getting callbacks for a callback handler always returns null

* [JBREM-125] - can not configure transport, port, or host for the stream server

* [JBREM-131] - invoker registry not update if server invoker changes locator

* [JBREM-134] - can not remove callback listeners from multiple callback servers

* [JBREM-137] - Invalid RemoteClientInvoker reference maintained by InvokerRegistry after invoker disconnect()

* [JBREM-141] - bug connecting client invoker when client detects that previously used one is disconnected

* [JBREM-143] - NetworkRegistry should not be required for detector to run on server side

** Task

* [JBREM-11] - Create seperate JBoss Remoting module in CVS

* [JBREM-20] - break out remoting into two seperate projects

* [JBREM-34] - Need to add configuration properties for HTTP server invoker

* [JBREM-39] - start connector on new thread

* [JBREM-55] - Clean up Callback implementation

* [JBREM-57] - Remove use of InvokerRequest in favor of Callback object

* [JBREM-62] - update UnifiedInvoker to use remote marshall loading

* [JBREM-67] - Add ability to set ThreadPool via configuration

* [JBREM-98] - remove isDebugEnabled() within code as is now depricated

* [JBREM-101] - Fix serialization versioning between releases of remoting

* [JBREM-104] - Release JBossRemoting 1.1.0

* [JBREM-110] - create jboss-remoting-client.jar

* [JBREM-113] - Convert remote tests to use JRunit instead of distributed test framework

* [JBREM-123] - update detection samples

* [JBREM-128] - standardize address and port binding configuration for all transports

* [JBREM-130] - updated wiki for checkout and build

* [JBREM-132] - write test case for JBREM-131

* [JBREM-133] - Document use of Client (as a session object)

* [JBREM-135] - Remove ClientInvokerAdapter

** Reactor Event

* [JBREM-65] - move callback specific classes into new callback package

* [JBREM-111] - pass socket's output/inputstream directly to marshaller/unmarshaller

Release Notes - JBoss Remoting - Version 1.0.2 final

** Bug

* [JBREM-36] - performance tests fail for http transports

* [JBREM-66] - Race condition on startup

* [JBREM-82] - Bad warning in Connector.

* [JBREM-88] - HTTP invoker only binds to localhost

* [JBREM-89] - HTTPUnMarshaller finishing read early

* [JBREM-90] - HTTP header values not being picked up on the http invoker server

** Task

* [JBREM-70] - Clean up build.xml. Fix .classpath and .project for eclipse

* [JBREM-83] - Updated Invocation marshalling to support standard payloads

Release Notes - JBoss Remoting - Version 1.0.1 final

** Feature Request

* [JBREM-54] - Need access to HTTP response headers

** Bug

* [JBREM-1] - Thread.currentThread().getContextClassLoader() is wrong

* [JBREM-31] - Exception handling in http server invoker

* [JBREM-32] - HTTP Invoker - check for threading issues

* [JBREM-50] - Need ability to set socket timeout on socket client invoker

* [JBREM-59] - Pull callback collection is unbounded - possible Out of Memory

* [JBREM-60] - Incorrect usage of debug level logging

* [JBREM-61] - Possible RMI exception semantic regression

** Task

* [JBREM-15] - merge UnifiedInvoker from remoting branch

* [JBREM-30] - Better integration for registering invokers with MBeanServe

* [JBREM-37] - backport to 4.0 branch before 1.0.1 final release

* [JBREM-56] - Add Callback object instead of using InvokerRequest

** Reactor Event

* [JBREM-51] - defining marshaller on remoting client

Release Notes - JBoss Remoting - Version 1.0.1 beta

** Bug

* [JBREM-19] - Try to reconnect on connection failure within socket invoker

* [JBREM-25] - Deadlock in InvokerRegistry

** Feature Request

* [JBREM-12] - Support for call by value

* [JBREM-26] - Ability to use MBeans as handlers

** Task

* [JBREM-3] - Fix Asyn invokers - currently not operable

* [JBREM-4] - Added test for throwing exception on server side

* [JBREM-5] - Socket invokers needs to be fixed

* [JBREM-16] - Finish HTTP Invoker

* [JBREM-17] - Add CannotConnectException to all transports

* [JBREM-18] - Backport remoting from HEAD to 4.0 branch

** Reactor Event

* [JBREM-23] - Refactor Connector so can configure transports

* [JBREM-29] - Over load invoke() method in Client so metadata not required