JBoss Remoting Guide

JBoss Remoting version 1.4.0 final

January 25, 2006

Table of Contents

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
  - HTTP(S)
  - Multiplex
  - Servlet
- **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

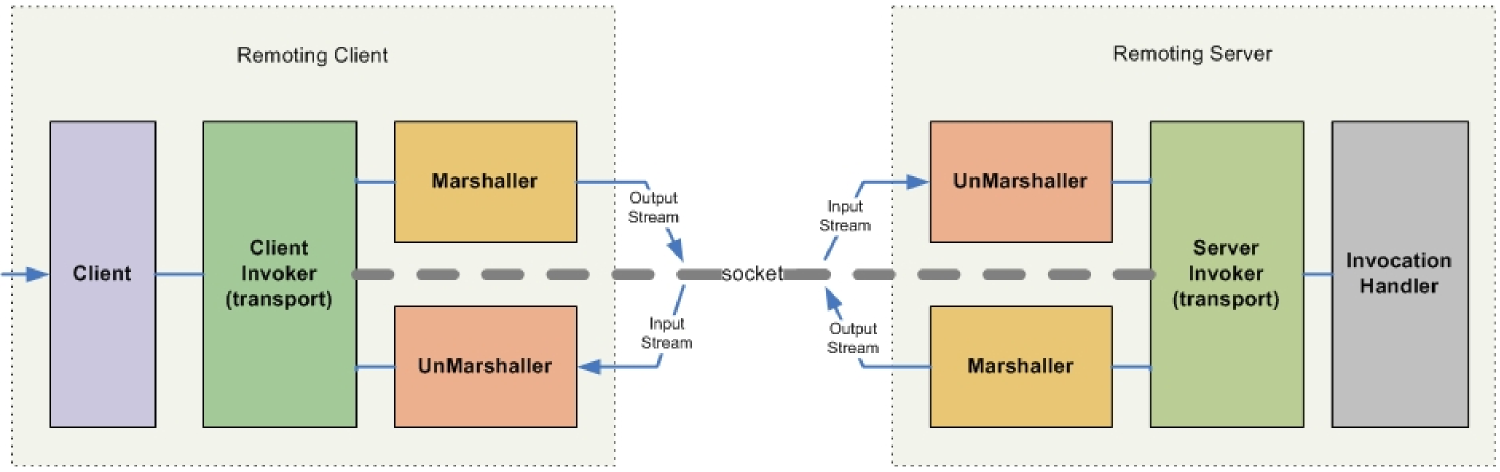
The JBossRemoting distribution can be downloaded from http://www.jboss.org/products/remoting [http://www.jboss.org/products/remoting] . This distribution contains everything needed to run JBossRemoting stand alone. The distribution includes binaries, source, documentation, javadoc, and sample code.

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.

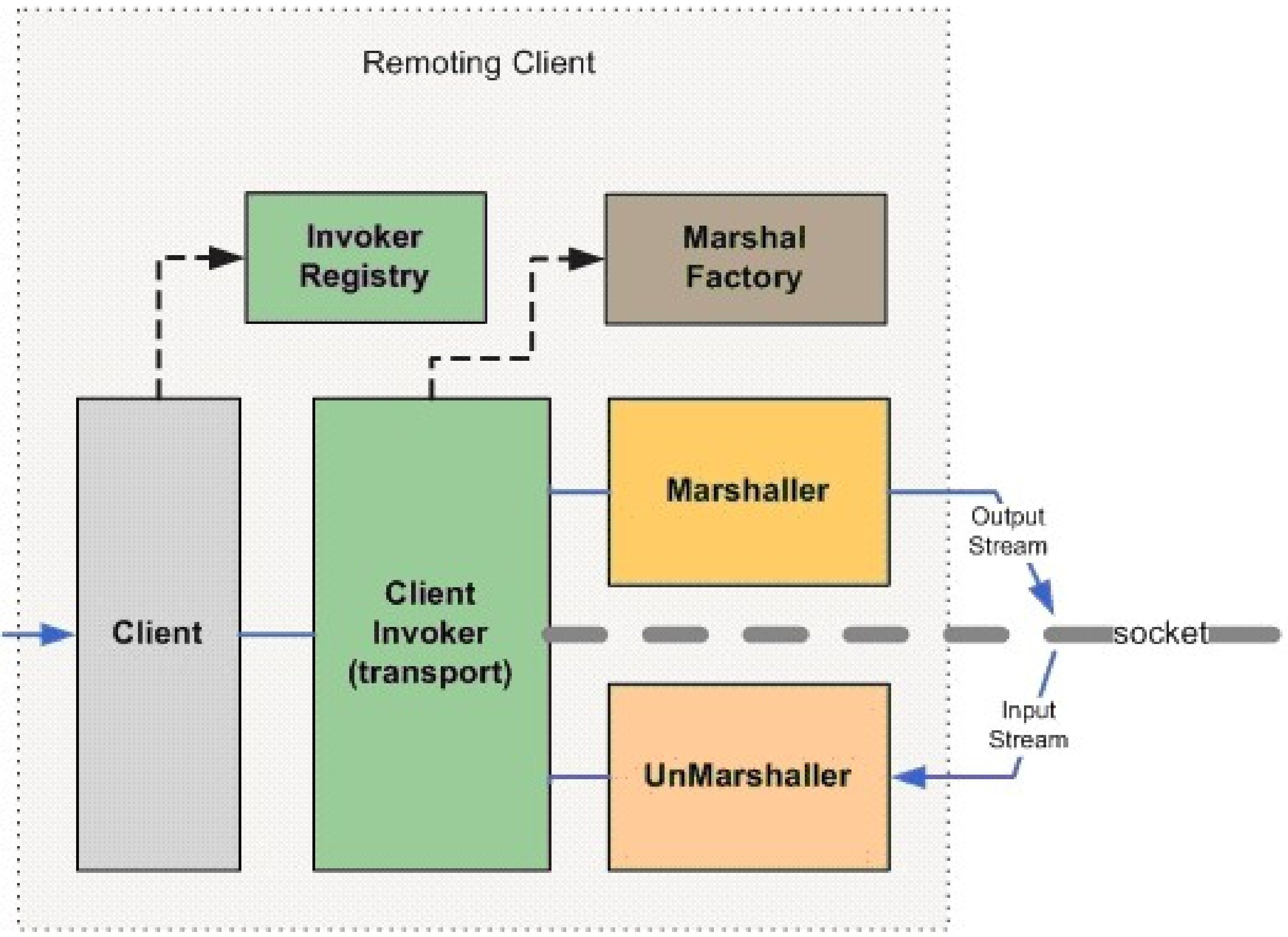
There are several layers to this framework that mirror each other on the client and server side. The outermost layer is the one which the user interacts with. On the client side, this is the Client class upon which the user will make its calls. On the server side, this is the InvocationHandler, which 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

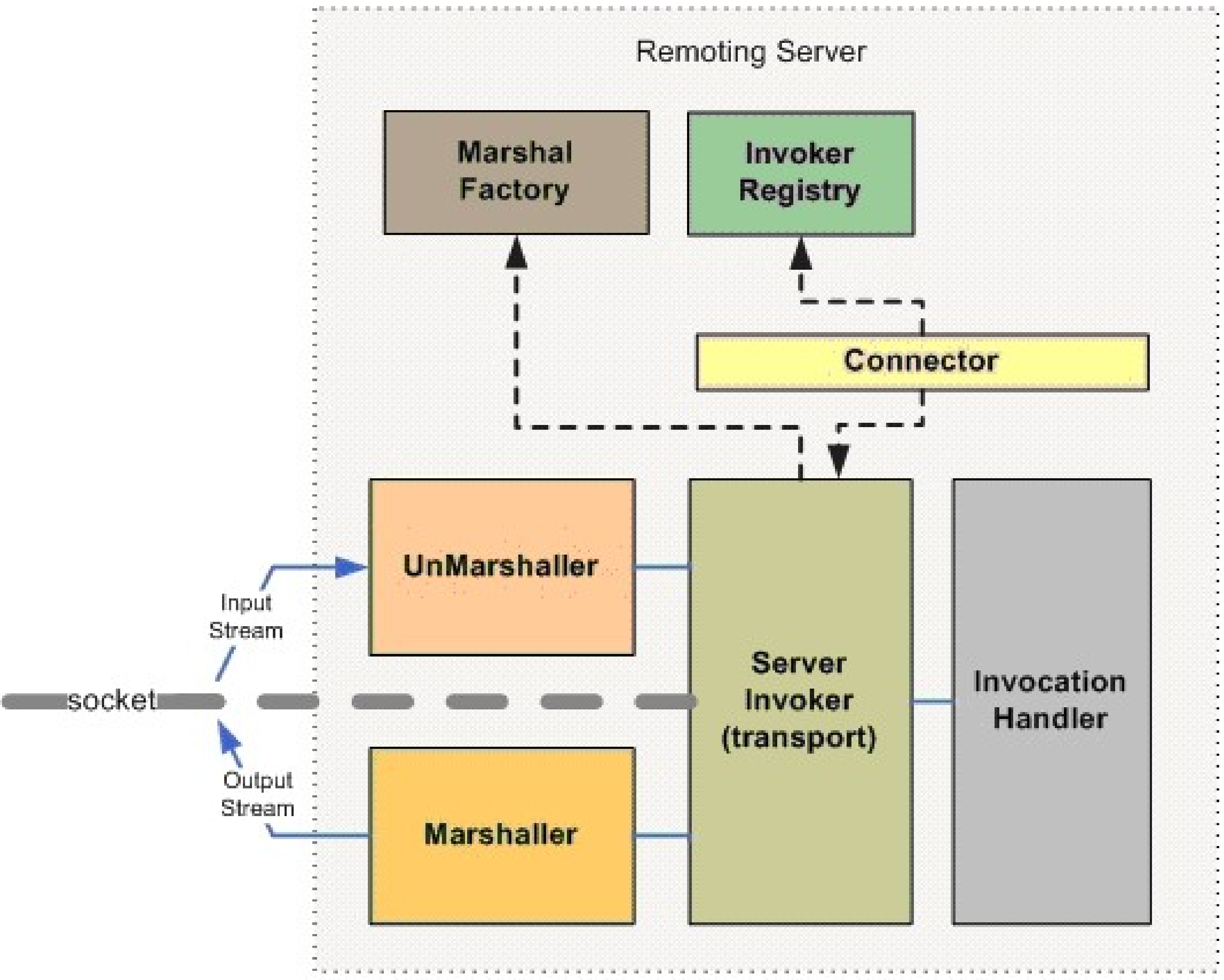
On the client side, there are a few utility class that help in figuring out which client invoker and marshal instances should be used.



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

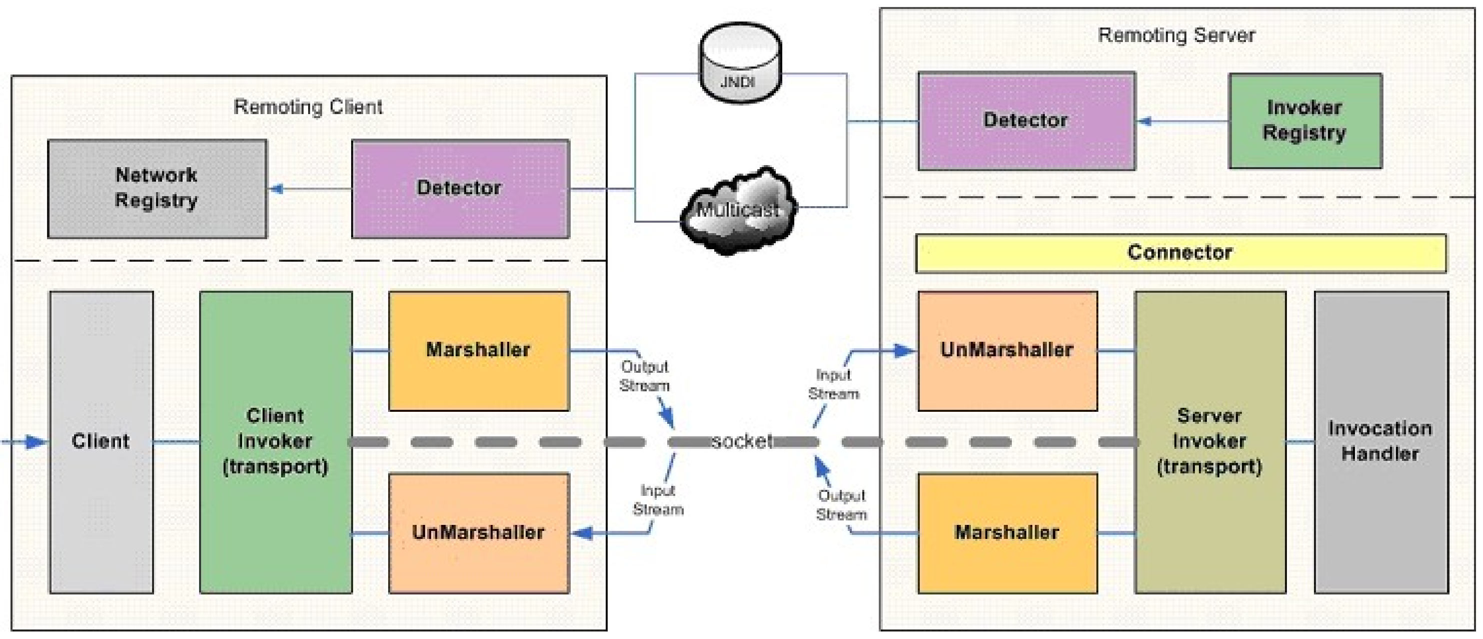
On the server side, there are also a few utility classes for determining the appropriate server invoker and marshal instances that should be used. There is also a server specific class for tying the invocation handler to the server invoker.



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 JBoss-Remoting 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 JBoss 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).

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 – the identity is what uniquely identifies a remoting server instance. Typically, there is only one identity per JVM in which a remoting server is running.

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.

4 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.

4.1. General Connector and Invoker configuration

The server invoker and invocation handlers are configured via the Connector. Only one invoker can be declared per connector (multiple InvokerLocator attributes or invoker elements within the Configuration attribute is not permitted). 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 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"
      xmbean-dd="org/jboss/remoting/transport/Connector.xml"
      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"
      xmbean-dd="org/jboss/remoting/transport/Connector.xml"
      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]]></attribute>
```

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).

4.2. 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, and response headers. 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's 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.

4.3. 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.

4.4. Transports (Invokers)

4.4.1. 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.

4.4.2. 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()`).

4.4.3. How the server bind address and port is ultimately determined

If the serverBindAddress property is set, it will be used for binding. If the serverBindAddress is not set, but the clientConnectAddress property is set, the server invoker will bind to local host address. If neither the serverBindAddress nor the clientConnectAddress properties are set, then will try to bind to the host specified within the InvokerLocator. If the host value of the InvokerLocator is also not set, will bind to local host.

If there is a system property called 'remoting.bind_by_host' and if is false, will bind by IP address instead of host. Otherwise will use host name. This only applies when configured address is 0.0.0.0. To facilitate setting this property, the following static variable is defined in `InvokerLocator`:

```
            public static final String BIND_BY_HOST = "remoting.bind_by_host";
```

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.

4.4.4. 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.

socketTimeout - The socket timeout value passed to the Socket.setSoTimeout() method. The default on the server side is 60000 (one minute). If the socketTimeout 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.

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.

socketTimeout - 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 threads. The default is 10.

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.

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.

4.4.5. SSL Socket Invoker

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

serverSocketFactory - Sets the server socket factory. If want ssl support use a server socket factory that supports ssl. The only requirement is that the server socket factory value must be an ObjectName, meaning the server socket factory implementation must be an MBean and also MUST implement the `org.jboss.remoting.security.ServerSocketFactoryMBean` interface.

4.4.6. 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.

4.4.7. 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.htm. 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 and since 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 gurantee 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.

Exception Handling

When using remoting on the client side to connect to a remoting server (or any web server for that matter) via the http transport, if the server returns a response code greater than 400, the remoting client will read the error stream and return that as the response. Thus, the response returned will be of type java.lang.Exception. NOTE: this does NOT mean that the call to the Client's invoke() method will throw the exception, but instead will return the actual Exception object instance as a normally returned response. Therefore, is important that if want to check if response is an error instead of normal response, will need to look at the response code put in the metadata Map passed to the invoke() method on the Client instance. See the HTTP Handler section above for more details.

4.4.8. HTTPS Invoker

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

serverSocketFactory - Sets the server socket factory. If want ssl support, use a server socket factory that supports ssl. The only requirement is that the server socket factory value must be an ObjectName, meaning the server socket factory implementation must be an MBean and also MUST implement the `org.jboss.remoting.security.ServerSocketFactoryMBean` interface.

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

4.4.9. 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.

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 will take precedence over setting the metadata Map.

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=176 [http://www.jboss.org/index.html?module=bb&op=viewforum&f=176]).

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. To override this behavior to allow this by changing the HostnameVerifier implementation, can use the 'org.jboss.security.ignoreHttpsHost' property'. This property can either be set using a system property or within the metadata Map passed to the Client's invoke() method (which will override both the default value and the setting from the system property).

4.4.10. 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"
                   xmbean-dd="org/jboss/remoting/transport/Connector.xml"
                   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 because if using automatic discovery, 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 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">

      <!--The the JBossRemoting server invoker servlet web.xml descriptor-->
      <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>
            </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>
```

This file can be changed to meet any web requirements you might have, such as adding security 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. Also note that there is a parameter, invokerName, that has the value of the object name of the servlet server invoker. This is what the ServerInvokerServlet uses to look up the server invoker which it will pass the requests on to.

Due to the way the servlet invoker is currently configured and deployed, it must run within the JBoss application server and is not portable to other web servers.

Exception handling

If the ServletServerInvoker catches any exception thrown from the invocation handler invoke() call, it will send an error to the client with a status of 500 and include the original exception message as its error message. From the client side, the client invoker will actually throw a CannotConnectException, which will have root exception as its cause. The cause should be an IOException with the server's message. For example, the stack trace from the exception thrown within the test case org.jboss.remoting.transport.servlet.test.ServletInvokerTestClient is:

```
                org.jboss.remoting.CannotConnectException: Can not connect http client invoker.
                at org.jboss.remoting.transport.http.HTTPClientInvoker.useHttpURLConnection(HTTPClientInvoker.java:
                at org.jboss.remoting.transport.http.HTTPClientInvoker.transport(HTTPClientInvoker.java:68)
                at org.jboss.remoting.RemoteClientInvoker.invoke(RemoteClientInvoker.java:113)
                at org.jboss.remoting.Client.invoke(Client.java:221)
                at org.jboss.remoting.Client.invoke(Client.java:184)
                at
                org.jboss.remoting.transport.servlet.test.ServletInvokerTestClient.testInvocation(ServletInvokerTes
                at
                org.jboss.remoting.transport.servlet.test.ServletInvokerTestClient.main(ServletInvokerTestClient.ja
                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:324)
                at com.intellij.rt.execution.application.AppMain.main(AppMain.java:78)
                Caused by: java.io.IOException: Server returned HTTP response code: 500 for URL:
                http://localhost:8080/servlet-invoker/ServerInvokerServlet
                at sun.net.www.protocol.http.HttpURLConnection.getInputStream(HttpURLConnection.java:791)
                at org.jboss.remoting.transport.http.HTTPClientInvoker.useHttpURLConnection(HTTPClientInvoker.java:
                ... 11 more
```

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.

4.4.11. 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/inde
x.html?ctrl:id=page.default.info&
http://labs.jboss.com/portal/index.html?ctrl:id=page.default.info&project=jbossremotingproject=jbossremoting]
```

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.

4.4.11.1. 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 `Multiplex-`

`ServerInvokers` 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()
```

4.4.11.2. 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 succinctly, 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.

4.4.11.2.1. 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.

4.4.11.3. 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 *socketTimeout* 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.

4.4.11.4. 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 invokers.

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

```
String parameters = "multiplexBindHost=localhost&multiplexBindPort=7070&clientMultiplexId=demoId1";
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();
```

4.4.11.5. Configuration properties

There are four categories of configuration properties supported by the multiplex invoker.

1. The following properties can be used to configure both master and virtual `MultiplexorServerInvokers`. They can be set at any time, but will not take effect until the invoker is stopped and restarted. A subset of the parameters applicable to the socket invoker is currently implemented.

   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 `MultiplexServerInvoker` and then communicated to a corresponding `MultiplexClientInvoker` via parameters in the Locator URI. These configurations cannot be changed during runtime, so can only be set up upon initial configuration of the multiplex invoker on the server side. A subset of the parameters applicable to the socket invoker is currently implemented.

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

3. 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.

4. 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.

4.5. 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).

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.

4.6. Callbacks

4.6.1. Callback overview

Although this section covers callback configuration, will need to first cover a little general information about callbacks within remoting. There are two models for callbacks, push and pull. In the push model, the client will register a callback server via an

InvokerLocator 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. The target server then has to manage these callback messages on the server 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. This 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:

```
Callback has been lost because not enough free memory to hold object.
```

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:

```
This is an invalid callback. The server ran out of memory, so callbacks were lost.
```

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:

```
Acts as a persistent list which writes Serializable objects to disk and will retrieve them in same order
in which they were added (FIFO). Each file will be named according to the current time (using Sys-
tem.currentTimeMillis() with the file suffix specified (see below). When the object is read and returned
by calling the getNext() method, the file on disk for that object will be deleted. If for some reason the
store VM crashes, the objects will still be available upon next startup. The attributes to make sure to
configure are:

file path - this determines which directory to write the objects. The default value is the property value
of 'jboss.server.data.dir' and if this is not set, then will be 'data'. For example, might be /
jboss/server/default/data.

file suffix - the file suffix to use for the file written for each object stored.

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

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.

4.6.2. Callback Configuration

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

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.

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.

CallbackStore configuration

The CallbackStore can be configured via the invoker configuration as well.

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'.

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"
                xmbean-dd="org/jboss/remoting/transport/Connector.xml"
                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.CallbackStore</attribute>
                      <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.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"
         xmbean-dd="org/jboss/remoting/transport/Connector.xml"
         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"
         xmbean-dd="org/jboss/remoting/transport/Connector.xml"
         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.CallbackStore</attribute>
               <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>
```

4.6.3. 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.

4.7. Programmatic configuration

It is possible to configure all this programmatically, if running outside the JBoss Application server for example, but is a little more tedious. Since the remoting components are all bound together by the `org.jboss.remoting.transport.Connector` class, will need to call its `setConfiguration(org.w3c.dom.Element xml)` method with same xml as in the mbean service configuration, before calling its `start()` method.

The xml passed to the Connector should have `<config>` element as the root element and continue from there with `<invoker>` sub-element and so on. An example of this can be found in org.jboss.test.remoting.configuration.SocketClientConfigurationTestCase.

4.8. SSL Support and configuration

There are two transports that now support SSL: sslsocket and https. This section will cover configuration, implementation, some samples, and some troubleshooting tips.

Both the sslsocket and https transports are extensions of their non-ssl counterparts, socket and http transports, so the same basic configurations will apply. Therefore, only the ssl specific configurations will be covered here.

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.

sslsocket

The sslsocket transport can be defined in one of two ways if using a service xml to declare the remoting server. The first is to use the sslsocket protocol keyword in the locator url of the InvokerLocator attribute value of the Connector service mbean. For example:

```
          <mbean code="org.jboss.remoting.transport.Connector"
                xmbean-dd="org/jboss/remoting/transport/Connector.xml"
                name="jboss.remoting:service=Connector,transport=SSLSocket"
                display-name="SSL Socket transport Connector">

             <attribute name="InvokerLocator">sslsocket://myhost:8084</attribute>
```

The other way is to not use the InvokerLocator attribute, but instead a more verbose Configuration attribute, which declares the invoker transport type as a sub-element. For example:

```
          <mbean code="org.jboss.remoting.transport.Connector"
                xmbean-dd="org/jboss/remoting/transport/Connector.xml"
                name="jboss.remoting:service=Connector,transport=SSLSocket"
                display-name="SSL Socket transport Connector">

             <attribute name="Configuration">
                <config>
                   <invoker transport="sslsocket">
                      <attribute name="numAcceptThreads">1</attribute>
                      <attribute name="maxPoolSize">303</attribute>
```

If defining the remoting server programmatically, not from a server xml file, all that is needed is to create the InvokerLocator with sslsocket as the protocol. Of course the other Connector operations will be needed as well. A simple example would be:

```
             Connector connector = new Connector();
             InvokerLocator locator = new InvokerLocator(“sslsocket://myhost:8084”);
             connector.setInvokerLocator(locator.getLocatorURI());
             connector.create();
             connector.addInvocationHandler(getSubsystem(), getServerInvocationHandler());
             connector.start();
```

SSL Server Socket Selection

All of the forms of configuration mentioned previously will use the default configuration for selecting which SSL server socket factory to use. Technically, this is done by calling on the `javax.net.ssl.SSLServerSocketFactory` ’s `getDefault()` method. This will require that both the `javax.net.ssl.keyStore` and the `javax.net.ssl.keyStorePassword` system properties are set. This can be done by either calling the `System.setProperty()` or via JVM arguments. This also means that all the SSL configurations default to those of the JVM vendor.

There are two ways in which to customize the SSL configuration to be used by the SSLSocketServerInvoker. The first is to explicitly set the server socket factory that the invoker should use to create its server sockets. This can be done programmatically via the following method (which is also exposed as a JMX operation):

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

The server socket factory to be used by the invoker can also 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 the JMX ObjectName of an MBean that implements the `org.jboss.remoting.security.ServerSocketFactoryMBean` interface. An example of this configuration would be:

```
        <mbean code="org.jboss.remoting.transport.Connector"
              xmbean-dd="org/jboss/remoting/transport/Connector.xml"
              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 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.

The order of selecting which server socket factory is:

1.If a `javax.net.ServerSocketFactory` has been specified via the `setServerSocketFactory()` method, use this.

2.If the `serverSocketFactory` property has been set, then take the String value, create an ObjectName from it, look up that MBean from the MBeanServer that the invoker has been registered with (by way of the Connector) and create a proxy to that MBean of type `org.jboss.remoting.security.ServerSocketFactoryMBean`. Then use this proxy. 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.

3.If the server socket factory has not been set explicitly via the `serverSocketFactory` property, then use the `javax.net.ssl.SSLServerSocketFactory` 's `getDefault()` method.

Note: If want to set the server socket factory via the invoker's `setServerSocketFactory()` method, it requires a bit of work, so would opt for using a configuration setting when possible. The following snippet of code shows how it can be done programmatically:

```
           Connector connector = new Connector();
           InvokerLocator locator = new InvokerLocator(“sslsocket://myhost:8084”);
           connector.setInvokerLocator(locator.getLocatorURI());
           connector.create();
           // create your server socket factory
           ServerSocketFactory svrSocketFactory = createServerSocketFactory();
           // notice that the invoker has to be explicitly cast to the
           // SSLSocketServerInvoker type
           SSLSocketServerInvoker socketSvrInvoker = (SSLSocketServerInvoker) connector.getServerInvoker();
           socketSvrInvoker.setServerSocketFactory(svrSocketFactory);

           connector.addInvocationHandler(getSubsystem(), getServerInvocationHandler());
           connector.start();
```

The ordering is also important in that the call to the Connector’s create() method will create the invoker so that it is available via the getServerInvoker() method. However, the server socket factory MUST be set before the Connector’s start() method is called, because this will cause the invoker’s start() method to be called, which will create the server socket to listen on (and is too late to swap out the server socket factory being used).

https

The https transport is a bit different from the sslsocket in configuration since the implementation is based off the Tomcat connectors. The first major difference is the transport protocol keyword to identify it, which is 'https'.

Next is defining the SSL implementation and server socket factory to be used. The SSL implementation to be used can be set via the 'SSLImplementation' attribute and should always have a value of `org.jboss.remoting.transport.coyote.ssl.RemotingSSLImplementation`. The server socket factory to be used should be set via the 'serverSocketFactory' attribute and should always be the javax.management.ObjectName value for an implementation of the `org.jboss.remoting.security.ServerSocketFactoryMBean` interface, which should already be registered with the MBeanServer and running (more details on this in a minute).

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

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

   <attribute name="Configuration">
      <config>
         <invoker transport="https">
            <!-- The following is for setting the server socket factory.  If want ssl support -->
            <!-- use a server socket factory that supports ssl.  The only requirement is that -->
            <!-- the server socket factory value must be an ObjectName, meaning the -->
            <!-- server socket factory implementation must be a MBean and also -->
            <!-- MUST implement the org.jboss.remoting.security.ServerSocketFactoryMBean interface. -->
            <attribute name="serverSocketFactory">jboss.remoting:service=ServerSocketFactory,type=SSL</attribut
            <attribute name="SSLImplementation">org.jboss.remoting.transport.coyote.ssl.RemotingSSLImplementati
            <attribute name="serverBindAddress">${jboss.bind.address}</attribute>
            <attribute name="serverBindPort">6669</attribute>
         </invoker>
         <handlers>
            <handler subsystem="mock">org.jboss.test.remoting.transport.mock.MockServerInvocationHandler</handl
         </handlers>
      </config>
   </attribute>
   <!-- This depends is included because need to make sure this mbean is running before configure invoker. -->
   <depends>jboss.remoting:service=ServerSocketFactory,type=SSL</depends>
</mbean>
```

Notice that the 'serverSocketFactory' attribute has a value of 'jboss.remoting:service=ServerSocketFactory,type=SSL', which is not defined in the configuration snippet. More on how to define this in the next section.

One of the major changes related to using the Tomcat connectors with regards to SSL is that everything within Tomcat is defined via properties configuration and there is no external API for making changes during runtime. This means that there is no way to set server socket factory implementation programatically, other than via configuration (so cannot call setServerSocketFactory() method on the server invoker as could with the ssl socket invoker). This also means that if not running within the JBoss Application Server container, but running stand alone, will need to setup an MBeanServer and register the server socket factory with it programatically. For an example of how to do this programatically, can refer to org.jboss.test.remoting.transport.http.ssl.basic.HTTPSInvokerTestServer.

The configuration for SSL support only works when using the java based http processor and not with the APR based transport. See section 4.7 for more information on using the APR based transport.

SSLSocketBuilder

Although any server socket factory can be set on the SSL socket server invoker and the https server invoker, 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 SSLSocketBuilder is where the ssl server socket (and ssl sockets for clients) originate and is where all the properties for the ssl server socket are configured (more on this further below). The SSLSocketBuilder is also a service MBean, so can be configured and started from within a service xml.

This is an example of both the configurations as might be found within a service xml:

```
<!-- 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 format), -->
   <!-- JCEKS (Java Cryptography Extension keystore - More secure version of JKS), and -->
   <!-- PKCS12 (Public-Key Cryptography Standards #12 keystore - RSA's Personal Information Exchange Sy
   <!-- These are not case sensitive. -->
   <attribute name="KeyStoreType">JKS</attribute>
</mbean>

<!-- The server socket factory mbean to be used as attribute to socket invoker -->
```

```
            <!-- See serverSocketFactory attribute above for where it is used -->
            <!-- 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>
```

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.

If want to be able 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 be done programmatically by calling the `setUseSSLServerSocketFactory()` and passing false as the parameter value.

The configuration properties are as follows:

SecureSocketProtocol - The protocol for the SSLContext. Some acceptable values are TLS, SSL, and SSLv3. Defaults to TLS (DEFAULT_SECURE_SOCKET_PROTOCOL)

KeyManagementAlgorithm - The algorithm for the key manager factory. Defaults to SunX509 (DEFAULT_KEY_MANAGEMENT_ALGORITHM)

KeyStoreType - The type to be used for the key store. Defaults to JKS (DEFAULT_KEY_STORE_TYPE). 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.

KeyStorePassword - 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 it is not explicitly set.

KeyPassword - 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 key password.

Some other points of note:

- A SecureRandom is NOT configurable. When calling SSLContext's init() method, it is actually null, so will use the default implementation.
- Note that there are currently no ways to specify providers, so will use the default provider (which is determined by the JVM vendor).
- If the key password is not set, will try to use the value of the key store password.

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.

Troubleshooting Tips

Common errors when using server socket factory:

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

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 SSLServerSocketFactory.

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

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 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.

It is VERY IMPORTANT that the StreamInvocationHandler implementation close the InputStream when it finished reading, as will close the real stream that lives within the client VM.

5.1. 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).

5.2. 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.

6 Serialization

JBoss Remoting allows for the plugging in of custom serialization implementations. This is available via the `org.jboss.remoting.serialization.SerializationStreamFactory` class, which will provide the implementation as a `org.jboss.remoting.serialization.SerializationManager` . The SerializationManager can then be called on to get implementations for `java.io.ObjectInputStream` and `java.io.ObjectOutputStream` . This SerializationManager is used by most of the standard remoting marshallers and unmarshallers. There are currently two implementations of the SerializationManager; one for the standard java serialization, which is the default, and one for 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.

7 Connection Exception Listeners

Client side

It is possible to register a listener with the remoting client to receive callbacks when a connection failure to a remoting server is detected, even when the client is idle.

The only requirement is to implement the `org.jboss.remoting.ConnectionListener` interface, which has only one method:

```
public void handleConnectionException(Throwable throwable, Client client)
```

Then call the `addConnectionListener(ConnectionListener listener)` method on the Client class and pass your listener instance. Can also call `addConnectionListener(ConnectionListener listener, int pingPeriod)` if want to specify how frequently wish to ping server.

Currently, the Client will use the `org.jboss.remoting.ConnectionValidator` class to handle the detection of connection failures. This is done by pinging the server periodically (defaults to every 2 seconds). If there is a failure during this ping, the exception and the Client will be passed to the listener.

Server side

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.

To turn on server side connection failure detection of remoting clients, will need 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:

```
public void setLeasePeriod(long leasePeriodValue)
```

method within Connector. The second criterion is that an implementation of the `org.jboss.remoting.ConnectionListener` interface is added as a connection listener to the Connector, via the method:

```
public void addConnectionListener(ConnectionListener listener)
```

Note, there is no way to set the connection listener via xml based configuration for the Connector. 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 handleConnectionException() 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.

One the client side, there is no API or configuration changes needed. 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 the 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.

For examples of how to use server side connection listeners, reference org.jboss.test.remoting.lease.LeaseTestServer and org.jboss.test.remoting.lease.LeaseTestClient.

8

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.

9

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

9.1. 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.

9.2. 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 simple page s
                                                   "<p>Should show up in browser or via invoker client</body></htm

   // 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 message from 
         // 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 a complex


      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 a text ba

      // 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 simple page

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.

9.3. 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.

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.

9.4. 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 manually via
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 NetworkRegistri
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 welcome mes
Fri Jan 13 09:36:50 EST 2006: [CLIENT]: Sending welcome message to remoting server with locator uri of: socket://12
Fri Jan 13 09:36:50 EST 2006: [CLIENT]: The newly discovered server sent this response to our welcome message: Rece
```

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 manually via
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 NetworkRegistri
Fri Jan 13 09:36:46 EST 2006: [SERVER]: Starting remoting server with locator uri of: socket://localhost:5400
Fri Jan 13 09:36:46 EST 2006: [SERVER]: Added our invocation handler; we are now ready to begin accepting messages
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 welcome m
```

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 of: Invoke
```

9.5. 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.

9.6. 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_sample.txt
```

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.

9.7. 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.

9.8. Transporters

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).

9.8.1. 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 first, 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 implementation 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, `Customer-ProcessorImpl` 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(locatorUR

      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).

9.8.2. 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(locatorURI, Order

      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 target '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.

9.8.3. 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 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, CustomerProces
   }

   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.

9.8.4. 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-server' 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 are no othe
*** Could not find doctor for patient.  This is an expected exception when there are not doctors available.
org.jboss.remoting.samples.transporter.complex.NoDoctorAvailableException: No doctor available for ailment 'financ
 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.

9.9. 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 *server-MultiplexId* 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 *client-MultiplexId* parameters, respectively.

- `Server2Client2`: A `MultiplexServerInvoker` starts according to server rule 2, after which a `MultiplexClientInvoker` 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`.

10

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` .

11

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

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.

12 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]).

13 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.

14

Release Notes

API incompatabilities between JBossRemoting 1.0.2 and 1.2.X

The following public API for JBossRemoting was changed in release 1.2.0 which will make it incompatible with previous versions:

- Removed ClientInvokerAdapter and dependent classes

- Callback related classes moved to new remoting callback package

- InvokerCallbackHandler accepts Callback type as parameter instead of InvocationRequest

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