JBossRemoting

Version 1.2.0 final

June 21, 2005

Table of Contents

What is JBossRemoting?

The purpose of JBossRemoting 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 addition 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 JBoss Application Server project, but will be the framework used for many of the JBoss projects 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.

Features

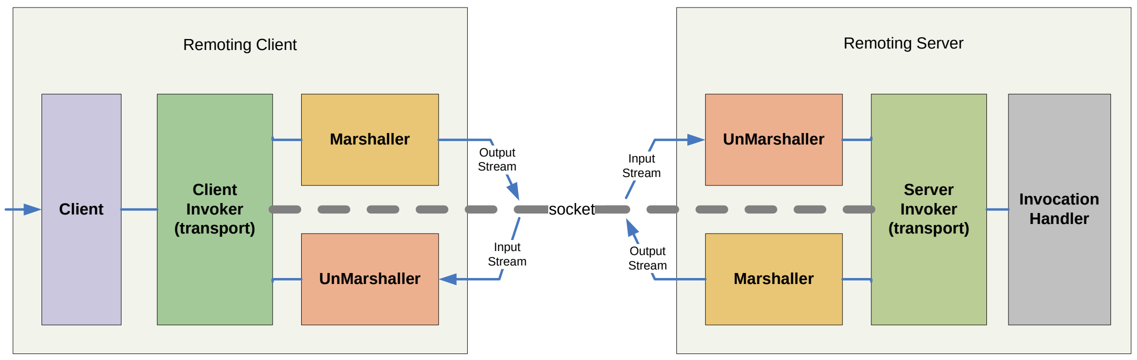
- **Server identification –** a simple String identifier which allows for remoting servers to be identified and called upon.
- **Pluggable transports –** can use different protocol transports, such as socket, rmi, http, etc., via the same remoting API.
- **Pluggable data marshallers –** can use different data marshallers and unmarshallers to convert the invocation payloads into desired data format for wire transfer.
- **Automatic discovery –** can detect remoting servers as they come on and off line.
- **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.

How to get it

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

Design

There are several layers to the JBossRemoting framework that mirror each other on the client and server side. The outer most layer is the one in which the user interacts with. On the client side, this is the Client class. On the server side, this is the InvocationHandler. 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.



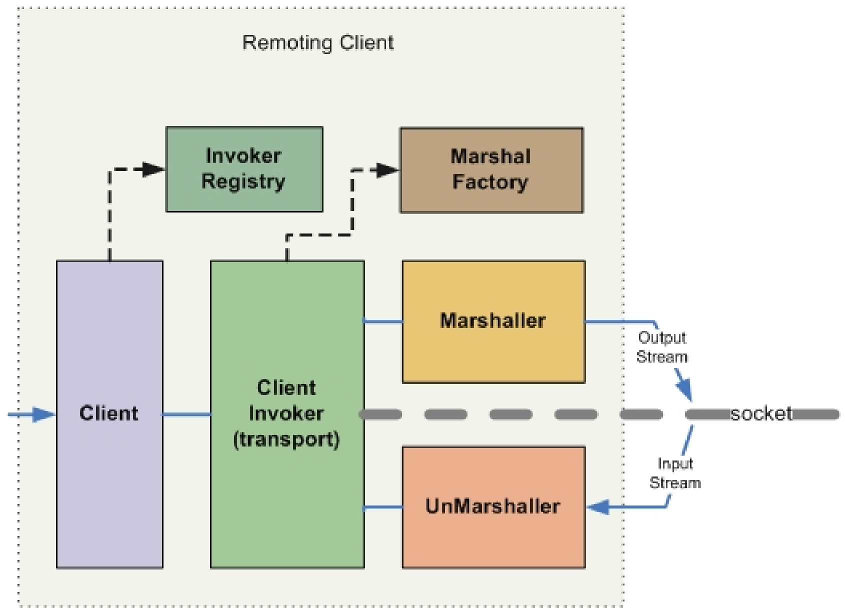
When a user calls on the Client to make an invocation, the client will pass this invocation request to the appropriate client invoker, based on transport specified by the locator url. The client invoker will then use the marshaller to convert the invocation request to the proper data format to send over the network.

On the server side, an unmarshaller will receive this data from the network and convert it back into a standard invocation request and send it onto the server invoker. The server invoker will then pass this invocation request onto the user's implementation of the invocation handler. The response from the invocation handler will pass back through the server invoker and onto the marshaller, which will then convert the invocation response 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 and will be passed back up through the client invoker and client to the original caller.

Client Side

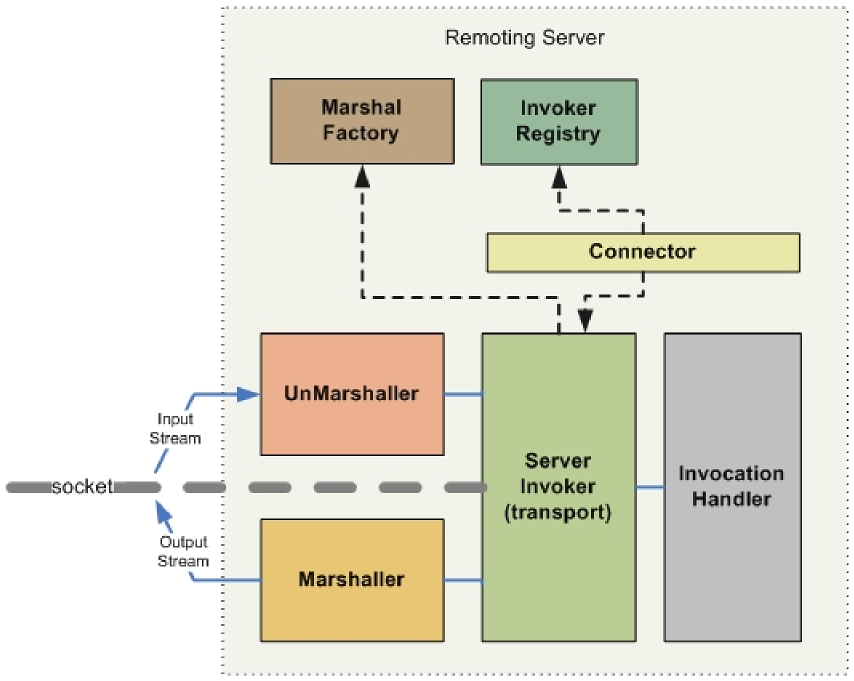
On the client side, there are a few utility class that help in figuring out which client invoker and marshall instances should be used.



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 marshall instances, but can be overridden within the locator specified.

Server Side

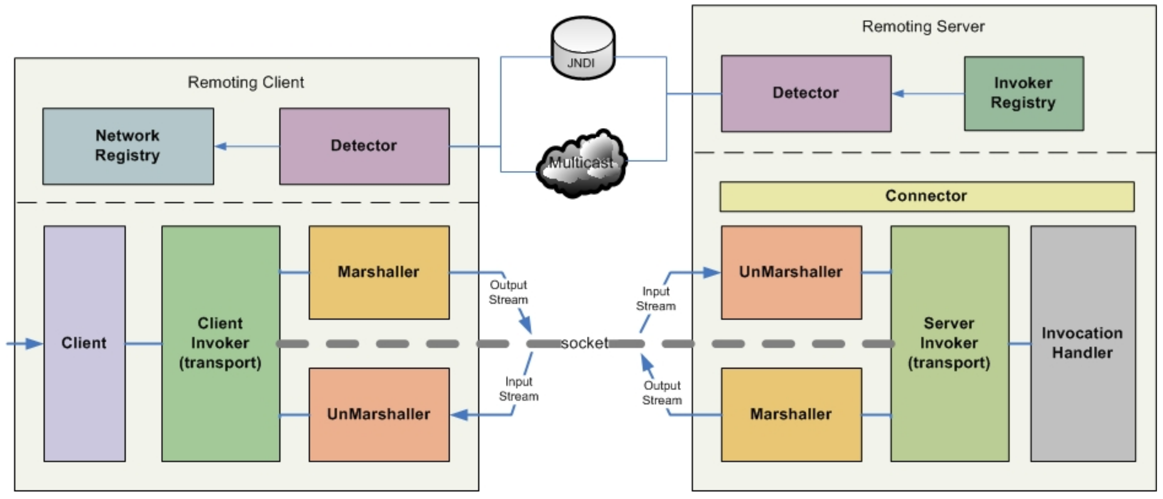
On the server side, there are also a few utility classes for determining the appropriate server invoker and marshall instances that should be used. There is also a server specific class for tying the invocation handler to the server invoker.



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.

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, but good to call in either case).

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

```
[transport]://[host]:<port>/<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 with the value of the local host name.

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 it's 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 from.

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 routed based on sub-system. For example, a particular socket based server invoker may handle invocations for both JMX and EJB. 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.

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.

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 a invocation handler is not required, it should only be done in the case of declaring a callback server. Otherwise client invocations can not be processed. The invocation handler is the only interface that is required by a 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 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 (i.e. `socket`, `rmi`, or `http`). 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 replace 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.

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). Some sample code of this (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). Some sample code of this (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 the reference to the single handler instance 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.

Discovery (Detectors)

Configuration common to all detectors:

Domains

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

An example entry of a Multicast detector in the jboss-service.xml that only accepts detections from the roxanne and sparky domains using port 5555 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>
      </attribute>
</mbean>
```

DefaultTimeDelay - amount of time, in milliseconds, which can elapse without receiving a detection event before a server will be suspected as being 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 to for the JNDI server.
Host - host to which the detector will connect to 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 any where 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.

Transports (Invokers)

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().

Note: This applies for all the server invokers except the rmi server invoker, which does not honor this configuration property (see **JBREM-127**)

serverBindPort - The port to listen for requests on. A value of 0 or less indicates that an 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 a 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.

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 setup upon initial configuration of the socket invoker on the server side. The following is a list of these and their affects.

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 the requests 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 the requests 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()`)

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 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 an used. If neither the serverBindPort or 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 use. In the case that the next available port is used because either the serverBindPort or the original InvokerLocator port value was either 0 or negatvie, the InvokerLocator will be updated to reflect the new port value.

Socket Invoker

The following configuration properties can be set at any time, but will note take affect until the socket invoker, on the server side, is stopped and restarted.

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 qualifies 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 setup upon initial configuration of the socket invoker on the server side. The following is a list of these and their affects.

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).

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

clientMaxPoolSize - the client side maximum number of threads. The default is 300.

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
```

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.

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

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 a MBean and also MUST implement the org.jboss.remoting.security.ServerSocketFactoryMBean interface.

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 (see below for more information on the isParam attribute).

HTTP Invoker

The HTTP Invoker allows for some of the properties to be passed as headers. 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 HTTPServerInvoker 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).

The following can be set on the HTTP server invoker via normal configuration properties mechanism.

maxNumThreadsHTTP - specifies the maximum number of threads to be used within the thread pool used receive incoming requests. 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 has no meaning. This property can also be retrieved or set programmatically via the MaxNumberOfHTTPThreads property.

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

The org.jboss.test.remoting.configuration.threadpool.HTTPThreadPoolConfigurationTestCase test cases demonstrates how to set a custom ThreadPool implementation via configuration.

Note: The HTTPServerInvoker is going to be very expensive as need to write out the size of the response (Content-Length). This basically means serializing the response object to byte array and getting size of the array (very expensive). See JBREM-138 for further details

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 a MBean and also MUST implement the org.jboss.remoting.security.ServerSocketFactoryMBean interface.

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 to simply add the following properties to the metadata Map passed on the Client's invoke() method:

```
http.proxyHost
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 a 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
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
http.proxy.password
```

For non-proxied invocations, the following properties need to be set:

```
http.basic.username
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 request belong. If have question about how to use these features that is not documented, please post them to the remoting forum (http://www.jboss.org/index.html?module=bb&op=viewforum&f=176).

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 onto 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 lookup the server invoker which it will pass the requests onto.

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 it's 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(HTTPClientInvoke
r.java:154)
	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(Servlet
InvokerTestClient.java:65)
	at 
org.jboss.remoting.transport.servlet.test.ServletInvokerTestClient.main(ServletInvokerTes
tClient.java:98)
	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(HTTPClientInvoke
r.java:139)
	... 11 more
```

Issues

One of the issues of using HTTP/Servlet invoker is that the invocation handlers (those that implement ServerInvocationHandler), can not provide very much detail in regards to what is returned in regards to a web context. For example, the content type used for the response is the same as that of the request. Also can not set specific response header values or send specific error status (see JBREM-140).

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.InvocationUnMarshall
er
```

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 marshall 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 marshall 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 marshall loader service running on the loader port, including any classes it depends on. This will happen automatically and not 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).

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. There is little configuration needed for this and is covered in detail in the remoting user's guide.

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 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 fore mentioned 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 System.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.

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>
```

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.

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 counter parts, `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 and be found at

http://wiki.jboss.org/wiki/Wiki.jsp?page=Remoting_example_service_xml .

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 it's 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 can not be done if just specifying the invoker configuration using the `InvokerLocator` attribute). The attribute value must be the JMX ObjectName of a 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 of 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 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 very similar to the sslsocket transport in regards to how the ssl configuration is done. Therefore will only mention the differences here. The first major difference is the transport protocol keyword to identify it, which is `https`. The two different service xml configurations 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="InvokerLocator">https://myhost:8084</attribute>
```

and

```
  <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">
```

Again, if using the https transport programmatically, the only change from using the sslsocket transport would be they locator url used. For example:

```
InvokerLocator locator = new InvokerLocator("https://myhost:8084");
```

The SSL server socket selection process is exactly the same as described in the sslsocket section above. This includes the `setServerSocketFactory()` method and the `serverSocketFactory` configuration attribute. The only difference would be if setting the server socket factory programmatically, would need to cast to the `HTTPSServerInvoker`. For example, the code would look like:

```
HTTPSServerInvoker httpsSvrInvoker = (HTTPSServerInvoker)
connector.getServerInvoker();
```

SSLSocketBuilder

Although any socket server factory can be set on the `SSLSocketServerInvoker` and the `HTTPSSocketInvoker`, 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 Syntax Standard). -->
      <!-- 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 testings

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:

1. 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.

1. 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.

1. 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.

1. 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.

Sending streams

Support for sending InputStreams using remoting has been added. 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 for better memory control by the user (verses the framework trying to copy a 3 Gig file into memory and blowing up).

Use of this new feature is simple. From the client side, there is new 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 stream passed 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.

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 socket make the invocation an a new connection back to the client (see issues below).

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 request 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, based of the JMS UIL2 invocation layer, is planned for remoting 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. However, I won't be able to get to this transport implementation for a while, so if you are interested in getting it done sooner, please e-mail me (tom dat jboss).

Connection Exception Listeners

Tt is possible to register a listener to receive callbacks when a connection failure 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 handlerConnectionException(Throwable throwable, Client
client)
```

Then call the addConnectionListener() method on the Client class and pass your listener instance.

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.

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 five sets of sample code, each with their own package; simple, oneway, detection, stream, and callback. Within each of these packages, there will be a server and a client class that will need to be executed. If running samples from command line and have ant installed, can use the following ant targets:

Simple invocation - run-simple-client & run-simple-server
Oneway invocation – run-oneway-client & run-oneway-server
Discovery and invocation – run-detector-client & run-detector-server
Callbacks (push & pull) – run-callback-client & run-callback-server
Sending streams – run-stream-client & run-stream-server

So if wanted to run the simple sample would open a command prompt and type:

```
ant run-simple-server
```

and then:

```
ant run-simple-client
```

Each target will compile the sample classes if they have not been already. Remember to always run the server first, then the client.

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 call on that server. This expectation is 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 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

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., which is aprx 5MB (due to thirdparty libs).

See 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 of 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 not 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 to include functional and performance tests for all the different transports.
`clean` - removes all the build artifacts and directories.
`most` - calls clean then jars target.
`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.

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 (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, Unified Invokers forum (http://www.jboss.org/index.html?module=bb&op=viewforum&f=176).

1. The RMI server invoker does not use the serverBindAddress property for its configuration, nor does it allow for custom socket factories (JBREM-127)

2. The HTTP server invoker does not wait for in flight request to finish processing before executing full stop when the stop() method is called. (JBREM-115)

3. If the bind address is 0.0.0.0, the InvokerLocator will call InetAddress.getLocalHost().getHostName(). Need to be configurable for either ip or host name. (JBREM-120)

4. The payload is currently deserialized on the server side (ServerInvoker). Need to only deserialize the remoting specific payload in the ServerInvoker and pass along the still marshaled target payload to the sub-system handler (UnifiedInvoker), since should have the same classloader context as the end target. (JBREM-42).

5. Dynamic classloading is not fully implemented. Is possible to dynamically load classes on the client from the server if configured to do so, but can not currently load classes on the server from the client. (JBREM-47).

6. The HTTP client and server invoker only support POST requests. This is a somewhat low priority until can find better use cases for this feature (JBREM-33).

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):

If you have an questions, comments, bugs, fixes, contributions, or flames, please post them to the JBoss Remoting, Unified Invokers forum (http://www.jboss.org/index.html?module=bb&op=viewforum&f=176). You can also find more information about JBoss Remoting on our wiki (http://www.jboss.org/wiki/Wiki.jsp?page=Remoting).

Thanks for checking it out.

-Tom

Tom Elrod
JBossRemoting Lead
JBoss, Inc.
tom@jboss.org

Release Notes

API incompatabilities between JBossRemoting 1.0.2 and 1.2.0

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

- Removed ClientInvokerAdapter and dependant classes
- Callback related classes moved to new remoting callback package
- InvokerCallbackHandler accepts Callback type as parameter instead of InvocationRequest

Release Notes - JBoss Remoting - Version 1.2.0 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 MBeanServer
* [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