# JBoss Remoting

## Version 1.0.2 final

March 24, 2005

# Table of Contents

## What is JBoss Remoting?

The purpose of JBoss Remoting is to provide a single API for most network based invocations and related service that uses pluggable transports and data marshallers. The JBoss Remoting 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, to fit these different needs.

JBoss Remoting is currently a sub-module of the JBoss Application Server and will likely be the framework used for many of the other projects when making remote calls. JBoss Remoting 1.0.1 final will be included in the JBoss AS 4.0.2 distribution and can be run as a service within the container as well. Service configurations are included in the configuration section below.

## Features

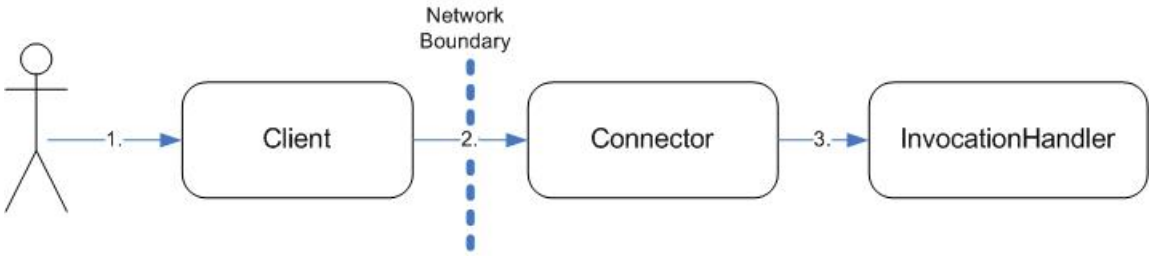
- **Server identification** – a simple String identifier which allows for remoting servers to be identified and called upon.
- **Pluggable transports** – can use different protocol transports, such as socket, rmi, http, etc., via the same remoting API.
- **Pluggable data marshallers** – can use different 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.

## How to get it

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

## Design

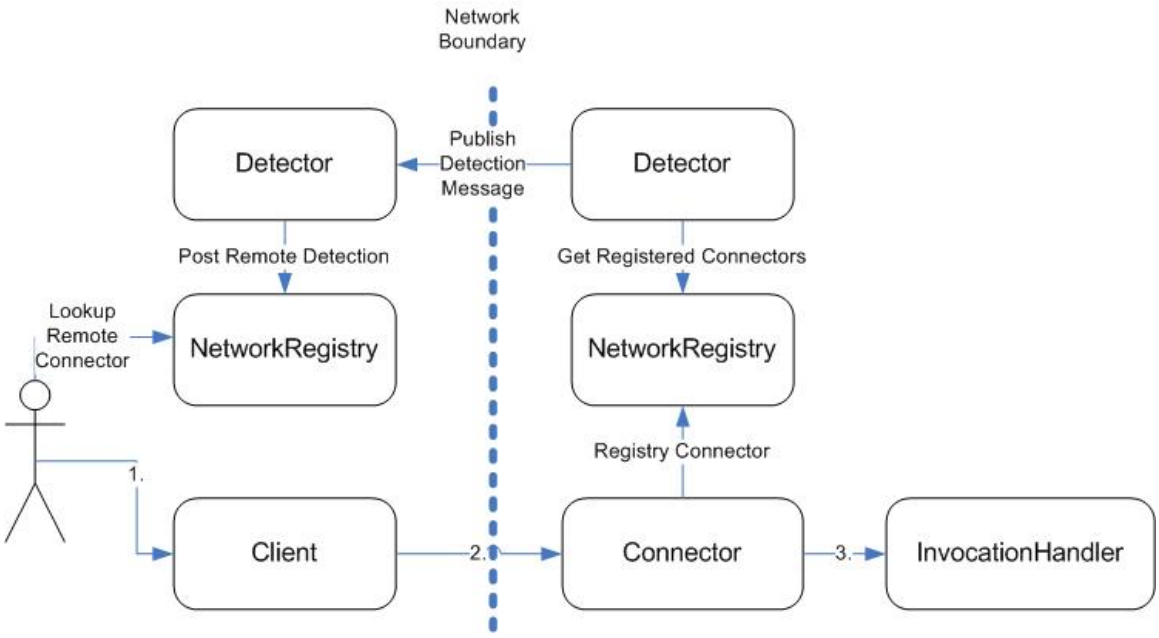
From the highest level, there are three components involved when making a remote invocation using JBoss Remoting; a client, a connector, and an invocation handler.




The user constructs a Client, providing the locator for which remote server to make the remote invocations on. The user then calls on the Client to make the invocation, passing the invocation payload. The Client will then make the network call to the remote server, which is the Connector. The connector will then call on the InvocationHandler to process the invocation. This handler is the user's implementation of the InvocationHandler interface.

The marshalling of the data, network protocol negotiation, and other related tasks are handled by the remoting framework. The effect of this is whatever payload object is passed from the user, noted by number 1 in diagram, is exactly what is passed to the InvocationHandler, noted by number 3 in diagram and all that was required by the user on the client was a locator, which can be expressed as a simple String.

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

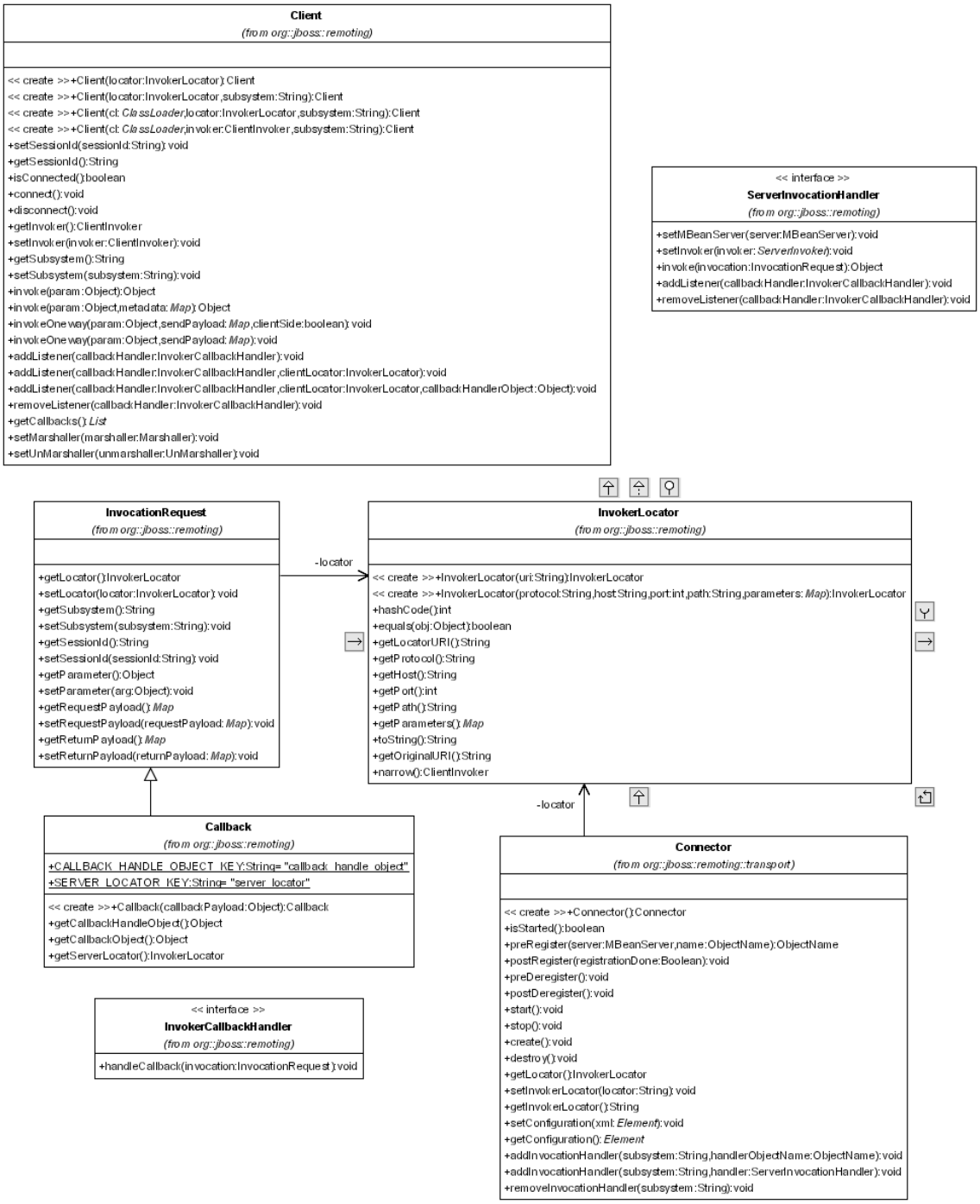



When the Connector is created, it will register itself with the local NetworkRegistry. The Detector on the server will publish a detection message containing the locator for all the Connectors registered with the NetworkRegistry.

The Detector on the client side will receive this detection message and post the locator information for the server Connectors to the NetworkRegistry. The user can then query the NetworkRegistry to determine all the Connectors that are available on the network. Based on the query result, the user can then determine which locator to use when creating the Client to be used for making invocations.

# Components

This section covers a few of the main components exposed within the Remoting API with a brief overview. Will start with a class diagram for those classes related to making invocations and callbacks.



**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).

**InvokerLocator** – is a class, which can be described as a string URI, for describing a particular JBoss 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, JBoss Remoting can make a client connection to the remote JBoss server. The format of the string URI is the same as a type URI:

```
[transport]://[ipaddress]:<port>/<parameter=value>&<parameter=value>
```

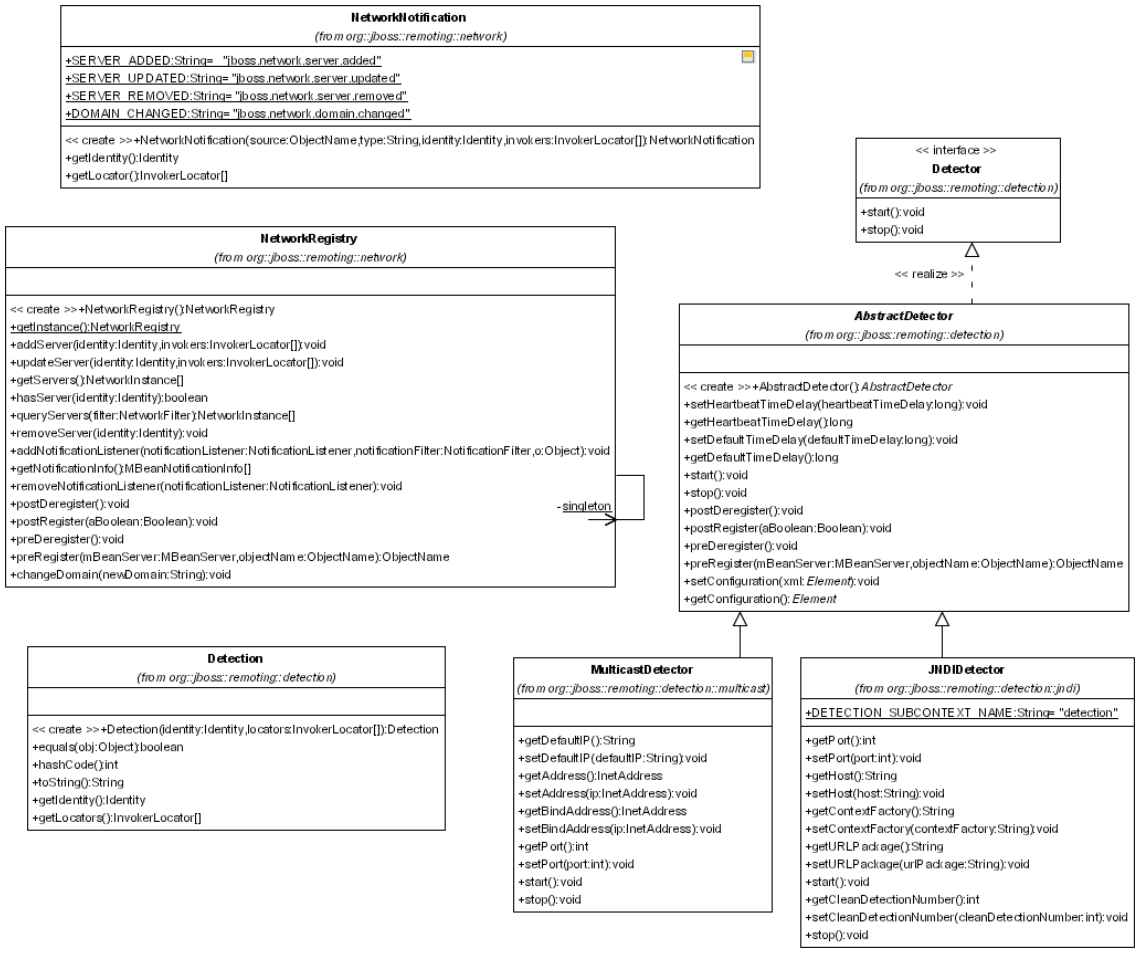
**Connector -** is an MBean that loads a particular `ServerInvoker` implementation for a given transport subsystem and one or more `ServerInvocationHandler` implementations that handle Subsystem invocations on the remote server JVM.

**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.

**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).

**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.

Next is the class diagram for classes related to automatic discovery.




**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.

**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.

**Detection** – is the detection message fired by the Detectors.

**MulticastDetector** – is the detector implementation that broadcasts its Detection message to other detectors using multicast.

**JNDIDetector** – is the detector implementation that registers its Detection message to other detectors in a specified JNDI server.

Another component that is not represented as a class, but is important to understand is the sub-system.

**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 may handle invocations for multiple invocation handlers, which need to routed based on sub-system. For example, a particular socket based Connector 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.

# 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*

Only one invoker can be declared per connector (multiple InvokerLocator attributes or invoker elements within the Configuration attribute is not permitted). At least one handler must also be specified as well, which 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 invoker, or transport, configuration via a service xml file. The first is to specify just the InvokerLocator attribute as a sub-element of the Connector MBean. All the client side configurations can be made part of the locator uri in this approach. For example, a possible configuration for a Connector using a socket invoker that has the client's max pool size of 30 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/? &clientMaxPoolSize=30]]>
    </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 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.samples.simple.SimpleServer):

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

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

To pass the handler by ObjectName, call Connector::addInvocationHandler(String subsystem, ObjectName handlerObjectName). Some sample code of this (from org.jboss.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.

## *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)*

### 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.

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

**serverBindAddress** - The address on which the server binds its listening socket. The default is an empty value which indicates the server should be bound on all interfaces.

**serverBindPort** - The port used for the server socket. A value of 0 indicates that an anonymous port should be chosen.

**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).

**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
```

**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()`)

If no client connector port or server bind port specified, will use the `PortUtil.findFreePort()` to find an available port.

If client (or server if client not present) bind address is set to 0.0.0.0, will use `InetAddress.getLocalHost().getHostAddress()` to get the host to use for the locator uri to be provided to client via discovery.

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

### 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 does not have properties in the same sense as the other invokers (this is still a todo). However, metadata will 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).

As of 1.0.1 beta release, the HTTP Invoker only supports POST requests on the server (to be fixed for 1.0.1 final release).

For example of how to use the HTTP Invoker (both client and server side), see the test classes under remoting/tests/src/org/jboss/remoting/transport/http. They give examples of how to make different calls (object, xml/soap, and html) and what headers will need to be set and how. Full documentation will be coming soon on this.

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).

## *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.

## 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 four sets of sample code, each with their own package; simple, oneway, detection, 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-clint & run-callback-server

So if wanted to run th 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.

## 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. HTTP Invoker is not complete yet. Only POST requests are supported on the server side (GET is not yet supported).
2. The HTTP Invoker has been stress tested and performance is slow, especially with oneway invocations. Under extremely high loads using oneway invocations with the HTTP Invoker will cause clients to experience invocation exceptions due to not being able to handle more requests.

## Future plans

Actually going to start with a little history here. JBoss Remoting was originally written by Jeff Haynie (jhaynie@vocalocity.net) and Tom Elrod (tom@jboss.org) and still exists in its older form in the JBoss 3.2 branch (has been backported to the 4.0 branch, but was not until after the 4.0.0 and 4.0.1 releases). This release is based off of jboss-head branch (which is actually HEAD) in CVS. The basics from the older version still remains in the current version, but is being refactored for this next release (and official first stand alone release). That being said, here is what is planned in the future (can see full road map at http://jira.jboss.com/jira/browse/JBREM?report=com.atlassian.jira.plugin.system.project:roadmap-panel):

- Add specific method for streaming large binary files.
- Add HTTP/HTTPS proxy and GET request support.
- Add Servlet Invoker (counter part to the HTTP Invoker)
- Add support for custom socket factories
- Add high availability to remoting
- Distributed garbage collection
- Client transport idle connection timeout
- Smart proxies
- Connection failure callback
- Dynamic classloading (partially implemented)
- Support for redeploy on server and synch on client
- Add UIL2 type transport
- Add JGroups transport
- Add SMTP transport
- Add NIO transport

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
JBoss Core Developer
JBoss, Inc.
tom@jboss.org

## Release Notes

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