JBoss Remoting

Version 1.0.1 beta

January 12, 2005

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.

Features

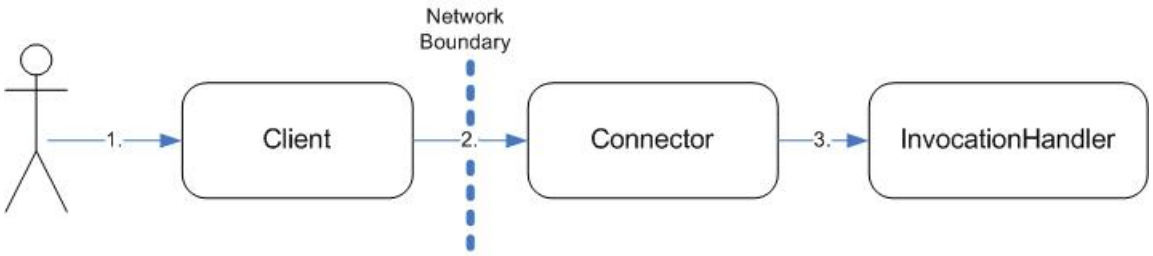
- **Server identification** – a simple String identifier which allows for remoting servers to be identified and called upon.
- **Pluggable transports** – can use different protocol transports, such as socket, rmi, http, etc., via the same remoting API.
- **Pluggable data marshallers** – can use different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

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

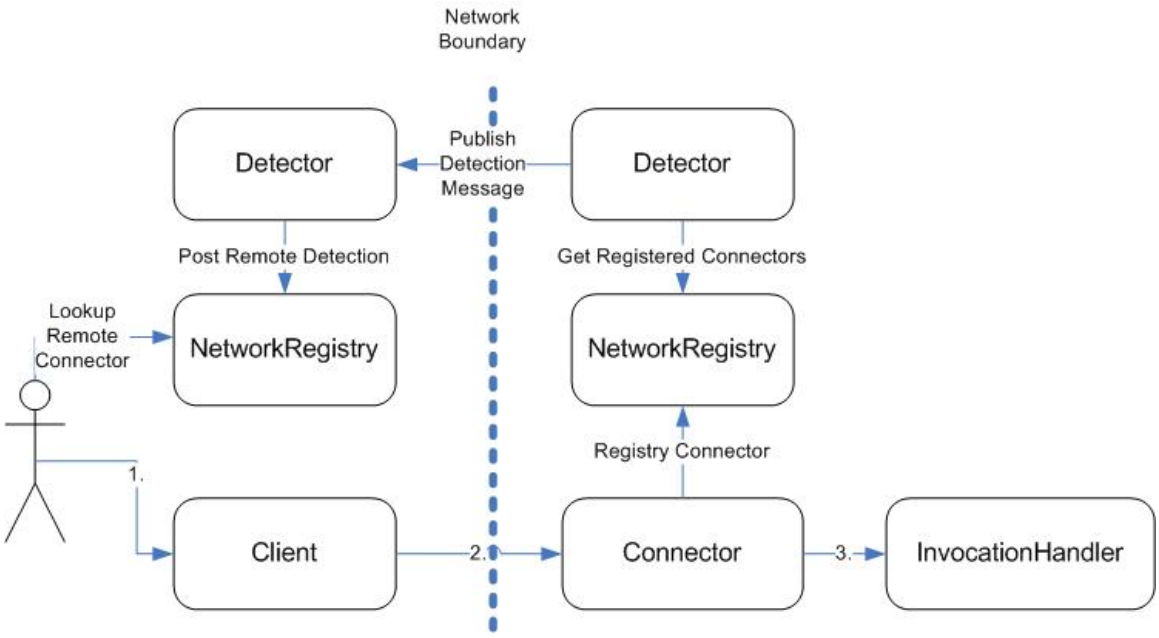
From the highest level, there are three components involved when making a remote invocation using JBoss Remoting; a client, a connector, and an invocation handler.



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>
```

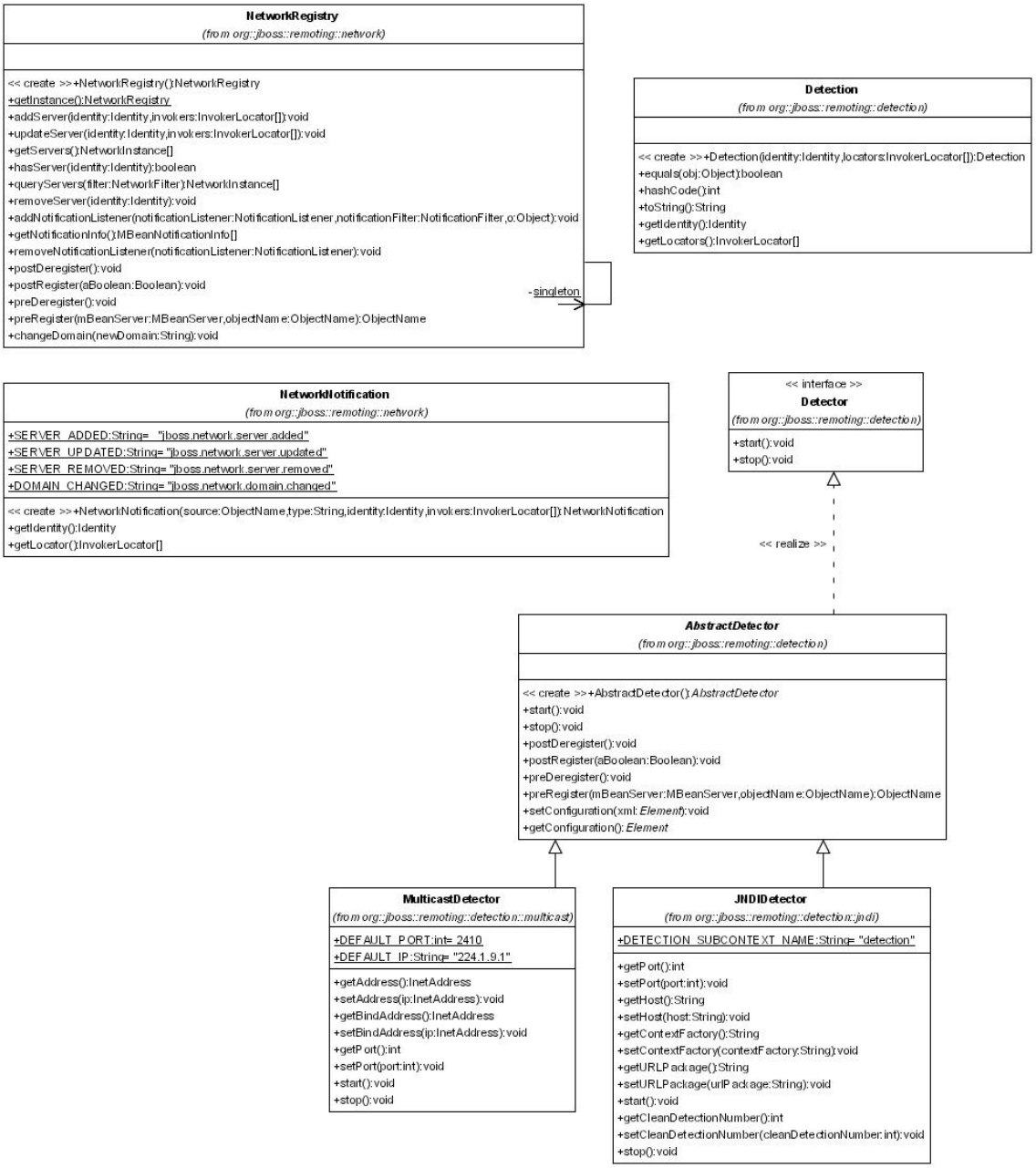
Connector - is an MBean that loads a particular `ServerInvoker` implementation for a given transport subsystem and one or more `ServerInvocationHandler` implementations that handle Subsystem invocations on the remote server JVM. There is exactly one `Connector` per transport type.

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.

How to use it – sample code

This section will cover basic examples of how to use JBoss Remoting and highlight some of the features previously discussed. The source code covered in the examples of this section are provided within the JBoss Remoting distribution (see "How do I get it" section above).

The sample classes discussed 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.

Simple Invocation

To start, we will cover how to make a simple invocation from a remoting client to a remoting server. Let's begin with the server (see org.jboss.samples.simple.SimpleServer). The two main things needed are a Connector and an InvocationHandler. The following shows how the Connector is created, configured, and started.

```
public void setupServer(String locatorURI) throws Exception
 {
   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);
 }
```

In this method, we are passed a locator as represented by a String. The default value will be rmi://localhost:5400. This String is used to create the InvokerLocator for the Connector and is turn what is registered with the NetworkRegistry and will be what the client needs to connect to this remoting server.

Once we have created the Connector, set its locator, and started it, we need to add an invocation handler. The InvocationHandler implementation used for this example is an inner class, SimpleInvocationHandler. The first parameter passed when adding an invocation handler is the name of the sub-system the handler is associated with.

The primary method of concern within the InvocationHandler is the invoke method, as seen here.

```
public Object invoke(InvocationRequest invocation) throws Throwable
{
  // Print out the invocation request
  System.out.println("Invocation request is: " +
                     invocation.getParameter());

  // Just going to return static string
  return RESPONSE_VALUE;
}
```

Here we are just printing out the parameter originally passed by the client. We then return a String, represented in a static constant String variable.

Now let's look at the client code (see org.jboss.samples.simple.SimpleClient).

```
public void makeInvocation(String locatorURI) throws Throwable
{
   InvokerLocator locator = new InvokerLocator(locatorURI);
   System.out.println("Calling remoting server with locator uri of: " + locatorURI);

   // This could have been new Client(locator), but want to show that subsystem param is null
   // Could have also been new Client(locator, "sample");
   Client remotingClient = new Client(locator, null);
   Object response = remotingClient.invoke("Do something", null);

   System.out.println("Invocation response: " + response);
}
```

We create the locator, just as we did in the server code and use it to create the Client instance. Is important to note that the sub-system is not required, but would be needed if there were multiple handlers being used by server connector. Then we make our invocation, passing "Do something" as our parameter. The second parameter is null and is only used to specify protocol specific hints (which will be discussed later, but would include information such as if the HTTP invoker should use POST or GET).

To run the these examples, open two command prompts and go to the examples directory. To run the server, run the ant target 'run-simple-server' and for the client, run the ant target 'run-simple-client'. Note that the server will start and wait 10 seconds for the client to run and then shutdown.

Output from the client is:

```
Calling remoting server with locator uri of: rmi://localhost:5400
org.jboss.remoting.transport.rmi.RMIServerInvoker_Stub[RemoteStub [ref:
[endpoint:[192.168.0.110:5400](remote),objID:[1bd0dd4:1002bed457b:-8000, 0]]]]
Invocation response: This is the return to SampleInvocationHandler invocation
```

Output from the server is:

```
Starting remoting server with locator uri of: rmi://localhost:5400
Invocation request is: Do something
```

Callbacks

Now we will look at setting up callbacks from the server. This example will build off of the previous simple example. First, lets look at the server code. It is exactly the same as the previous simple server, but in the SampleInvocationHandler, have added a collection to store listeners when they are added and changed the invoke() method.

```
public Object invoke(InvocationRequest invocation) throws Throwable
{
  InvocationRequest callbackInvocationRequest = new
       InvocationRequest(invocation.getSessionId(),
       invocation.getSubsystem(), "This is the payload of callback invocation.",
       null, null, invocation.getLocator());
  Iterator itr = listeners.iterator();
  while (itr.hasNext())
  {
    InvokerCallbackHandler callbackHandler = (InvokerCallbackHandler) itr.next();
    callbackHandler.handleCallback(callbackInvocationRequest);
  }

  return RESPONSE_VALUE;

}
```

The invoke method has been changed to call on callback handlers, if any exist, upon being called. Note that the handleCallback() method of CallbackHandler interface requires type InvocationRequest. In this example, we use the values of the InvocationRequest passed to use for most of the parameters used to construct the InvocationRequest we will use for the callback.

The client code for the callback example is also based of the previous simple example, but requires a few more changes. To start, let's look at code required for pull callbacks.

```
public void testPullCallback() throws Throwable
{
  CallbackHandler callbackHandler = new CallbackHandler();
  // by passing only the callback handler, will indicate pull callbacks
  remotingClient.addListener(callbackHandler);
  // now make invocation on server, which should cause a callback to happen
  makeInvocation();

  List callbacks = remotingClient.getCallbacks();
  Iterator itr = callbacks.iterator();
  while (itr.hasNext())
  {
    System.out.println("Callback value = " + itr.next());
  }

  // remove callback handler from server
  remotingClient.removeListener(callbackHandler);
}
```

First, we have to create a CallbackHandler, which is a simple inner class that implements the InvokerCallbackHandler interface. Then we add this listener to the Client instance that has already been created. After making an invocation, which will generate a callback, we call on the client to get any callbacks. We then remove the callback handler from the client, so that callbacks are no longer collected for our handler.

Next, let's look at the code for a push callback. This is a little more complicated as we'll now need a remoting server to receive the callbacks from the remoting server.

```
public void testPushCallback() throws Throwable
{
  // Need to create remoting server to receive callbacks.

  // Using loctor with port value one higher than the target server
  String callbackLocatorURI = transport + "://" + host + ":" + (port + 1);
  InvokerLocator callbackLocator = new InvokerLocator(callbackLocatorURI);

  // call to create remoting server to
  // receive client callbacks.
  setupServer(callbackLocator);

  CallbackHandler callbackHandler = new CallbackHandler();
  // by passing only the callback handler, will indicate pull callbacks
  remotingClient.addListener(callbackHandler, callbackLocator);
  // now make invocation on server, which should cause a callback to happen
  makeInvocation();

  // need to wait for brief moment so server can callback
  Thread.sleep(2000);

  // remove callback handler from server
  remotingClient.removeListener(callbackHandler);
}
```

This is done similar to the way we created one in the CallbackServer class. Now when we call on the client to add the callback listener, we pass the callback handler and the locator for the remoting server we just created. [Note: an interesting point is that the locator we provide does not have to be for a local remoting server, it could be for another remote server, but the practicality for this is minimal.]

Now we a make an invocation on the server using the client and this will cause the invocation handler on the server to generate a callback. In our example, we wait for a few seconds to allow the server to callback on our client callback handler, before we remove the callback handler as a listener. While we are waiting, the CallbackHandler's handleCallback() method should have been called with the callback InvocationRequest.

To run the these examples, open two command prompts and go to the examples directory. To run the server, run the ant target 'run-callback-server' and for the client, run the ant target 'run-callback-client'. Note that the server will start and wait 10 seconds for the client to run and then shutdown.

Output from the client:

```
Calling remoting server with locator uri of: rmi://localhost:5400
org.jboss.remoting.transport.rmi.RMIServerInvoker_Stub[RemoteStub [ref:
[endpoint:[192.168.0.78:5400](remote),objID:[1bd0dd4:1002eca578e:-8000, 0]]]]
Invocation response: This is the return to SampleInvocationHandler invocation
Callback value = org.jboss.remoting.InvocationRequest@544ec1
Starting remoting server with locator uri of: InvokerLocator [rmi://127.0.0.1:5401/]
Received callback value of: This is the payload of callback invocation.
Invocation response: This is the return to SampleInvocationHandler invocation
```

Output from the server:

```
Starting remoting server with locator uri of: rmi://localhost:5400
org.jboss.remoting.transport.rmi.RMIServerInvoker_Stub[RemoteStub [ref:
[endpoint:[192.168.0.78:5401](remote),objID:[ecd7e:1002eca69fb:-8000, 0]]]]
```

Detectors

In this example, we will use the same code from the simple invocation example, but will use automatic detection to determine which server the client will call upon. For the server code, it is almost exactly the same, with the addition of a new method, setupDetector().

```
public void setupDetector() throws Exception
{
   MBeanServer server = MBeanServerFactory.createMBeanServer();

   NetworkRegistry registry = NetworkRegistry.getInstance();
   server.registerMBean(registry, new ObjectName("remoting:type=NetworkRegistry"));

   MulticastDetector detector = new MulticastDetector();
   server.registerMBean(detector, new ObjectName("remoting:type=MulticastDetector"));
   detector.start();
}
```

In this method we have added the code to create and register the NetworkRegistry and a MulticastDetector. Once the detector is started, it will watch for any new Connectors that are started and send out detection messages.

On the client side, we will do basically the same thing, except on the client, we add ourselves as a notification listener so we will be notified when a new server has been discovered.

```
public void setupDetector() throws Exception
{
  MBeanServer server = MBeanServerFactory.createMBeanServer();

  NetworkRegistry registry = NetworkRegistry.getInstance();
  server.registerMBean(registry, new ObjectName("remoting:type=NetworkRegistry"));

  // register class as listener, so know when new server found
  registry.addNotificationListener(this, null, null);

  MulticastDetector detector = new MulticastDetector();
  server.registerMBean(detector, new ObjectName("remoting:type=MulticastDetector"));
  detector.start();
}
```

When the NetworkRegistry is told about a new server being discovered, it will fire a notification, which will call back on our notification listener method.

```
public void handleNotification(Notification notification, Object handback)
{
  if(notification instanceof NetworkNotification)
  {
    NetworkNotification networkNotification = (NetworkNotification)notification;
    InvokerLocator[] locators = networkNotification.getLocator();
    for(int x = 0; x < locators.length; x++)
    {
      try
      {
        makeInvocation(locators[x].getLocatorURI());
      }
      catch (Throwable throwable)
      {
        throwable.printStackTrace();
      }
    }
  }
}
```

Once we get the notification, we will check to see if it is an NetworkNotification and if so, get the locators for the newly found server and make an invocation on it. Also notice that all the invoker variables, such as host, port, transport, have been removed because all this is supplied for us in the NetworkNotification.

To run the these examples, open two command prompts and go to the examples directory. To run the server, run the ant target ‘run-detector-server’ and for the client, run the ant target ‘run-detector-client’. Note that the server will start and wait 10 seconds for the client to run and then shutdown.

Output from the client:

```
Calling remoting server with locator uri of: rmi://127.0.0.1:5400/
org.jboss.remoting.transport.rmi.RMIServerInvoker_Stub[RemoteStub [ref:
[endpoint:[192.168.0.110:5400](remote),objID:[e83912:100307cbfe5:-7fff, 0]]]]
Invocation response: This is the return to SampleInvocationHandler invocation
```

Output from the server:

```
Starting remoting server with locator uri of: rmi://localhost:5400
Invocation request is: Do something
```

Configuration

This covers the configuration for JBoss Remoting discovery, connectors, and transports (as of 1.0.1 beta release). 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 service.xml file. Examples of service.xml files can be seen below.

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).

General Connector and Invoker configuration

Important to note that only one invoker can be declared per connector (so don't put multiple InvokerLocator attributes or invoker elements within the Configuration attribute). At least one handler must also be specified as well. For more information about

handlers, please see the JBoss Remoting User Guide. In short, this 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 TCP_NODELAY set to false and 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/?enabl
eTcpNoDelay=false&clientMaxPoolSize=30]]></attribute>

        <attribute name="Configuration">
<config>
                <handler
subsystem="mock">org.jboss.remoting.transport.mock.MockServerInvocation
Handler</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.MockServerInvocation
Handler</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).

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 instanciate the handler instance upon startup (which requires there be a void parameter constructor), such as:

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

Known issues

This is an beta release of JBoss Remoting, so although this release is more stable than the alpha release, along with more features, it is still not complete. The final release is planned for end of January or early Feburary 2005. Here are some of the high level issues that are currently known. 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 support should be included by final release). The HTTP Invoker has not been stress tested yet, so stability under load is not yet known.

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
- 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 1.0.1 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