JBoss Remoting

Version 1.0.1 alpha

November 15, 2004

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

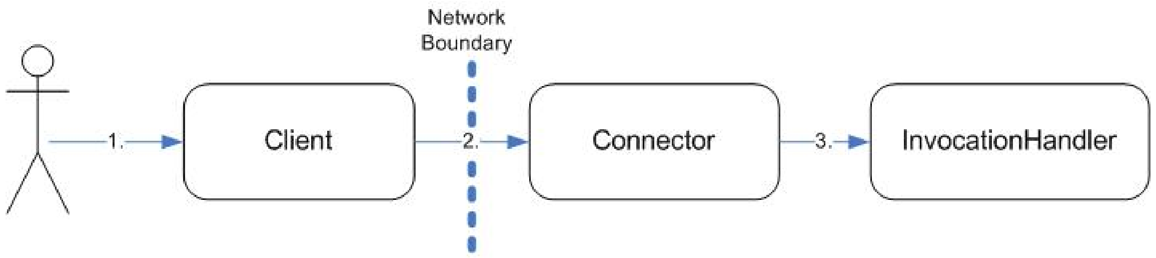
- **Server identification** – a simple String identifier which allows for remoting servers to be identified and called upon.
- **Pluggable transports** – can use different protocol transports, such as socket, rmi, http, etc., via the same remoting API.
- **Pluggable data marshallers** – can use different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

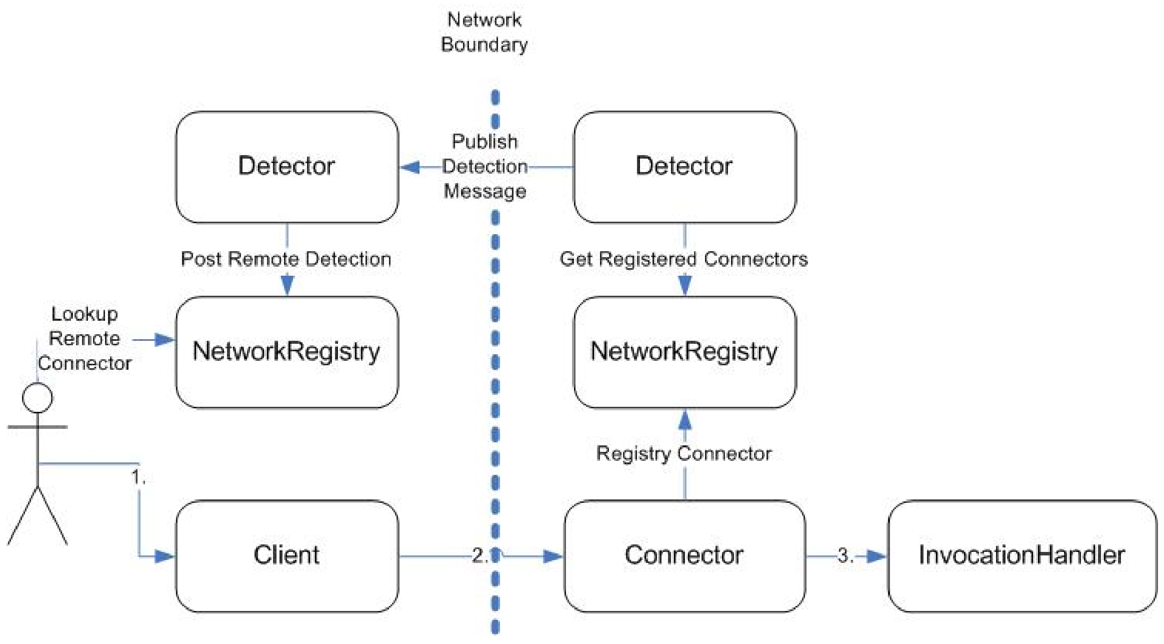
From the highest level, there are three components involved when making a remote invocation using JBoss Remoting; a client, a connector, and an invocation handler.



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.

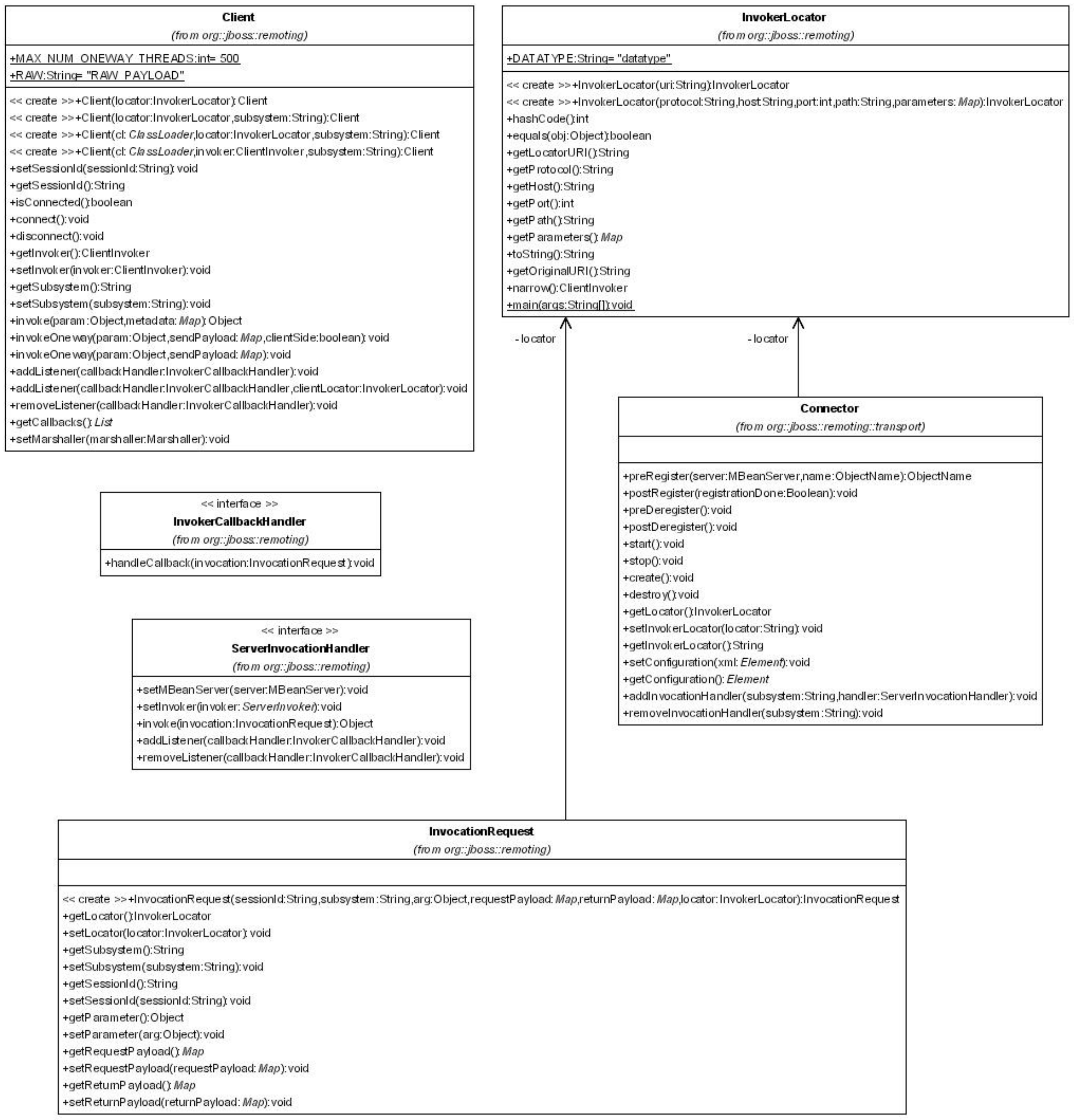


When the Connector is created, it will register itself with the local NetworkRegistry. The Detector on the server will publish a detection message containing the locator for all the Connectors registered with the NetworkRegistry.

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
(from org::jboss::remoting)
+MAX_NUM_ONEWAY_THREADS:int= 500
+RAW:String= "RAW_PAYLOAD"
<< create >>+Client(locator:InvokerLocator):Client
<< create >>+Client(locator:InvokerLocator,subsystem:String):Client
<< create >>+Client(cl: ClassLoader,locator:InvokerLocator,subsystem:String):Client
<< create >>+Client(cl: ClassLoader,invoker:ClientInvoker,subsystem:String):Client
+setSessionId(sessionId:String):void
+getSessionId():String
+isConnected():boolean
+connect():void
+disconnect():void
+getInvoker():ClientInvoker
+setInvoker(invoker:ClientInvoker):void
+getSubsystem():String
+setSubsystem(subsystem:String):void
+invoke(param:Object,metadata: Map):Object
+invokeOneway(param:Object,sendPayload: Map,clientSide:boolean):void
+invokeOneway(param:Object,sendPayload: Map):void
+addListener(callbackHandler:InvokerCallbackHandler):void
+addListener(callbackHandler:InvokerCallbackHandler,clientLocator:InvokerLocator):void
+removeListener(callbackHandler:InvokerCallbackHandler):void
+getCallbacks(): List
+setMarshaller(marshaller:Marshaller):void
InvokerLocator
(from org::jboss::remoting)
+DATATYPE:String= "datatype"
<< create >>+InvokerLocator(uri:String):InvokerLocator
<< create >>+InvokerLocator(protocol:String,host:String,port:int,path:String,parameters: Map):InvokerLocator
+hashCode():int
+equals(obj:Object):boolean
+getLocatorURI():String
+getProtocol():String
+getHost():String
+getPort():int
+getPath():String
+getParameters(): Map
+toString():String
+getOriginalURI():String
+narrow():ClientInvoker
+main(args:String[]):void
- locator
- locator
Connector
(from org::jboss::remoting::transport)
+preRegister(server:MBeanServer,name:ObjectName):ObjectName
+postRegister(registrationDone:Boolean):void
+preDeregister():void
+postDeregister():void
+start():void
+stop():void
+create():void
+destroy():void
+getLocator():InvokerLocator
+setInvokerLocator(locator:String):void
+getInvokerLocator():String
+setConfiguration(xml: Element):void
+getConfiguration(): Element
+addInvocationHandler(subsystem:String,handler:ServerInvocationHandler):void
+removeInvocationHandler(subsystem:String):void
<< interface >>
InvokerCallbackHandler
(from org::jboss::remoting)
+handleCallback(invocation:InvocationRequest):void
<< interface >>
ServerInvocationHandler
(from org::jboss::remoting)
+setMBeanServer(server:MBeanServer):void
+setInvoker(invoker: ServerInvoker):void
+invoke(invocation:InvocationRequest):Object
+addListener(callbackHandler:InvokerCallbackHandler):void
+removeListener(callbackHandler:InvokerCallbackHandler):void
InvocationRequest
(from org::jboss::remoting)
<< create >>+InvocationRequest(sessionId:String,subsystem:String,arg:Object,requestPayload: Map,returnPayload: Map,locator: InvokerLocator):InvocationRequest
+getLocator():InvokerLocator
+setLocator(locator:InvokerLocator):void
+getSubsystem():String
+setSubsystem(subsystem:String):void
+getSessionId():String
+setSessionId(sessionId:String):void
+getParameter():Object
+setParameter(arg:Object):void
+getRequestPayload(): Map
+setRequestPayload(requestPayload: Map):void
+getReturnPayload(): Map
+setReturnPayload(returnPayload: Map):void

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>
```

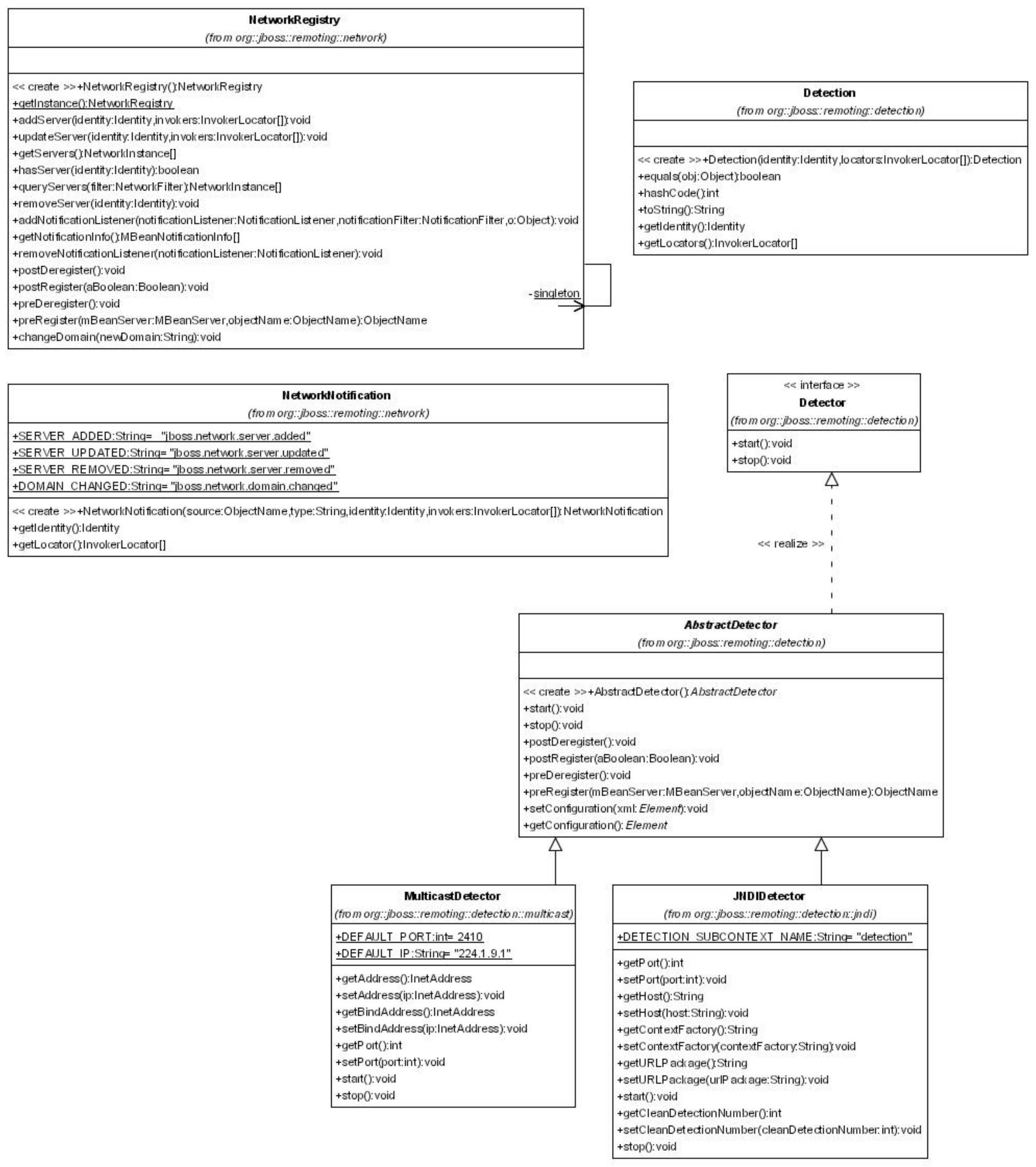
Connector - is an MBean that loads a particular `ServerInvoker` implementation for a given transport subsystem and one or more `ServerInvocationHandler` implementations that handle Subsystem invocations on the remote server JVM. There is exactly one `Connector` per transport type.

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
(from org::jboss::remoting::network)
<< create >>+NetworkRegistry():NetworkRegistry
+getInstance():NetworkRegistry
+addServer(identity:Identity,invokers:InvokerLocator[]):void
+updateServer(identity:Identity,invokers:InvokerLocator[]):void
+getServers():NetworkInstance[]
+hasServer(identity:Identity):boolean
+queryServers(filter:NetworkFilter):NetworkInstance[]
+removeServer(identity:Identity):void
+addNotificationListener(notificationListener:NotificationListener,notificationFilter:NotificationFilter,o:Object):void
+getNotificationInfo():MBeanNotificationInfo[]
+removeNotificationListener(notificationListener:NotificationListener):void
+postDeregister():void
+postRegister(aBoolean:Boolean):void
+preDeregister():void
+preRegister(mBeanServer:MBeanServer,objectName:ObjectName):ObjectName
+changeDomain(newDomain:String):void
-singleton
Detection
(from org::jboss::remoting::detection)
<< create >>+Detection(identity:Identity,locators:InvokerLocator[]):Detection
+equals(obj:Object):boolean
+hashCode():int
+toString():String
+getIdentity():Identity
+getLocators():InvokerLocator[]
NetworkNotification
(from org::jboss::remoting::network)
+SERVER_ADDED:String= "jboss.network.server.added"
+SERVER_UPDATED:String= "jboss.network.server.updated"
+SERVER_REMOVED:String= "jboss.network.server.removed"
+DOMAIN_CHANGED:String= "jboss.network.domain.changed"
<< create >>+NetworkNotification(source:ObjectName,type:String,identity:Identity,invokers:InvokerLocator[]):NetworkNotification
+getIdentity():Identity
+getLocator():InvokerLocator[]
<< interface >>
Detector
(from org::jboss::remoting::detection)
+start():void
+stop():void
<< realize >>
AbstractDetector
(from org::jboss::remoting::detection)
<< create >>+AbstractDetector():AbstractDetector
+start():void
+stop():void
+postDeregister():void
+postRegister(aBoolean:Boolean):void
+preDeregister():void
+preRegister(mBeanServer:MBeanServer,objectName:ObjectName):ObjectName
+setConfiguration(xml:Element):void
+getConfiguration():Element
MulticastDetector
(from org::jboss::remoting::detection::multicast)
+DEFAULT_PORT:int= 2410
+DEFAULT_IP:String= "224.1.9.1"
+getAddress():InetAddress
+setAddress(ip:InetAddress):void
+getBindAddress():InetAddress
+setBindAddress(ip:InetAddress):void
+getPort():int
+setPort(port:int):void
+start():void
+stop():void
JNDIDetector
(from org::jboss::remoting::detection::jndi)
+DETECTION_SUBCONTEXT_NAME:String= "detection"
+getPort():int
+setPort(port:int):void
+getHost():String
+setHost(host:String):void
+getContextFactory():String
+setContextFactory(contextFactory:String):void
+getURLPackage():String
+setURLPackage(urlPackage:String):void
+start():void
+getCleanDetectionNumber():int
+setCleanDetectionNumber(cleanDetectionNumber:int):void
+stop():void

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
```

Known issues

This is an early release of JBoss Remoting, so there is a lot of work being done to finish out the implementation. Here are some of the issues that are currently known. If you find more, please post them to the JBoss Remoting, Unified Invokers forum (http://www.jboss.org/index.html?module=bb&op=viewforum&f=176).

1. Socket invokers not working properly. It should actually work, but is being refactored so can stream the invocation payload, so may be some issues while being refactored.
2. Asyn socket invokers not working. They have been commented out while refactoring.

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 both the JBoss 3.2 and 4.0 branch. 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:

- Fix broken invokers (socket and async sockets)
- HTTP invoker – this has already been started and a simple client version is already coded and server part is in the works as well. Kurt Rush (kurt.rush@gmail.com) has offered to help with this (thanks Kurt ☺).
- Add specific method for streaming large binary files.
- 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

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