JBoss Remoting Users Guide

JBoss Remoting version 2.5.4.SP5

November 9, 2013

Table of Contents

1

Purpose of this document

JBoss Remoting is a project that provides a general purpose distributed invocation framework for other JBoss projects and products, including the community Application Server (AS) and the Enterprise Application Platforms (EAP). Remoting is roughly similar to Java RMI, but instead of using stubs, it identifies servers by URL. Other distributed frameworks that appear in the same context are JGroups (http://www.jgroups.org/) and Netty (http://www.jboss.org/netty). The technologies served by Remoting, in particular, are EJB2, EJB3, and JBoss Messaging (http://labs.jboss.com/jbossmessaging). JBoss Web Services also uses Remoting on its client side, but the use is hidden, so to speak: there are no configuration files.

Remoting is quite flexible, with multiple, pluggable transports, marshallers, serializers, etc., and more than anyone would ever want to know is described in the Remoting Guide [http://docs.jboss.org/jbossremoting/2.5.4.SP3/html/], but the current document, instead, focuses on those aspects of Remoting that are useful in the context of EJB2, EJB3, and JBoss Messaging. The Users Guide is not meant to introduce the reader to writing applications based on Remoting, but rather is meant to provide

1. a basic understanding of how Remoting works,
2. a compendium of the important configuration parameters, and
3. a description of how Remoting is configured in the Application Server.

The information in this guide applies to AS 5, EAP 5, and AS 6.

2

Central concepts

Remoting is based on the client server model, and communication between a client and a server is handled by one of Remoting's several **transports**. A transport is characterized by a pair of classes, one for the client side and one for the server side, which communicate by a shared protocol. Most of the core concepts discussed below are related to one side or the other. However, the URL, or the internal object used to represent the URL, of a server is relevant to both sides.

- **org.jboss.remoting.InvokerLocator:** represents a URL which denotes a Remoting server. For example,

```
socket://localhost:1234/?timeout=10000&maxPoolSize=200&clientMaxPoolSize=50
```

  The protocol element, "socket", indicates that the socket transport is to be used. It will be described below (The socket transport chapter). The values of the parameters "timeout", "maxPoolSize", and "clientMaxPoolSize" are available on both the client and server sides, but they may be ignored on one side or the other. They are further described below (The socket transport chapter).

2.1. Server side concepts

- **org.jboss.remoting.ServerInvocationHandler:**

  is a server side object that incorporates application logic. A `ServerInvocationHandler` may be associated with a particular subsystem, specified by an arbitrary string. For example, JBoss Messaging uses "JMS".

- **org.jboss.remoting.ServerInvoker:**

  is the server side object that fields invocations and passes them to the appropriate `ServerInvocationHandler`. Each of EJB2, EJB3, and JBoss Messaging have their own `ServerInvocationHandlers`. `ServerInvoker` is sub-classed for each transport; e.g., `org.jboss.remoting.transport.socket.SocketServerInvoker`, `org.jboss.remoting.transport.bisocket.BisocketServerInvoker`, etc.

- **org.jboss.remoting.transport.Connector:**

  is the external face of the `ServerInvoker`, by which the `ServerInvoker` is configured declaratively. The configuration includes a designated transport, `ServerInvoker` parameters, and one or more `ServerInvocationHandler` classes.

2.2. Client side concepts

- **org.jboss.remoting.RemoteClientInvoker:**

is a client side object corresponding to a particular `ServerInvoker`. It is responsible for marshalling an invocation to the wire and unmarshalling the response. Each `RemoteClientInvoker` is subclassed for each transport; e.g., `org.jboss.remoting.transport.socket.SocketClientInvoker`, `org.jboss.remoting.transport.bisocket.BisocketClientInvoker`, etc.

- **org.jboss.remoting.Client:**

  is the conduit between application code and the `RemoteClientInvoker`. Multiple `Clients` can share a single `RemoteClientInvoker` if they have the same `InvokerLocator` and the same set of configuration parameters. In a pure Remoting application, the application code would be responsible for creating the `Client`(s). However, in the context of the Application Server, `Clients` and `RemoteClientInvokers` are hidden below the surface. For example, a JBoss Messaging producer or consumer creates one or more `Clients`. In EJB2 and EJB3, `Clients` are create by an interceptor embedded in an object which is retrieved from the server side. In these cases, JBoss Messaging, EJB2, and EJB3 are the applications from the Remoting perspective. A `Client` may be created with a subsystem string which associates it with a particular `ServerInvocationHandler` on the server side. Note that if a `ServerInvoker` is configured with only a single `ServerInvocationHandler`, the use of a subsystem string is optional.

2.3. Callbacks

One more concept is necessary to make sense of the bisocket transport, discussed below in the The bisocket transport chapter. An ordinary invocation leads to a synchronous response from the server, but asynchronous communication is also possible from the server to the client. When a call is make to one of the overloaded variants of the `Client` method

```
addListener(InvokerCallbackHandler callbackHandler) throws Throwable;
```

the passed instance of `InvokerCallbackHandler` is registered as a listener for asynchronous communication, and, on the server side, the `ServerInvocationHandler` associated with the `Client` is given a "proxy" for the `InvokerCallbackHandler` which it can use to send asynchronous `org.jboss.remoting.callback.Callback` objects, or, more simply, **callbacks**.

There are two kinds of callbacks. For **push callbacks**, a dedicated `ServerInvoker` is created on the client side and the server side `ServerInvocationHandler` makes invocations, by way of the "proxy", on that `ServerInvoker` to send callback objects to the client side. Under the surface, the "proxy" creates a `Client` to handle those invocations. On the other hand, **pull callbacks** generated by the `ServerInvocationHandler` are stored on the server side, and a poller is created on the client side which makes invocations on the server side `ServerInvoker` to retrieve any stored callbacks.

3

Declarative configuration

When used in a standalone manner, Remoting clients and servers can be configured either by adding parameters to the InvokerLocator or by directly passing configuration maps to their constructors. In the context of the Application Server, however, Remoting objects are configured in xml files. There are two variations. In versions 4 and earlier of the Application Server, the various components are held together by an MBeanServer, and, in the course of initializing the Application Server, deployers create MBeans from xml description files. In version 5, the primacy of the MBeanServer has been replaced by the JBoss Microcontainer, which creates POJOs from xml description files. Both MBean and POJO descriptors are available in version 5.

3.1. MBean descriptors

An MBean descriptor of a Remoting `Connector` looks like the following abbreviated version derived from a JBoss Messaging example:

```
<mbean code="org.jboss.remoting.transport.Connector"
       name="jboss.messaging:service=Connector,transport=bisocket"
       display-name="Bisocket Transport Connector">
   <attribute name="Configuration">
      <config>
         <invoker transport="bisocket">
            ...
            <attribute name="serverBindAddress">localhost</attribute>
            <attribute name="serverBindPort">4457</attribute>
            <attribute name="marshaller" isParam="true">org.jboss.jms.wireformat.JMSWireFormat</attrib
            <attribute name="unmarshaller" isParam="true">org.jboss.jms.wireformat.JMSWireFormat</attr
            <attribute name="timeout" isParam="true">300000</attribute>
            <attribute name="callbackTimeout">10000</attribute>
            ...
         </invoker>
         <handlers>
            <handler subsystem="JMS">org.jboss.jms.server.remoting.JMSServerInvocationHandler</handler
         </handlers>
      </config>
   </attribute>
</mbean>
```

In this case, the `Connector` reads the <config> attribute and creates the InvokerLocator

```
bisocket://localhost:4457?marshaller=org.jboss.jms.wireformat.JMSWireFormat&unmarshaller=org.jboss.jms.wi
```

which is passed into the `BisocketServerInvoker` that it creates (see The bisocket transport chapter).

Notes.

1. The <transport> element specifies that this `Connector` will create a `BisocketServerInvoker`.

2. The "marshaller", "unmarshaller", and "timeout" parameters, which are all specified with the "isParam" attribute set to "true", appear in the InvokerLocator. They will be available on both the server side and the client side.

3. The "callbackTimeout" parameter, which is specified without the "isParam" attribute, does not appear in the InvokerLocator. It will be available on the server side but will not be available on the client side.

4. The <handler> element tells the `Connector` to create an instance of `org.jboss.jms.server.remoting.JMSServerInvocationHandler` to handle all invocations associated with the "JMS" subsystem.

3.2. POJO descriptors

When an MBean descriptor like the one in the previous section is given, the `Connector` explicitly parses the <config> element. In Application Server 5 and EAP 5, the microcontainer automatically parses POJO descriptors written in its descriptor language, creates the described POJOs, and can inject them into other POJOs. Remoting uses a POJO descriptor of a `org.jboss.remoting.ServerConfiguration` object to inject configuration information into a `Connector`. For example, the following POJO descriptors are abbreviated and modified versions of those used by the EJB2 subsystem:

```
<bean name="UnifiedInvokerConnector" class="org.jboss.remoting.transport.Connector">
   <property name="serverConfiguration"><inject bean="UnifiedInvokerConfiguration"/></property>
</bean>

<bean name="UnifiedInvokerConfiguration" class="org.jboss.remoting.ServerConfiguration">
   <constructor>
      <parameter>socket</parameter>
   </constructor>

   <!-- Parameters visible to both client and server -->
   <property name="invokerLocatorParameters">
      <map keyClass="java.lang.String" valueClass="java.lang.String">
         <entry>
            <key>serverBindAddress</key>
            <value>localhost</value>
         </entry>
         <entry>
            <key>serverBindPort</key>
            <value>4446</value>
         </entry>
         <entry><key>enableTcpNoDelay</key> <value>true</value></entry>
      </map>
   </property>

   <!-- Parameters visible only to server -->
   <property name="serverParameters">
      <map keyClass="java.lang.String" valueClass="java.lang.String">
         <entry><key>maxPoolSize</key><value>500</value></entry>
      </map>
   </property>

   <property name="invocationHandlers">
      <map keyClass="java.lang.String" valueClass="java.lang.String">
         <entry><key>JSR88</key> <value>org.jboss.deployment.remoting.DeployHandler</value></entry>
```

```
        </map>
      </property>
   </bean>
```

Here, the `ServerConfiguration` is created and injected into the `Connector` object, which creates the InvokerLocator

```
socket://localhost:4446?enableTcpNoDelay=true
```

Notes.

1. The <constructor> element specifies that this `Connector` will create a `SocketServerInvoker`.
2. The entries in the "invokerLocatorParameters" property appear in the InvokerLocator and are available on both the client and server sides.
3. The entries in the "serverParameters" property do not appear in the InvokerLocator and are available only on the server side.
4. The "invocationHandlers" property tells the `Connector` to create an instance of `org.jboss.deployment.remoting.DeployHandler` to handle all invocations associated with the "JSR88" subsystem.

The actual `ServerInvocationHandler` used by EJB2, by the way, is an instance of a different class and is injected into the `SocketServerInvoker` programmatically. It is described by the following POJO descriptor:

```
<bean name="UnifiedInvoker" class="org.jboss.invocation.unified.server.UnifiedInvoker">
   <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(name="jboss:service=invoker,type=unified'
   <property name="connector"><inject bean="UnifiedInvokerConnector"/></property>
   <depends>TransactionManager</depends>
</bean>
```

An `org.jboss.invocation.unified.server.UnifiedInvoker` is a `ServerInvocationHandler`, and it uses the "UnifiedInvokerConnector" to inject itself into the `SocketServerInvoker` created by the "UnifiedInvokerConnector". The <annotation> property, by the way, is used to assign an MBean name to a POJO.

There is an alternative way to configure a `Connector` with a `ServerConfiguration`. In the following modified version of two POJOs used by the EJB3 subsystem, the `InvokerLocator` is given explicitly to the `Connector` and the injected `ServerConfiguration` is used only to specify a `ServerInvocationHandler`:

```
<bean name="org.jboss.ejb3.RemotingConnector"
  class="org.jboss.remoting.transport.Connector">

   <property name="invokerLocator">
      <value>socket://localhost:3873?timeout=300000</value>
   </property>
   <property name="serverConfiguration">
      <inject bean="ServerConfiguration" />
   </property>
</bean>
```

```
<!-- Remoting Server Configuration -->
<bean name="ServerConfiguration"
  class="org.jboss.remoting.ServerConfiguration">
   <property name="invocationHandlers">
      <map keyClass="java.lang.String" valueClass="java.lang.String">
         <entry>
            <key>AOP</key>
            <value>
               org.jboss.aspects.remoting.AOPRemotingInvocationHandler
            </value>
         </entry>
      </map>
   </property>
</bean>
```

4

The socket transport

A Remoting transport is represented by a matching pair of classes derived from `RemoteClientInvoker` and `SocketInvoker` (see the Central concepts chapter). They are matched in the sense that they share a wire protocol. In the socket transport, the classes are `org.jboss.remoting.transport.socket.SocketClientInvoker` and `org.jboss.remoting.transport.socket.SocketServerInvoker`. `SocketClientInvoker` is, in fact, further subclassed by the actual client invoker class, `org.jboss.remoting.transport.socket.MicroSocketClientInvoker`.

The socket transport is a relatively low level transport in the sense that it interacts directly with Java sockets and server sockets. `SocketServerInvoker` is a classic old i/o style server, in which it listens to a server socket and, for each new socket created, it either creates a new worker thread, represented by `org.jboss.remoting.transport.socket.ServerThread` or reuses an idle `ServerThread` obtained from a pool. A `ServerThread` manages a single socket, waiting for an invocation to come in on the wire, passing the invocation to the `ServerInvoker` which passes it to the appropriate `ServerInvokerHandler`, and then writing the response to the wire.

On the client side, `MicroSocketClientInvoker` takes an invocation passed in by a `Client`, either creates a new connection that wraps a Java socket or reuses an idle connection from a pool, writes the invocation to the socket, and reads and returns the result. An important consideration is the determination of the viability of a pooled connection. A slow but reliable method is to write a byte to the socket and wait for the byte to come back. This method is turned off by default. Another method is for the server side to write a couple of bytes when the connection is closed, and for the client side to check for available bytes. This method is always turned on, but it is less reliable since the server side may be unable to write the bytes, or they may not have arrived yet when the test is made. A palliative for these possibilities is to configure `MicroSocketClientInvoker` to retry a failed invocation in the event of a `java.net.SocketException` or certain variants of a `java.io.IOException`.

A number of parameters may be configured on each side, including, for example, timeout times, pool sizes, and number of retry attempts. Note that most of these are given their default values by JBoss Messaging, EJB2, and EJB3, and, in general, the configured values of the others should not be changed without a good reason.

4.1. Server side parameters

acceptThreadPriorityIncrement - can be used to increment the priority of the accept thread, which manages the `ServerSocket`. The value is added to `java.lang.Thread.NORM_PRIORITY`, and the resulting value must be no more than `java.lang.Thread.MAX_PRIORITY`. This parameter might be useful on a heavily loaded machine if the accept thread is getting starved. Introduced in release 2.5.4.SP2.

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. The default value is 200.

continueAfterTimeout - indicates what a `ServerThread` should do after experiencing a

`java.net.SocketTimeoutException`. If set to true, or if JBoss Serialization is being used (which is never the case, by default, which is why JBoss Serialization is not discussed in this guide), the server thread will continue to wait for the next invocation; otherwise, it will return itself to the thread pool. For Java serialization, the default value is false.

evictabilityTimeout - indicates the number of milliseconds during which a `ServerThread` waiting for the next invocation will not be interruptible. The default value is 10000 milliseconds.

idleTimeout - indicates the number of seconds a pooled `ServerThread` can be idle, that is, waiting on the next invocation, before it should be removed from the thread pool. The value for this property must be greater than zero in order to enable idle timeouts. The default value is -1.

immediateShutdown - indicates, when set to "true", that, when `Connector.stop()` is called and it calls `SocketServerInvoker.stop()`, all `ServerThreads` are shut down immediately, even if they are processing an invocation. The default value is false.

maxPoolSize - the maximum number of `ServerThreads` that can exist at any given time. The default value is 300.

numAcceptThreads - the number of threads listening on the `ServerSocket`. The default value is 1.

serverSocketClass - specifies the fully qualified class name for a custom `SocketWrapper` implementation to use on the server. By default, `org.jboss.remoting.transport.socket.ServerSocketWrapper` is used. JBoss Messaging uses a custom wrapper.

socket.check_connection - indicates if a client side pooled connection should be checked by sending a single byte from the client to the server and then back to the client. This parameter needs to be set on both client and server to work. It is false by default.

timeout - the socket timeout value passed to the `Socket.setSoTimeout()` method. The default on the server side is 60000 milliseconds.

writeTimeout - a timeout value imposed on socket write operations. This feature is enabled by setting writeTimeout to a value, in milliseconds, greater than zero. By default, the feature is not enabled.

The following socket parameters, in addition to SO_TIMEOUT, can be configured on the server side: SO_KEEPALIVE, OOBINLINE, SO_RCVBUF, SO_REUSEADDR, SO_SNDBUF, SO_LINGER, and "traffic class". They are configured by passing the following parameter keys to `SocketServerInvoker`:

keepAlive - sets socket parameter SO_KEEPALIVE

oOBInline - sets socket parameter OOBINLINE

receiveBufferSize - sets socket parameter SO_RCVBUF

reuseAddress - sets socket parameter SO_REUSEADDR

sendBufferSize - sets socket parameter SO_SNDBUF

soLinger - sets socket parameter SO_LINGER

soLingerDuration - when socket parameter SO_LINGER is set to "true", sets linger duration

trafficClass - sets socket traffic class

4.2. Client side parameters

clientMaxPoolSize - the maximum number of socket connections that can exist at any given time. The default value is 50.

clientSocketClass - specifies the fully qualified class name for a custom `SocketWrapper` implementation to use on the client. By default, `org.jboss.remoting.transport.socket.ClientSocketWrapper` is used. JBoss Messaging uses a custom wrapper.

generalizeSocketException - If set to false, a failed invocation will be retried in the case of `SocketExceptions`. If set to true, a failed invocation will be retried in the case of `SocketExceptions` and also any `IOException` whose message matches the regular expression `^.*(?:connection.*reset|connection.*closed|connection.*abort|broken.*pipe|connection.*shutdown).*$`. See also the "numberOfCallRetries" parameter, below. The default value is false.

numberOfCallRetries - the number of times a failed invocation will be retried. For example, it is possible that the server side of a socket connection could time out, leaving the connection invalid. In that case, the socket will be discarded and another, possibly new, socket will be used. After numberOfCallRetries attempts, an `InvocationFailureException`, whose cause is the original exception, will be thrown. The default value is 3. See also the "generalizeSocketException" parameter, above.

socket.check_connection - indicates if a client side pooled connection should be checked by sending a single byte from the client to the server and then back to the client. This parameter needs to be set on both client and server to work. It is false by default.

timeout - The socket timeout value passed to the Socket.setSoTimeout() method. The default on the client side is 1800000 milliseconds (30 minutes).

useOnewayConnectionTimeout - indicates if, during a client side oneway invocation, `MicroSocketClientInvoker` should wait for a version byte from the server, which prevents the anomalous behavior described in JBREM-706 "In socket transport, prevent client side oneway invocations from artificially reducing concurrency". The default value is true.

writeTimeout - a timeout value imposed on socket write operations. This feature is enabled by setting writeTimeout to a value, in milliseconds, greater than zero. By default, the feature is not enabled.

The following socket parameters, in addition to SO_TIMEOUT, can be configured on the client side: TCP_NODELAY, SO_KEEPALIVE, OOBINLINE, SO_RCVBUF, SO_REUSEADDR, SO_SNDBUF, SO_LINGER, and "traffic class". They are configured by passing the following parameter keys to `MicroSocketClientInvoker`:

enableTcpNoDelay - sets socket parameter TCP_NODELAY. The default value is false.

keepAlive - sets socket parameter SO_KEEPALIVE

oOBInline - sets socket parameter OOBINLINE

receiveBufferSize - sets socket parameter SO_RCVBUF

reuseAddress - sets socket parameter SO_REUSEADDR. The default value is true.

sendBufferSize - sets socket parameter SO_SNDBUF

soLinger - sets socket parameter SO_LINGER

soLingerDuration - when socket parameter SO_LINGER is set to "true", sets linger duration

trafficClass - sets socket traffic class

5

The bisocket transport

The bisocket transport is derived from the socket transport and differs in one design feature. It was created for JBoss Messaging, where the design criteria were

1. message payloads are be sent to client side consumers in callbacks,
2. push callbacks, being faster, are preferred, and
3. opening ports on the client side is undesirable.

In the socket transport, push callbacks are handled on the client side by a dedicated `SocketServerInvoker`, which uses a Java server socket; that is, it opens a port. When a connection is needed on the server side to send push callbacks, a socket is created by contacting the server socket on the client side.

The bisocket transport, instead, uses server sockets only on the server side, including (1) the one created by `SocketServerInvoker` to create connections to handle ordinary invocations, and (2) a **secondary server socket**. The first use of the secondary server socket is to create a **control connection** in response to a request from the client side. Subsequently, all connections for sending push callbacks are created by sending a message over the control connection to the client side, asking it to make a connection to the secondary server socket.

If the control connection fails, then no new calback connections can be created, so it is possible to configure the server side `BisocketServerInvoker` to ping the client side `BisocketClientInvoker` at fixed intervals. If the `BisocketClientInvoker` detects a missing ping, it can connect to the secondary server socket and recreate the control connection.

Originally, JBoss Messaging turned off pinging because the default ping period was small enough to cause spurious failures. Currently, the default values are accepted.

Note. If opening ports on the client side is not a problem, then JBoss Messaging will run just fine with the somewhat simpler socket transport.

The following bisocket configuration parameters are added to those used in the socket transport:

5.1. Server side parameters

acceptThreadPriorityIncrement - can be used to increment the priority of the thread which manages the secondary `ServerSocket` (as well as the thread that manages the primary `ServerSocket` in the socket transport). The value is added to `java.lang.Thread.NORM_PRIORITY`, and the resulting value must be no more than `java.lang.Thread.MAX_PRIORITY`. This parameter might be useful on a heavily loaded machine if the accept thread is getting starved. Introduced in release 2.5.4.SP2.

pingFrequency - the frequency, in milliseconds, with which the server side sends a ping over the control connection. The default value is 5000 milliseconds.

secondaryMaxThreads: Determines the maximum number of sockets accepted by the secondary `ServerSocket` that can be processed simultaneously. Introduced in release 2.5.4.SP2. Default value is 50.

secondaryTimeout: Determines the timeout used when reading initial data from a socket accepted by the secondary `ServerSocket`. Introduced in release 2.5.4.SP2. Default value is 60000 ms.

5.2. Client side parameters

maxControlConnectionRestarts - the maximum number of times the client side will attempt to recreate the control connection following a ping failure. The default value is 10.

maxRetries - the maximum number of times two processes are attempted: (1) the attempt to create a socket, either for the control connection or for a callback connection, and (2) the attempt to get the port of the secondary server socket. The default value is 10.

pingFrequency - the frequency, in milliseconds, with which the server side is expected to send a ping over the control connection. The default value is 5000 milliseconds.

pingWindowFactor - the value which, when multiplied times the pingFrequency, gives the window within which a ping is expected by the client side. The default value is 2. It follows that the default window is 10 seconds.

secondaryBindPort - the port to which the secondary server socket is to be bound. By default, an arbitrary port is selected.

secondaryConnectPort - the port clients are to use to connect to the secondary server socket. By default, the value of secondaryBindPort is used. secondaryConnectPort is useful if the server is behind a translating firewall.

6

The sslsocket and sslbisocket transports

The sslsocket transport is derived from the socket transport (The socket transport) and differs only in the use of `javax.net.ssl.SSLSockets` and `javax.net.ssl.SSLServerSockets` instead of the usual Java sockets and server sockets. Similarly, the sslbisocket transport is derived from the bisocket transport (The bisocket transport) and differs only in the use of `SSLSockets` and `SSLServerSockets`.

6.1. Server side

Remoting provides a configurable extension of `javax.net.ssl.SSLServerSocketFactory` called `org.jboss.remoting.security.SSLSocketFactoryService`. It depends on an instance of `org.jboss.remoting.security.SSLSocketBuilder`, which creates and configures an instance of `javax.net.ssl.SSLContext` and uses it to create an instance of `javax.net.SSLServerSocketFactory`. It is the `SSLSocketBuilder` which can be configured with keystores, etc. `SSLSocketBuilder` is described in more detail in the Remoting Guide [http://docs.jboss.org/jbossremoting/2.5.3.SP1/html/].

For example, JBoss Messaging uses instances of `SSLSocketFactoryService` and `SSLSocketBuilder` as follows in its configuration of the sslbisocket transport:

```
<mbean code="org.jboss.remoting.security.SSLServerSocketFactoryService"
       name="jboss.messaging:service=ServerSocketFactory,type=SSL"
       display-name="SSL Server Socket Factory">
   <depends optional-attribute-name="SSLSocketBuilder"
            proxy-type="attribute">jboss.messaging:service=SocketBuilder,type=SSL</depends>
</mbean>

<mbean code="org.jboss.remoting.security.SSLSocketBuilder"
       name="jboss.messaging:service=SocketBuilder,type=SSL"
       display-name="SSL Server Socket Factory Builder">

   <!--
        IMPORTANT - If making ANY customizations, this MUST be set to false.
        Otherwise, will use default settings and the following attributes will be ignored.
   -->
   <attribute name="UseSSLServerSocketFactory">false</attribute>

   <attribute name="KeyStoreURL">${jboss.server.home.url}/deploy/messaging/messaging.keystore</attribu
   <attribute name="KeyStorePassword">secureexample</attribute>
   <attribute name="KeyPassword">secureexample</attribute>
   <attribute name="SecureSocketProtocol">TLS</attribute>
   <attribute name="KeyStoreAlgorithm">SunX509</attribute>
   <attribute name="KeyStoreType">JKS</attribute>
</mbean>
```

Through these two MBeans, JBoss Messaging provides itself with a suitably configured instance of an `SSLServerSocketFactory` on the server side.

6.2. Client side

Although a `SSLSocketBuilder` can be used to create a `javax.net.SSLSocketFactory`, the one on the server side typically will not be available on the client side, so it is the responsibility of the particular subsystem (JBoss Messaging, EJB2, EJB3) to create a `RemoteClientInvoker` with a suitable `SSLSocketFactory`. For example, JBoss Messaging takes two steps:

1. it uses a Remoting transport, sslbisocket, whose `SSLBisocketClientInvoker` is designed to create an `SSLSocketFactory` through the use of a properly configured `SSLSocketBuilder`, and

2. it captures appropriate SSL parameters and passes them to the `SSLBisocketClientInvoker`:

```
Map configuration = new HashMap();

String trustStoreLoc = System.getProperty("org.jboss.remoting.trustStore");
if (trustStoreLoc != null)
{
   configuration.put("org.jboss.remoting.trustStore", trustStoreLoc);
   String trustStorePassword = System.getProperty("org.jboss.remoting.trustStorePassword");
   if (trustStorePassword != null)
   {
      configuration.put("org.jboss.remoting.trustStorePassword", trustStorePassword);
   }
}
...
Client client = new Client(new InvokerLocator(serverLocatorURI), configuration);
...
```

6.3. Additional parameters

The following parameters are applicable to both the client and server sides for the sslsocket and sslbisocket transports:

enabledCipherSuites - a String array which is passed to `SSLSocket.setEnabledCipherSuites()`

enabledProtocols - a String array which is passed to `SSLSocket.setEnabledProtocols()`

enableSessionCreation - a boolean value which is passed to `SSLSocket.setEnableSessionCreation()`

7

The http transport family

Unlike the socket transport and its derivatives, which interact directly with `Sockets` and `ServerSockets`, the http family of transports uses `java.net.HttpURLConnections`. While they do not exhibit the same performance as the socket transports, they have the advantage of using the universal http protocol.

Any of JBoss Messaging, EJB2, and EJB3 can use the http transports instead of the socket family of transports. Additional configuration information may be found in the wiki articles EJB, JMS and JNDI over HTTP with Unified Invoker [http://community.jboss.org/wiki/EJBJMSandJNDIoverHTTPwithUnifiedInvoker] and EJB, JMS and JNDI over HTTP via NAT Firewall with Unified Invoker [http://community.jboss.org/wiki/EJBJMSandJNDIoverHTTPviaNATFirewallwithUnifiedInvoker].

7.1. http and https transports

On the client side, an `org.jboss.remoting.transport.http.HTTPClientInvoker`, a subclass of `RemoteClientInvoker`, creates a `java.net.HttpURLConnection` for each invocation. The caching of `HttpURLConnections` and their `Socket` is left to the implementation. On the server side, the `org.jboss.remoting.transport.coyote.CoyoteInvoker`, which is a subclass of `ServerInvoker` and is based on the coyote module in Tomcat, processes http requests, calls on `ServerInvoker` to hand invocations off to the appropriate `ServerInvocationHandler`, and returns a result along with a response code.

The https transport is derived from the http transport and uses `javax.net.ssl.SSLSockets` and `javax.net.ssl.SSLServerSockets` instead of `Sockets` and `ServerSockets`.

7.2. servlet and sslservlet transports

The servlet and sslservlet transports share the client side code of the http and https transports, respectively. On the server side, the difference is that the servlet and sslservlet transports use a servlet, `org.jboss.remoting.transport.servlet.web.ServerInvokerServlet`, to hand an invocation off to an `org.jboss.remoting.transport.servlet.ServletServerInvoker`. In other words, the servlet and sslservlet transports share a port with all other servlets running in the Application Server, while the http and https transports use a separate port managed by a `CoyoteInvoker`.

When the `ServerInvokerServlet` is initialized, it needs to be informed of which `ServletServerInvoker` to use. One way of doing that is to give it the appropriate `InvokerLocator`. For example, the following web.xml file is used by JBoss Messaging:

```
<web-app>

  <servlet>
    <servlet-name>JmsServerInvokerServlet</servlet-name>
```

```
      <description>The JmsServerInvokerServlet receives JMS requests via HTTP
        protocol from within a web container and passes it onto the
        ServletServerInvoker for processing.
      </description>
      <servlet-class>org.jboss.remoting.transport.servlet.web.ServerInvokerServlet</servlet-class>
      <init-param>
         <param-name>locatorUrl</param-name>
         <param-value>
            <![CDATA[servlet://${jboss.bind.address}:8080/servlet-invoker/JmsServerInvokerServlet/?dataT
         </param-value>
         <description>The servlet server invoker</description>
      </init-param>
      <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
      <servlet-name>JmsServerInvokerServlet</servlet-name>
      <url-pattern>/JmsServerInvokerServlet/*</url-pattern>
    </servlet-mapping>

  </web-app>
```

7.3. Server side parameters: http and https transports

In general, parameters passed to a `CoyoteInvoker` are passed to the underlying Tomcat implementation. For example, the Tomcat attribute "maxThreads" can be used to set the maximum number of threads to be used to accept incoming http requests. However, the Remoting attributes "serverBindAddress" and "serverBindPort" should be used instead of the Tomcat attributes "address" and "port". For more information on the configuration attributes available for the Tomcat connectors, please refer to http://tomcat.apache.org/tomcat-6.0-doc/config/http.html.

`CoyoteInvoker` can also configure an instance of `org.apache.catalina.Executor`, an interface which extends `java.util.concurrent.Executor`. If the parameter "executor" is associated with a comma separated list of key=value pairs, `CoyoteInvoker` will parse the list, use the resulting `Map` to configure an instance of `org.apache.catalina.core.StandardThreadExecutor`, and inject it into the underlying Tomcat implementation. For example, JBoss Messaging's `remoting-http-service.xml` configuration file could contain

```
<attribute name="executor">minSpareThreads=20,maxThreads=100</attribute>
```

Note that if an executor is injected, the other threadpool related attributes such as "maxThreads", which would otherwise be applied to a threadpool created by Tomcat, are ignored. The configurable attributes of the `StandardThreadExecutor` are described in http://tomcat.apache.org/tomcat-6.0-doc/config/executor.html.

7.4. Server side parameters: servlet and sslservlet transports

The following parameter may be used to configure an instance of ServletServerInvoker:

unwrapSingletonArrays - key indicating if, when `javax.servlet.http.HttpServletRequest.getParameterMap()` returns a map containing pairs of the form (key, value) where value has the form `java.lang.String[]` with length 1, the pair should be replaced with a pair (key, ((String[] value)[0]). The default value is "false".

7.5. Client side parameters: http, https, servlet, sslservlet transports

The following parameters may be used to configure an instance of `org.jboss.remoting.transport.http.HTTPClientInvoker`:

disconnectAfterUse - key indicating if the `java.net.HttpURLConnection` should be disconnected after the invocation. The default value is "false".

ignoreErrorResponseMessage - key indicating if `HTTPClientInvoker` should try to get a response message and response code in the event of an exception. The default value is "false".

numberOfCallAttempts - This parameter is relevant only on the client side, where it determines the maximum number of attempts that will be made to complete an invocation. The default value is 1.

unmarshalNullStream - key indicating if `HTTPClientInvoker` should make the call to `UnMarshaller.read()` when the `InputStream` is null. The default value is "true".

7.6. Client side parameters: https and sslservlet transports

The following parameters may be used to configure an instance of `HTTPSClientInvoker`:

org.jboss.security.ignoreHttpsHost - key indicating if `HTTPSClientInvoker` should ignore host name verification, i.e., it will not check for a mismatch between the URL's hostname and the server's hostname during handshaking. By default, standard hostname verification will be performed.

hostnameVerifier - key indicating the hostname verifier that should be used by `HTTPSClientInvoker`. The value should be the fully qualified classname of a class that implements `javax.net.ssl.HostnameVerifier` and has a void constructor.

useDefaultSslSocketFactory - if set to true, then `HTTPSClientInvoker.createSocketFactory()` will call `HttpsURLConnection.getDefaultSSLSocketFactory()` instead of trying to build a SocketFactory.

7.7. JBoss Messaging, http transports, and callbacks

Unlike the bisocket transport, which was designed especially to allow push callbacks without opening a port on the client side, a server in any of the http transports necessarily opens a `ServerSocket`, so push callbacks are ruled out when JBoss Messaging runs on any of the http transports. Pull callbacks are used instead, and there are some configuration parameters that can be set:

blockingMode - indicates whether to use blocking or nonblocking mode when doing pull callbacks. In nonblocking mode a poller periodically polls for waiting callbacks, and, if there are none, returns. In blocking mode, the poller periodically polls for waiting callbacks, and, if there are none, the call blocks on the server side until a callback is made available, at which point the poller immediately retrieves the callback to the client side. Blocking mode, then, is more responsive, and it is used by JBoss Messaging. By default, nonblocking mode is used.

blockingTimeout - when pull callbacks are used in blocking mode, indicates the amount of time the callback

poller should block on the server side waiting for a callback. The default value is 5000 milliseconds, but JBoss Messaging uses 30000 milliseconds.

8

Network Connection Monitoring

Remoting has two mechanisms for monitoring the health of established connections, which inform listeners on the client and server sides when a possible connection failure has been detected. Currently, only JBoss Messaging uses these facilities. Note that JBoss Messaging establishes connections programmatically, so some unused declarative configuration options are omitted from this discussion.

8.1. Server side monitoring

A remoting server has the capability to detect when a client is no longer available. This is done by establishing an `org.jboss.remoting.Lease` on the server side, managed by a `ServerInvoker`. On the client side, an `org.jboss.remoting.LeasePinger` periodically sends PING messages to the server, and on the server side an `org.jboss.remoting.Lease` informs registered listeners if the PING doesn't arrive within the specified timeout period. A `LeasePinger` is created by a `RemoteClientInvoker` and it sends PINGS to a particular `Lease`. That `LeasePinger`/`Lease` pair defines, for purposes of connection monitoring in Remoting, the abstract concept of a **connection**.

The following parameter is relevant to leasing configuration on the server side:

`clientLeasePeriod` - specifies the timeout period used by the server to determine if a PING is late. The default value is "5000". This is also the suggested lease period returned by the server when the client inquires if leasing is activated.

The following parameters are relevant to leasing configuration on the client side:

`lease_period` - if set to a value greater than 0 and less than the suggested lease period returned by the server, will be used to determine the time between PING messages sent by `LeasePinger`. This parameter is not used by JBoss Messaging.

`leasePingerTimeout` - specifies the per invocation timeout value use by `LeasePinger` when it sends PING messages. This parameter is not used by JBoss Messaging.

The actual lease window established on the server side is initially set to twice the clientLeasePeriod value, but it can expand dynamically to adjust to actual network conditions. As long as PINGs arrive within 75% of the lease window, the window will remain unchanged. However, if a PING arrives at between 75% and 100% of the lease window, the lease window will be expanded to twice the time since the previous PING. For example, if the current lease window is 20 seconds, and if a PING arrives 17 seconds after the previous PING, the lease window will be set to 34 seconds.

8.2. Client side monitoring

On the client side, an `org.jboss.remoting.ConnectionValidator` periodically sends a PING message to the server and reports a failure if the response does not arrive within a specified timeout period. The PING is sent on one thread, and another thread determines if the response arrives in time. Separating these two activities allows Remoting to detect a failure regardless of the cause of the failure.

A `ConnectionValidator` is created by a call to one of the overloaded `Client.addConnectionListener()` methods, and since multiple `Clients` may share a `RemoteClientInvoker`, multiple `ConnectionValidators` may be associated with a particular Remoting connection.

The following parameters are supported by `ConnectionValidator`:

validatorPingPeriod - specifies the time, in milliseconds, that elapses between the sending of PING messages to the server. The default value is 2000.

validatorPingTimeout - specifies the time, in milliseconds, allowed for arrival of a response to a PING message. The default value is 1000.

failureDisconnectTimeout - if the parameter "stopLeaseOnFailure" (see Interactions between client side and server side connection monitoring) is set to "true", then "failureDisconnectTimeout" determines the disconnect timeout value to be used by `org.jboss.remoting.LeasePinger` in shutting down. In particular, if "failureDisconnectTimeout" is set to "0", then `LeasePinger` will avoid any network i/o.

NOTE. The default values of "validatorPingPeriod" and "validatorPingTimeout" have often been found in practice to be too small, increasing the likelihood of spurious connection failures.

NOTE. It is important to set "validatorPingPeriod" to a value greater than the value of "validatorPingTimeout". Doing so gives the `ConnectionValidator` a chance to notify all `ConnectionListeners`, which might result in shutting down the connection, before the next PING is sent.

8.3. Interactions between client side and server side connection monitoring

As of Remoting version 2.4, the client side and server side connection monitoring mechanisms can be, and by default are, more closely related, in two ways.

1. If the parameter value **tieToLease** is set to true, then, when the server receives a PING message from an `org.jboss.remoting.ConnectionValidator`, it will return a boolean value that indicates whether a lease currently exists for the connection being monitored. If leasing is activated on the client and server side, then a value of "false" indicates that the lease has failed, and the `ConnectionValidator` will treat a returned value of "false" the same as a timeout; that is, it will notifiy listeners of a connection failure. The default value of this parameter is "true". **Note.** If leasing is not activated on the client side, then this parameter has no effect.

2. If the parameter **stopLeaseOnFailure** is set to true, then, upon detecting a connection failure, `ConnectionValidator` will stop the `LeasePinger`, if any, pinging a lease on the same connection. The default value is "true".

8.4. Client and server identities

Note that by default, a `LeasePinger` has no identity, so if it is replaced by another `LeasePinger` that pings the same `Lease`, the connection remains unchanged. Suppose that leasing is enabled and that a `RemoteClientInvoker` stops and is replaced by a new `RemoteClientInvoker` with a new `LeasePinger`. If the replacement occurs quickly, the server side `Lease` may never miss a PING, in which case there is no evidence that anything changed on the client side. That is, the connection is still alive, as far as the server is concerned. That semantics might be perfectly acceptable for some applications, but other applications might interpret the same events as a connection failure followed by a new connection. In particular, JBoss Messaging needs the latter semantics.

As of release 2.5.2, an important concept related to connection monitoring, **client connection identity**, is available. Remoting can be configured to treat a connection as being defined by a `LeasePinger`/`Lease` pair in which the `LeasePinger` has an identity. More specifically, when configured to do so by setting the parameter **useClientConnectionIdentity** to "true", Remoting identifies a connection with a `LeasePinger`/`Lease` pair in which the `Lease` expects PINGs to arrive from a particular `LeasePinger`.

A `Client` participates in a connection when it is connected by way of the new method

```
public void connect(ConnectionListener listener, Map metadata) throws Exception;
```

This method serves to connect the `Client` to the server by way of a new or existing `RemoteClientInvoker`, and it also registers the new `ConnectionValidator` with the `RemoteClientInvoker`'s `LeasePinger`. Subsequently, if any `ConnectionValidator` registered with that `LeasePinger` detects a connection failure, it will (if "stopLeaseOnFailure" is "true") stop the `LeasePinger`, and the `LeasePinger` will cause each registered `ConnectionValidator` to notify each of its registered `ConnectionListeners` of the connection failure. Once the `LeasePinger` has been shut down and all of the notifications have been made, the connection anchored by the `LeasePinger` is defunct, and the associated `Clients` should be disconnected by a call to `Client.disconnect()`. If such a `Client` is reconnected by a call to `Client.connect()`, it will be associated with a new `LeasePinger` and, therefore, a new connection.

As of release 2.5.3.SP2, Remoting also supports the concept of **server connection identity**. Suppose that a `ServerInvoker` managing a `Lease` stops and is replaced by a new `ServerInvoker` and `Lease`. If the replacement occurs between PINGs from a `ConnectionValidator`, there is no evidence that the server has been replaced. Again, that semantics might be appropriate for some applications, but JBoss Messaging needs a semantics in which the original connection is considered to have been destroyed and replaced. If the parameter **useServerConnectionIdentity** is set to "true", then, when a `ServerInvoker` responds to a PING from a `ConnectionValidator`, it returns a token of its unique identity. If the identity has changed, then `ConnectionValidator` considers the connection to have been broken and it notifies all registered listeners.

9

Configuration files: where are they now?

In the following, $JBOSS_HOME is the root directory of the Application Server installation, and $CONFIG is a server configuration, e.g., default, all, etc.

9.1. EJB 2

- **AS 5, EAP 5, AS6**:
    - $JBOSS_HOME/server/$CONFIG/deploy/remoting-jboss-beans.xml

9.2. EJB 3

- **AS 5, EAP 5, AS6**:
    - $JBOSS_HOME/server/$CONFIG/deploy/ejb3-connectors-jboss-beans.xml

9.3. JBoss Messaging

- **AS 5, EAP 5**:
    - $JBOSS_HOME/server/$CONFIG/deploy/messaging/remoting-bisocket-service.xml
    - also see $JBOSS_HOME/docs/examples/jms
- **AS 6**: uses Hornetq instead of JBoss Messaging