JBoss Remoting

Jeff Haynie
jhaynie@vocalocity.net

Overview

The JBoss Remoting (JBR) framework is a core JBoss component that provides a light-weight building block API for constructing network aware services. The JBR package namespace is `org.jboss.remoting` and part of the `jboss-remoting` CVS module.

The JBR framework provides no particular remoting service alone; however, it does provide a set of interfaces and classes for easily building network aware services. For example, the JMX component uses JBR to provide JMX Remoting services, making MBeans accessible and transportable within a JBoss networked environment.

Concepts

There are several key concepts that are covered throughout this document which are used by the JBR and are important to understand.

- *Transportability*. The framework supports transports which are separated from the actual invocation to make the basic remote infrastructure reusable by a variety of different network transport protocols.

- *Class Loading*. The basic framework handles bi-direction class loading in the case either side lacks the appropriate Java class bytes to handle either making the invocation or receiving the results of an invocation. This is important in a dynamic networked environment, where services are invoked dynamically, sometimes without prior knowledge of the location of the remote server or services.

- *Domains*. The basic framework supports federations (called "domains") of JBoss servers, separated logically by a domain name. Only servers within the same domain will be visible to each other.

- *Discovery*. The basic framework supports discovering JBoss servers within a particular domain, without prior knowledge of their location, transport or services available to other servers in the same network.

The JBR provides a lightweight framework that has no specific knowledge of the specific contents of remote invocations, except that it builds the invocations, marshals and unmarshals their serialized contents, and transports them to and from their destinations. This provides the maximum amount of flexibility for building network aware services, without services having to worry about the specific transports and serialization schemes used by the framework.

Main Participants

Identity

The `Identity` object provides a unique location of a particular JBoss server on the network and contains 4 main subparts:

1. *InstanceID* – the instance ID is a globally unique identifier, which is the same among reboots of the same JBoss server instance.

2. *JMX Server ID* – the JBoss JMX MBeanServer ID, which is calculated each time the JBoss server instance is started. The ID is a globally unique identifier.

3. *Domain* – the logical name of the federation which the server is part of.

4. *IPAddress* – the IP Address of the server.

The Identity InstanceID is set normally by reading the file, `jboss.identity`, in the `<jboss.home>/server/<servername>/data` directory. If the file is not present, it will be created the first time and the ID will be written to this file. Additionally, the system property `jboss.identity` will be set to this value.

You can programmatically set this ID by setting the system property `jboss.identity` to any unique ID on startup, using the `-Djboss.identity=<value>` syntax. *WARNING: If two or more servers on the network use identical identity values, the results will be indeterminate.*

The Identity domain is set normally to the value, `JBOSS`. However, this value can be set programmatically by setting the system property `jboss.identity.domain`, using the `-Djboss.identity.domain=<value>` syntax.

The `Identity` class is `Serializable`, and thus can be passed remotely across the wire or stored to a file. Additionally, the Identity is used by Detectors when sending and receiving detection events.

InvokerLocator

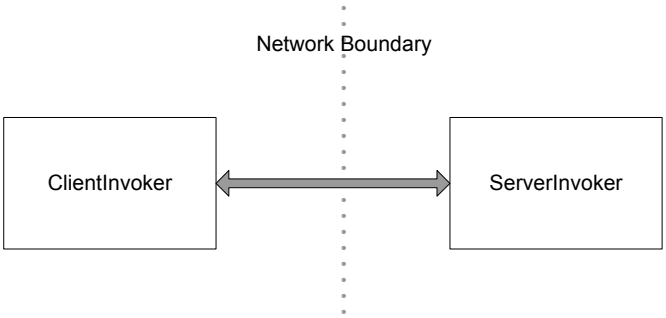
The `InvokerLocator` is a class, which can be described as a string URI, for describing a particular JBoss server JVM and transport protocol. For example, the `InvokerLocator` string `socket://192.168.10.1:8080` describes a TCP/IP Socket-based transport, which is listening on port 8080 of the IP address, 192.168.10.1. Using the string URI, or the `InvokerLocator` object, the JBR 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>
```

Invokers

Invokers are the transport subsystems that allow a method invocation to be transported back and forth between client and server in a JBR network. The `ClientInvoker` will transport a local method to a remote `ServerInvoker`, which will invoke the method on the remote JVM and transport the result, or exception, back to the `ClientInvoker` for delivery.

Both the `ClientInvoker` and `ServerInvoker` are abstract classes, and extend the abstract class, `AbstractInvoker`. Each transport subsystem extends both the `ClientInvoker` and `ServerInvoker` abstract classes for the particular protocols.



InvokerRegistry

The `InvokerRegistry` is a singleton that keeps references to both client and server `Invokers` that are available locally so that they can be reused for multiple clients.

Subsystem

A Subsystem describes a service that uses the JBR for remote invocations. The Subsystem will access the remote server by creating a `Client` object and passing the `InvokerLocator` and Subsystem string value. Once the `Client` is created, invocations can be made remotely by calling `invoke`. A subsystem, for example, is JMX or AOP.

Connector

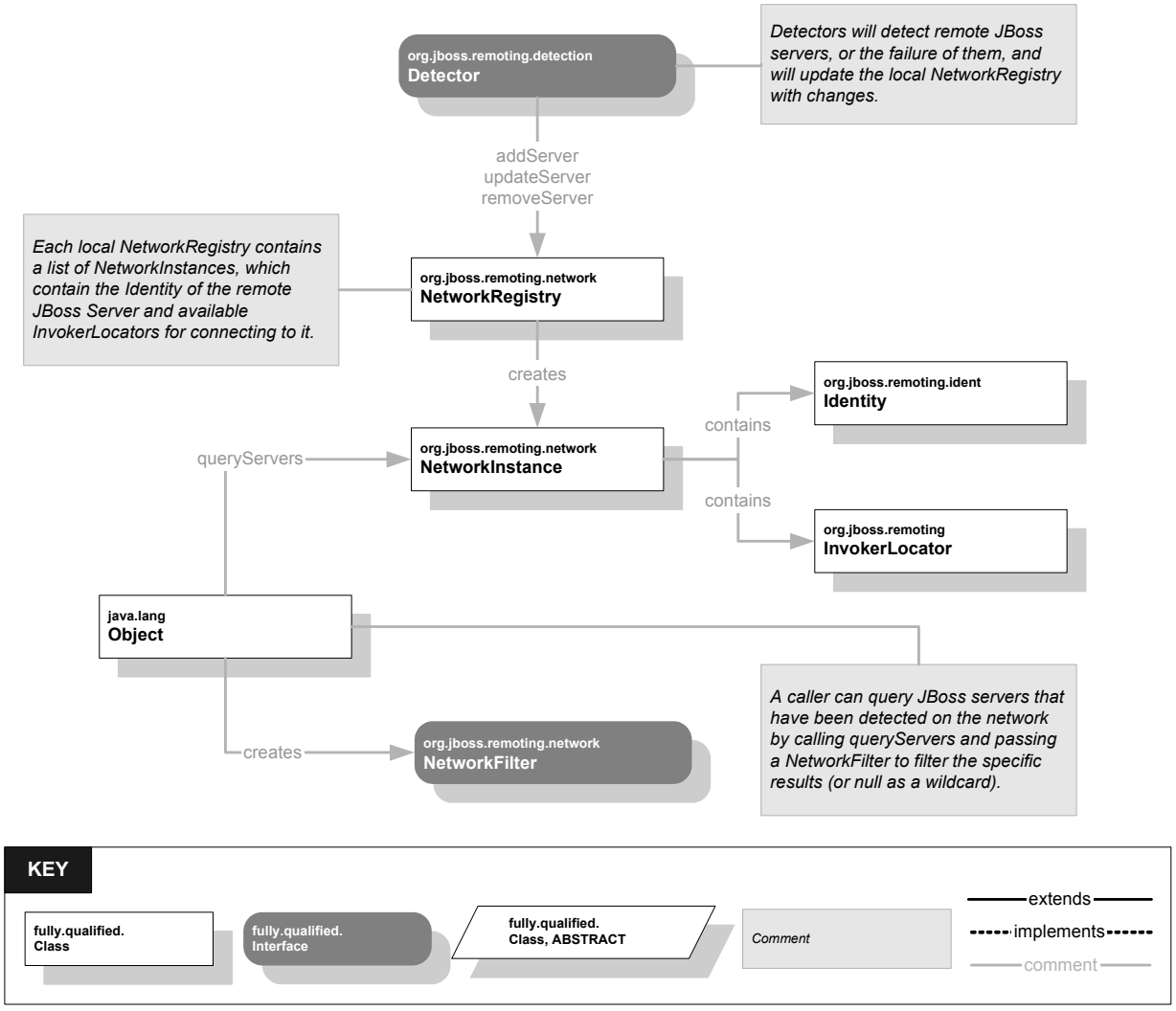
A `Connector` is an MBean that loads a particular `ServerInvoker` implementation for a given transport subsystem and one or more `ServerInvocationHandler` implementations that handle Subsystem invocations on the remote server JVM. There is exactly one `Connector` per transport type.

Detector

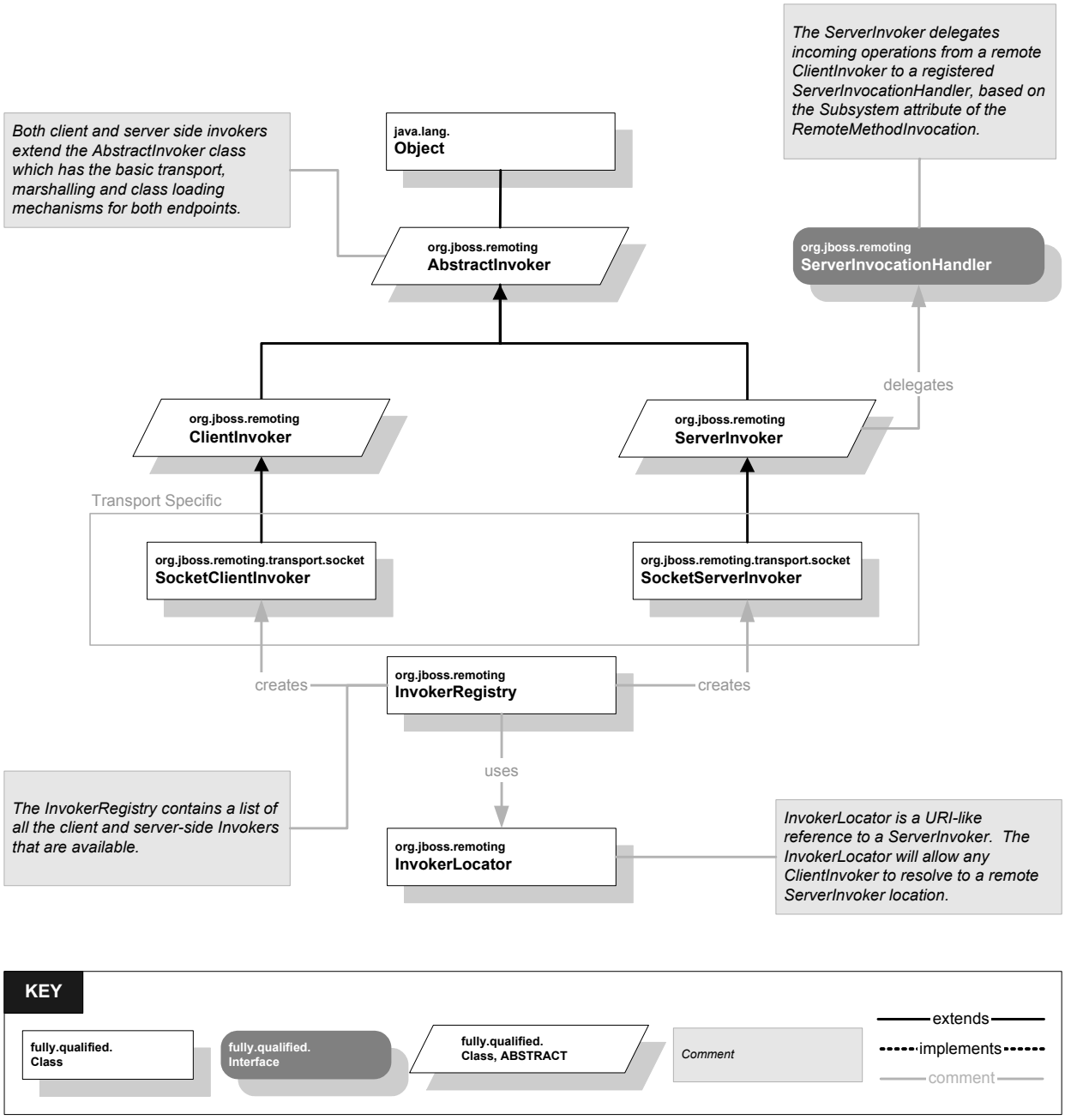
A `Detector` is an interface which is implemented for each detection mechanism. Multiple detectors can be used at the same time for discovering JBoss server peers within the same JBoss domain.

NetworkRegistry

An MBean service that provides a full network map of all the JBoss servers within the same JBoss domain. The `NetworkRegistry` can be queried to find servers based on a particular `NetworkFilter`, or a listener can be added to the `NetworkRegistry` to receive JMX Notifications when servers are found or lost.



JBR Class Diagram using the Socket transport



Transports

Transports provide the ability to move invocations across the wire in a protocol-specific way. For example, a Socket transport will read and write data to a TCP/IP Socket stream. SOAP transports build XML based representations of the binary invocation buffer and send via a SOAP protocol transport, usually HTTP.

The JBR framework provides the following transports out-of-the-box:

- *Socket* – the socket transport uses the `ServerInvoker`, `SocketServerInvoker`, and the `ClientInvoker`, `SocketClientInvoker`, in the package `org.jboss.remoting.transport.socket`.

- *RMI* – the RMI transport uses the `ServerInvoker`, `RMIServerInvoker`, and the `ClientInvoker`, `RMIClientInvoker`, in the package `org.jboss.remoting.transport.rmi`.

- *SOAP* – the SOAP transport uses the `ServerInvoker`, `SOAPServerInvoker`, and the `ClientInvoker`, `SOAPClientInvoker`, in the package `org.jboss.remoting.transport.soap.axis`.

More transports will be provided as the JBR framework evolves.

To write a new transport, you must subclass the `ClientInvoker` and `ServerInvoker` classes. Once the invokers are subclassed for each side, the `InvokerRegistry` should be called to register the invoker transport, by calling

```
InvokerRegistry.registerInvoker (String, Class, Class)
```

with the name of the transport, and both Client and Server invoker implementation classes.

[JGH NOTE: We provide should provide a serializable InvokerResolver class with each invoker transport that will assist a JVM to resolve and install a new transport if not registered in the remote InvokerRegistry]

Detectors

Detectors provide the ability to discover remote JBoss servers that are part of the same JBoss domain. Detectors will register remote JBoss server Identities and available `InvokerLocators` used to connect to the remote server.

The JBR provides the following detectors out-of-the-box:

- *Multicast* – the Multicast detector uses a multicast socket/port for detecting JBoss servers within the same domain.

- *JNDI* – the JNDI detectors browses a JNDI context for detecting JBoss servers within the same domain.

More detectors will be provided as the JBR framework evolves.

To write a new Detector, you should subclass the `AbstractDetector` class, or implement the `Detector` interface.

Class Loading

The key part to any network aware service is having the ability to transport objects to and from remote locations. However, because Java uses a set of classes to describe an objects implementation structure and instance data, the class must be accessible in the JVM before loading the object. This is true also when unmarshalling an object from a serialized buffer. However, let's say the class is not available locally, because it is an anonymous inner class created on-the-fly and passed as a parameter to a remote method. For this type of behavior to be supported, the class must be loaded dynamically in the remote JVM.

In certain RPC frameworks, such as RMI, dynamic classloading of client classes is achieved by downloading the remote classes using codebase annotation. For RMI, each time a class is serialized, it contains the codebase (which is controlled by a System property, or dynamically set by a smart `ClassLoader`) URL of a webserver that can service classes remotely.

For JBR, class loading is supported bi-directionally and by the specific transport servicing the request. For example, a SOAP transport cannot support codebase signing like the native RMI system does, since SOAP server endpoints cannot connect back to their client endpoints. This requires an alternative mechanism for servicing classes for servers and clients.

Additionally, clients may receive objects back from a remote invocation, in which the class that the caller is receiving is not available locally. This is also supported by the JBR framework.

Bi-directional class loading is accomplished by following these steps:

1. The caller invokes the `ClientInvoker` (using the subclassed `ClientInvoker`) invoke method, which causes the ClientInvoker to create a `RemoteMethodInvocation` instance representing the serialized method call and arguments. The instance also contains an array of Strings of each parameter's class name.

2. In a while loop, the `ClientInvoker` marshals the `RemoteMethodInvocation` object into a byte array and calls the abstract `transport` method, which is implemented by

the subclass specific transport. Each parameter is individually wrapped in a `ClassBytes` wrapper, which contains the name of the parameter class and the byte array of the serialized data buffer.

3. On the Server side, the ServerInvoker will attempt to de-serialize each parameter of the `RemoteMethodInvocation` object one at a time. If the serialization fails with a `ClassNotFoundException`, the class name will be passed to an instance of the `ClassRequestedMethodInvocationResult` class and returned as a serialized buffer. If the de-serialization completes successfully, the invocation object will be wrapped in an `InvocationRequest` object and passed to the appropriate `ServerInvocationHandler.invoke()` method. The invoke returns either a result or raises a Throwable exception. The ServerInvoker serializes the exception or result in a `RemoteMethodInvocationResult` object and returns the serialized buffer.

4. The transport returns a byte array, which contains the result of the invocation, in the `RemoteMethodInvocationResult` object. The RemoteMethodInvocationResult object contains the result as a `ClassBytes` object.

5. The client attempts to deserialize the `RemoteMethodInvocationResult` byte array:

    a. If the deserialization fails, the client will invoke (within the same while loop) back to the server passing an instance of the `ClassRequiredMethodInvocation` class with the name of the class that was not found locally.

    b. If the deserialization succeeds, and the result contains a result object or null, `ClientInvoker` returns the value to the caller (breaking out of the while loop).

    c. If the deserialization succeeds, and the result contains an exception, the exception will be thrown (breaking out of the while loop).

    d. If the deserialization succeeds, and the result is a `ClassRequestedMethodInvocationResult` class, the class will be loaded by the `ClassByteClassLoader` into a `ClassBytes` instance. The while loop will continue, and will add the ClassBytes as a parameter to the `RemoteMethodInvocation` and will be re-invoked (within the same calling context). *This will continue until a, b or c above is met.*

If either side receives class bytes from a remote location, the class bytes are added to the `ClassBytesClassLoader` so that subsequent invocations will have the class locally. However, these classes are not persisted after a shutdown and are deleted when the JVM exits.

AOP Remoting

The AOP framework can remote any POJO and pass that POJO across the wire to another JBoss server, maintaining the Interceptor chain and the instrumented class bytes.

A specialized `ServerInvocationHandler`, `AOPRemotingInvocationHandler`, receives the remote invocation and dispatches the invocation via the `Dispatcher` singleton. The `Dispatcher` will invoke the object using the AOP Interceptor framework.

A remote class proxy can be created dynamically by calling:

```
Remoting.createRemoteProxy (Object oid, Class clazz, InvokerLocator
                            locator);
```

The Remoting class is in the `org.jboss.aop.remoting` package and returns a `ClassProxy` instance.

[JGH: need to do more documentation]

JMX Remoting

[JGH: TODO]

JMS Remoting

[JGH: TODO]