RMI Protocol

Stefan Wehr

June 14, 2006

Note: The following text is more or less directly copied from Section 10 of the "Java RMI Specification" (`http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html`)

The RMI protocol makes use of two other protocols for its on-the-wire format: Java Object Serialization and HTTP. The Object Serialization protocol is used to marshal call and return data. The HTTP protocol is used to "POST" a remote method invocation and obtain return data when circumstances warrant.

1 Serialization

Call and return data in RMI calls are formatted using the Java Object Serialization protocol. Each method invocation's CallData is written to a Java object output stream that contains the ObjectIdentifier (the target of the call), an Operation (a number representing the method to be invoked), a Hash (a number that verifies that client stub and remote object skeleton use the same stub protocol), followed by a list of zero or more Arguments for the call.

To enable class loading from remote hosts, RMI overrides the `annotateClass` and `resolveClass` methods of `ObjectOutputStream` and `ObjectInputStream`, respectively. The method `annotateClass` of `ObjectOutputStream` is called when a class is serialized. RMI uses this method to store the URL from which the class can be loaded in the serialized data.

When a class is deserialized, the `resolveClass` method of `ObjectInputStream` is called. RMI uses this method to retrieve the URL from which the class can be loaded.

2 Multiplexing

The purpose of multiplexing is to provide a model where two endpoints can each open multiple full duplex connections to the other endpoint in an environment where only one of the endpoints is able to open such a bidirectional connection using some other facility (e.g., a TCP connection). RMI use this simple multiplexing protocol to allow a client to connect to an RMI server object in

some situations where that is otherwise not possible. For example, some security managers for applet environments disallow the creation of server sockets to listen for incoming connections, thereby preventing such applets to export RMI objects and service remote calls from direct socket connections. If the applet can open a normal socket connection to its codebase host, however, then it can use the multiplexing protocol over that connection to allow the codebase host to invoke methods on RMI objects exported by the applet.

The multiplexing protocol facilitates the use of virtual connections, which are themselves bidirectional, reliable byte streams, representing a particular session between two endpoints. The set of virtual connections between two endpoints over a single concrete connection comprises a multiplexed connection.

2.1 Flow Control

A simple packeting flow control mechanism is used to allow multiple virtual connections to exist in parallel over the same concrete connection. The high level requirement of the flow control mechanism is that the state of all virtual connections is independent; the state of one connection may not affect the behavior of others. For example, if the data buffers handling data coming in from one connection become full, this cannot prevent the transmission and processing of data for any other connection. This is necessary if the continuation of one connection is dependent on the completion of the use of another connection, such as would happen with recursive RMI calls. Therefore, the practical implication is that the implementation must always be able to consume and process all of the multiplexing protocol data ready for input on the concrete connection (assuming that it conforms to this specification).

Each endpoint has two state values associated with each connection: how many bytes of data the endpoint has requested but not received (input request count) and how many bytes the other endpoint has requested but have not been supplied by this endpoint (output request count).

An endpoint's output request count is increased when it receives a REQUEST operation from the other endpoint, and it is decreased when it sends a TRANSMIT operation. An endpoint's input request count is increased when it sends a REQUEST operation, and it is decreased when it receives a TRANSMIT operation. It is a protocol violation if either of these values becomes negative.

It is a protocol violation for an endpoint to send a REQUEST operation that would increase its input request count to more bytes that it can currently handle without blocking. It should, however, make sure that its input request count is greater than zero if the user of the connection is waiting to read data.

It is a protocol violation for an endpoint to send a TRANSMIT operation containing more bytes than its output request count. It may buffer outgoing data until the user of the connection requests that data written to the connection be explicitly flushed. If data must be sent over the connection, however, by either an explicit flush or because the implementation's output buffers are full, then the user of the connection may be blocked until sufficient TRANSMIT operations can proceed.