The Multiplex Subsystem of the JBoss Remoting Project

Ron Sigal

July 4, 2006

1. Introduction.

The Multiplex subsystem of the JBoss Remoting Project (referred to herein on occasion simply as “Muliplex”) supports the multiplexing of multiple data streams over a single network connection, based on a reimplementation of the following classes from `java.net`:

1. `Socket`
2. `ServerSocket`
3. `SocketInputStream`
4. `SocketOutputStream`

and the following classes from `javax.net`:

1. `SocketFactory`
2. `ServerSocketFactory`

It is motivated by circumstances in which the number of available ports on a system is restricted by a firewall or other considerations. Since the Remoting project is the principal client of Multiplex, we illustrate multiplexing primarily in the context of a Remoting application. Remoting supports two modes of client-server communication: (1) method calls from client to server, with a synchronous response, and (2) client requests for an asynchronous callback from the server. The usual need for separate ports to support both synchronous and asynchronous modes is obviated by the Multiplexing subsystem.

2. The Prime Scenario.

The typical application of multiplexing in the Remoting context is illustrated by the **Prime Scenario**, in which a client requiring both synchronous and asynchronous responses from a server is behind a firewall and has only a single port at its disposal. Without the restriction to a single port, we would have the situation in Figure 1, which requires no multiplexing. With the restriction, we have the Prime Scenario, as in Figure 2.

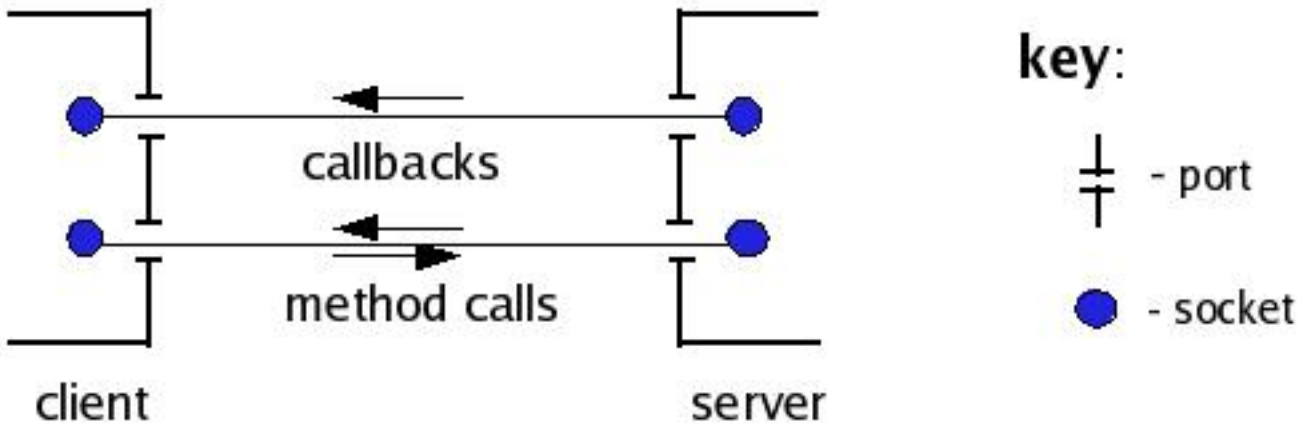

Figure 1. Method calls and callbacks with no port restrictions.

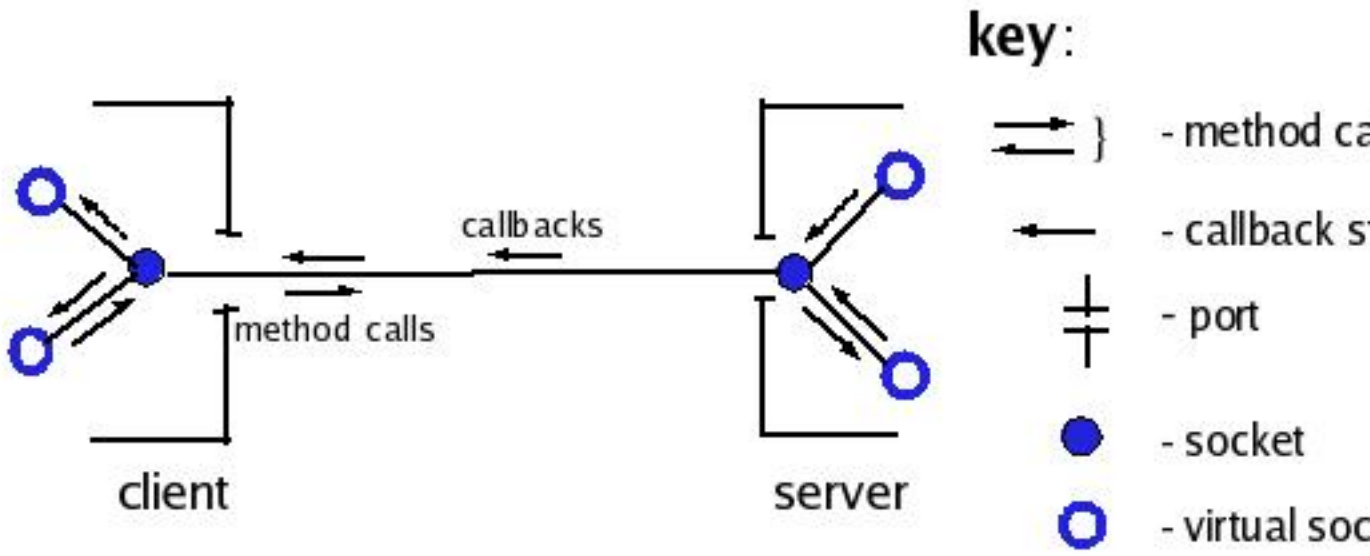

Figure 2. Method calls and callbacks in the Prime Scenario.

Multiplexing is supported primarily by the concept of the **virtual socket**, implemented by the `VirtualSocket` class. `VirtualSocket` is a subclass of `java.io.Socket`, and supports the full socket API. As is the case with actual sockets, virtual sockets are created in one of two ways:

1. a constructor (or factory) call on a client, or
2. a call to the `accept()` method of a server socket on a server.

Accordingly, the other principal Multiplex concept is the **virtual server socket**, implemented by two classes:

1. `MasterServerSocket`, and

2. `VirtualServerSocket`.

These are both subclasses of `java.io.ServerSocket`, and both implement the full server socket API. Since virtual sockets are implemented on the foundation of actual sockets, and the creation of actual sockets requires a server socket, we need the support of actual server sockets in the creation of virtual sockets. It is the role of `MasterServerSocket` to provide that support. The `accept()` method of `MasterServerSocket` calls `super.accept()` to create an actual socket which is then wrapped in a mechanism which supports one or more virtual sockets. Every Muliplex application requires at least one `MasterServerSocket`, and the Prime Scenario requires exactly one. Figure 3 illustrates the process in which a virtual socket *v1* connects to a `MasterServerSocket`, which creates and returns a reference to a new virtual socket *v2*.

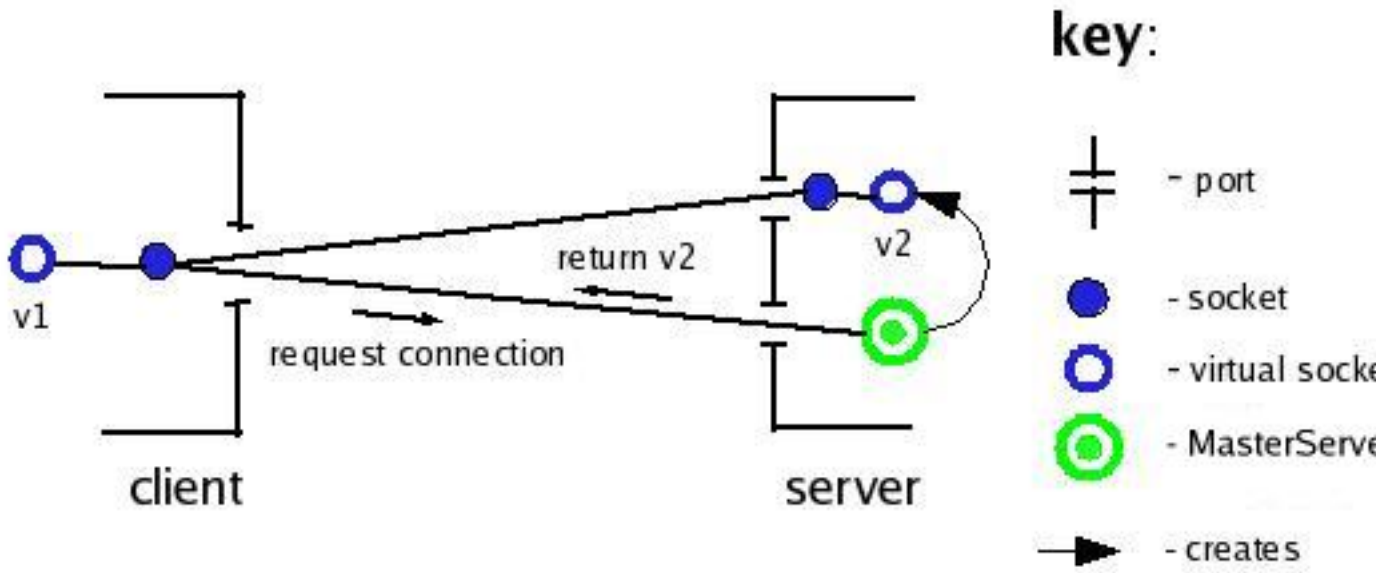

Figure 3. Setting up a synchronous connection.

In Figure 3 we have a connection between *v1* and *v2*, which can support synchronous communication but which offers nothing not provided by actual sockets. The support of multiplexed callbacks, however, requires the use of the other virtual server socket class, `VirtualServerSocket`. Unlike `MasterServerSocket`, `VirtualServerSocket` does not depend on superclass facilities, but rather it uses an ordinary client socket, with which implements its own version of the `accept()` method, able to create any number of virtual sockets, all of which share a single port with the `VirtualServerSocket`. It is important to understand how its use of an actual socket determines the nature of a `VirtualServerSocket`. Unlike a server socket, a client socket must be connected to another socket to function, and a `VirtualServerSocket` has the same property. It follows that a `VirtualServerSocket` can process requests from just one host, the host to which its actual socket is connected.

The role of the `VirtualServerSocket` is illustrated in Figure 4. A constructor (or factory method, which calls a constructor) is called on the server to create virtual socket *v3* to support callbacks. The constructor sends a connection request to the `VirtualServerSocket` on the client, which creates new virtual socket *v4* and sends back to *v3* a reference to *v4*. At this point the Prime Scenario is set up.

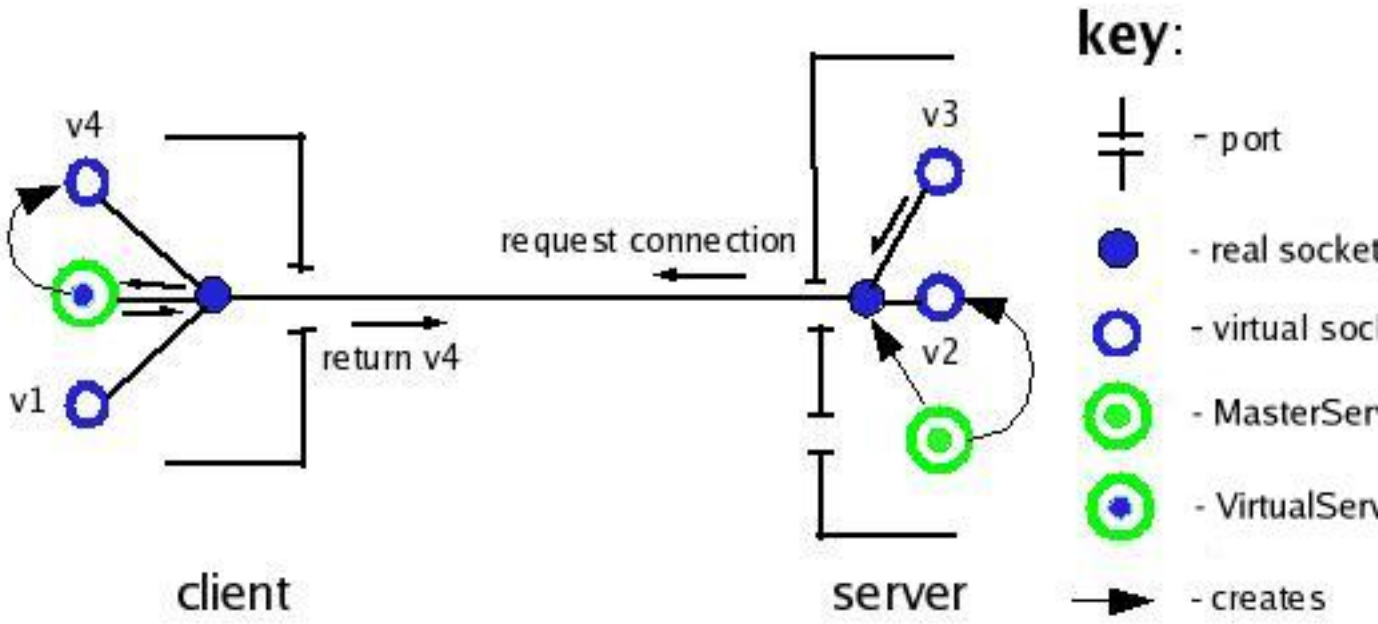

Figure 4. Adding an asynchronous connection to Figure 3.

3. Virtual socket groups.

In order to understand the creation of structures like the Prime Scenario and others described below, it is important to understand the concept of a **virtual socket group**. A virtual socket group is a set of virtual sockets, and zero or one `VirtualServerSockets`, sharing a single actual socket. We say that the socket group is *based on* its actual socket. Depending on the state of its underlying actual socket and the nature of its peer socket group, if any, a socket group may be in one of three states. Let *G* be a socket group based on actual socket *S*. Then *G* may be

1. **bound**: *S* is bound but not connected, or

2. **connected**: *S* is connected to socket *S′* and the socket group based on *S′* does not contain a `VirtualServerSocket`, or

3. **joinable**: *S* is connected to socket *S′* and the socket group based on *S′* does contain a `VirtualServerSocket`.

Although it is possible for a socket to be neither bound nor connected, we do not consider a socket group to exist until its underlying socket is at least bound to a local address. A connected or joinable socket group is said to be **visible**, and a bound socket group is **invisible**. A socket group is characterized by the pair of addresses

(*localAddress, remoteAddress*)

where these are the local and remote addresses of the actual socket underlying the socket group. *localAddress* may take the special form (*, *port*), where the wildcard value “*” denotes any hostname by which the local host is known.

Depending on the state of the socket group, *remoteAddress* may have the special value *undefined*, indicating that a connection has not yet been established.

There are two ways of creating a new virtual socket group or of joining an existing socket group: through a **binding action** or a **connecting action**. A binding action is either

1. a call to any of the `VirtualServerSocket` constructors other than the default constructor (i.e., those with a port parameter), or
2. a call to a `bind()` method in `VirtualSocket` or `VirtualServerSocket`.

A connecting action belongs to one of five categories:

1. a call to any `VirtualSocket` or `VirtualServerSocket` constructor that requires a remote address (note that unlike `java.net.ServerSocket`, `VirtualServerSocket` has a such a constructor),
2. a call to a `connect()` method (again, `VirtualServerSocket` has a nonstandard `connect()` method),
3. a call to `VirtualServerSocket.accept()`,
4. a call to `MasterServerSocket.accept()`, or
5. a call to `MasterServerSocket.acceptServerSocketConnection()`.

Each binding action has an associated local address, and each connecting action has an associated remote address and an optional local address. For binding actions, and connecting actions in the first two categories, the addresses are given explicitly in the method call. For a call to `VirtualServerSocket.accept()`, the addresses are those of the socket group to which the server socket belongs, and for the two `MasterServerSocket` methods, the addresses are those of the actual socket they create.

Depending on their associated local and remote addresses and on the socket groups that exist at the time of the action, a binding or connecting action may have the effect of creating a new socket group or adding a new member to an existing socket group. The rules are straightforward, but there is one source of possible confusion, the accidental connection problem discussed below, that must be guarded against. Let *V* be a virtual socket or virtual server socket undergoing either a binding or connecting action.

1. **binding action rule**: If there are visible socket groups whose local address matches the action's local address, then *V* joins one of them chosen at random. Otherwise, a new bound socket group is created and *V* joins it.
2. **connecting action rule**:
    a. For actions in the first two categories, where *V* is a `VirtualSocket` (respectively, a `VirtualServerSocket`):
        i. If the action has a remote address but no local address:
            A. If there are any joinable (resp., connected) socket groups with a matching remote address, then *V* joins one of them chosen at random.
            B. If there are no such socket groups, an attempt is made to connect to a `MasterServerSocket` at the remote address, and if the attempt succeeds, a new socket group is created and *V* joins it.
        ii. If the action has both a local address and a remote address:

A. If there is a joinable (resp., connected) socket group with matching addresses, then *V* joins it

B. Otherwise, if the local address (in particular, its port) is currently in use, the action results in a `IOException`.

C. Otherwise, a new socket group *G* is created and bound to the local address. Then an attempt is made to connect to a `MasterServerSocket` at the remote address, and if the attempt succeeds, *V* joins *G*.

b. For `VirtualServerSocket.accept()` calls, the new virtual socket joins the socket group to which the server socket belongs.

c. For `MasterServerSocket.accept()` calls, a new socket group is created with the new virtual socket as its first member.

d. For `MasterServerSocket.acceptServerSocketConnection()` calls, a new socket group with zero members is created.

NOTES:

1. A bound socket group is inaccessible to the connect action rules (which is why it is called "invisible"). The reason is to avoid a situation in which one virtual socket "highjacks" another virtual socket's group. Suppose that virtual socket *v1* binds itself to ("localhost", 5555), but before it gets a chance to connect to ("www.jboss.com", 6666), virtual socket *v2* binds to ("localhost", 5555) and then connects to ("www.ibm.com", 7777). Then when *v1* tries to connect to ("www.jboss.com", 6666), the attempt fails. This situation cannot occur because at the moment when *v2* does its bind, *v1*'s socket group is invisible and *v2* is forced to create it own socket group.

2. The connecting action rules are different for `VirtualSocket` and `VirtualServerSocket` (specifically, the former can join only joinable socket groups, while the later can join connected socket groups) because `VirtualSocket` needs a `VirtualServerSocket` to create a peer virtual socket for it to connect to, and a `VirtualServerSocket` does not need such a peer.

3. **N.B.** It is important to understand a possible side effect of a binding action. When *V* joins a socket group through a binding action, it is possible that the group is already connected. In this case, a subsequent connecting action (in particular, a call to `connect()`) to any address other than the socket group's remote address is invalid, leading to an `IOException` with the message "socket is already connected.". This is called the **accidental connection problem**, and it is avoidable. Both `VirtualSocket` and `VirtualServerSocket` have constructors and nonstandard versions of the `connect()` which accept both local and remote addresses. These treat binding and connecting as a single atomic process.

The socket group rules are illustrated in the following two sections.

4. Coding the Prime Scenario.

In order to set up the Prime Scenario, the following steps are necessary (the socket names conform to Figure 4):

1. On the server, create a `MasterServerSocket` and bind it to port *P*.

2. On the client, create a virtual socket *v1* and connect it to port *P*.

3. Let *Q* be the port on the client to which *v1* is bound. Create a `VirtualServerSocket` on the client, bind it to *Q*, and connect it to *P*.

4. On the server, create a virtual socket *v3* and connect it to port *Q*.

The Prime Scenario provides an example of creating socket groups. In step 2, a socket group *G1* is created on the client through the construction of *v1*. It enters the connected state, bound to an arbitrary port *Q* on the client and connected to port *P* on the server. In step 3 a `VirtualServerSocket` joins *G1* by way of binding to *Q* on the client and connecting to *P* on the server. In fact, the socket group rules imply that it is enough to bind the server socket to port *Q*. Connecting it to *P* on the server occurs as a side effect of the binding action. Finally, step 4 adds virtual socket *v4* to *G1*. While *G1* is being built on the client, a socket group *G2* is being built on the server. Step 2 results in the creation of *G2*, along with its first member, a new virtual socket, *v2*, returned by the `accept()` method of the `MasterServerSocket`. Step 4 adds a second member, *v3*, to *G2*.

See Listing 1 and Listing 2 for a simple example of coding these steps. Variants of these samples may be found in the directory /org/jboss/remoting/samples/multiplex.

5. More general scenarios.

Although Multiplex was motivated by the Prime Scenario, it can also support other connection structures. We describe two alternatives in this section.

5.1. The *N*-socket scenario.

The **N-socket scenario** demonstrates that a socket group is not restricted to just two virtual sockets. It also demonstrates that a `VirtualServerSocket` does not depend on the prior existence of a connected virtual socket. As long as it has access to a `MasterServerSocket` ready to accept a connection, it can get started. In fact, the `MasterServerSocket.accept()` method will silently accept a connection from a `VirtualServerSocket` while it is waiting for a connection request from a virtual socket, but the `acceptServerSocketConnection()` method is designed specifically to accept a connection request from a `VirtualServerSocket`.

The connection structure of the *N*-socket scenario is depicted in Figure 5 (for $N = 3$), and the code for a simple client and server is given in Listing 3 and Listing 4. In the example a socket group with 3 elements is constructed on the server. It is created with the call

```
serverSocket.acceptServerSocketConnection()
```

which creates an actual socket and a socket group which, though it has no members, is connected to a `VirtualServerSocket` on the client. The next three lines,

```
Socket socket1 = new VirtualSocket(“localhost”, 5555);
Socket socket2 = new VirtualSocket(“localhost”, 5555);
Socket socket3 = new VirtualSocket(“localhost”, 5555);
```

```
```

populate the socket group with three virtual sockets. On the client there is a socket group with four members, first created with the call

```
serverSocket.connect(connectAddress);
```

and then further populated by the three subsequent lines

```
Socket socket1 = serverSocket.accept();
Socket socket2 = serverSocket.accept();
Socket socket3 = serverSocket.accept();
```

Variants of the *N*-Socket Scenario client and server may be found in the directory /org/jboss/remoting/samples/multiplex.

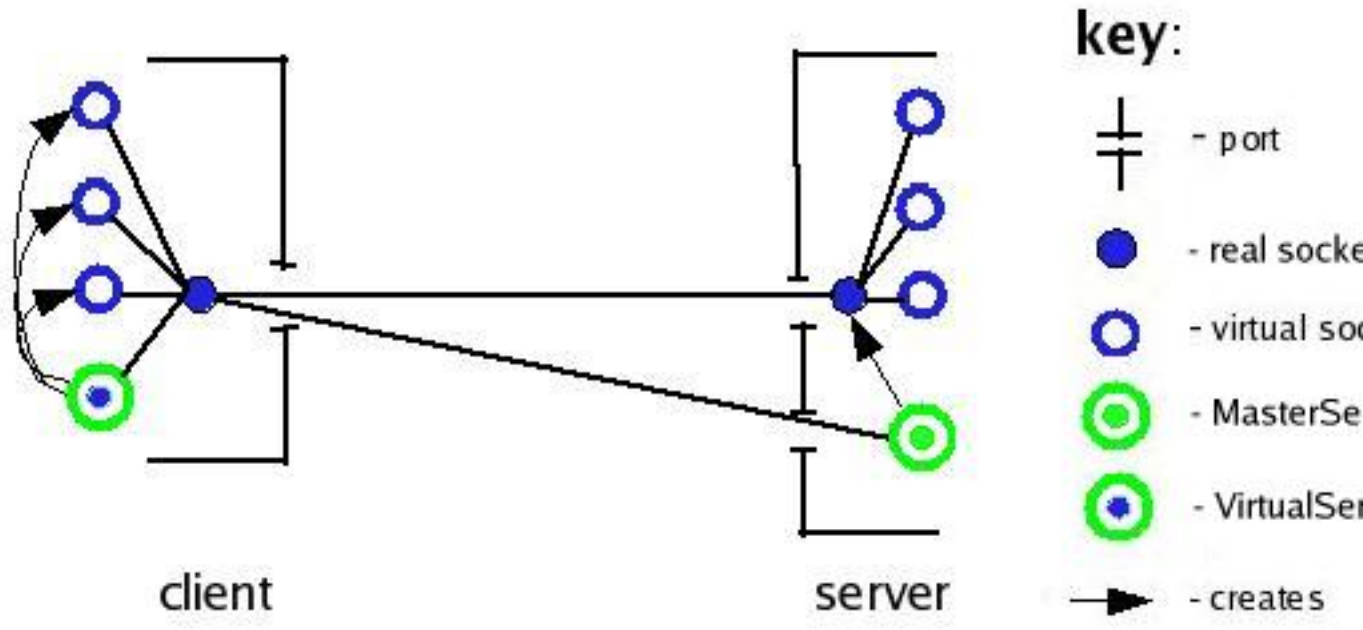

Figure 5. The connection structure in the *N*-Socket Scenario.

5.2. The Symmetric Scenario.

The connection structure in the **Symmetric Scenario** consists of socket groups on two hosts, each of which contains a `VirtualServerSocket` and some number of virtual sockets. The scenario is not truly symmetric, since each connection

structure has to begin with a connection request to a `MasterServerSocket`, but once that happens the “client” and “server” are identical, as depicted in Figure 6d. Once the line

```
serverSocket.connect(address);
```

on the client (see Listing 5) and the line

```
int port = mss.acceptServerSocketConnection();
```

on the server (see Listing 6) are executed, the client has a socket group characterized by the address pair

((*, 5555), (“localhost“, 7777))

and consisting of a `VirtualServerSocket`, and the server has a socket group with zero members characterized by the address pair

((“localhost“, 7777), (“localhost”, 5555)).

(See Figure 6a.) And once the line

```
vss.connect(address);
```

is executed on the server, the new `VirtualServerSocket` joins the server's socket group, as shown in Figure 6b. After the lines

```
Socket virtualSocket1 = new VirtualSocket(“localhost”, port);
```

and

```
Socket virtualSocket1 = vss.accept();
```

are executed on the client and server, respectively, each socket group has a new virtual socket (see Figure 6c), and finally, after the lines

```
Socket virtualSocket2 = new VirtualSocket(“localhost”, 5555);
```

and

```
Socket virtualSocket2 = serverSocket.accept();
```

are executed on the server and client, respectively, each socket group has a second virtual socket (see Figure 6d).

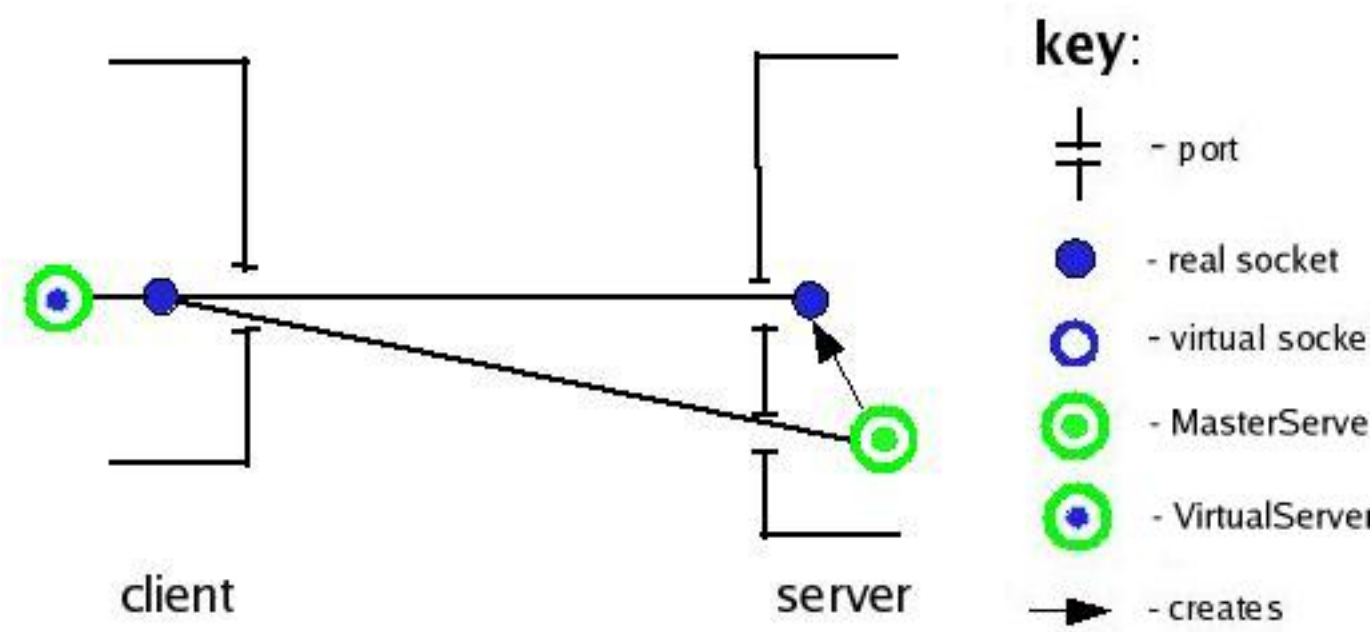

Figure 6a. The connection structure in the Symmetric Scenario: stage 1.

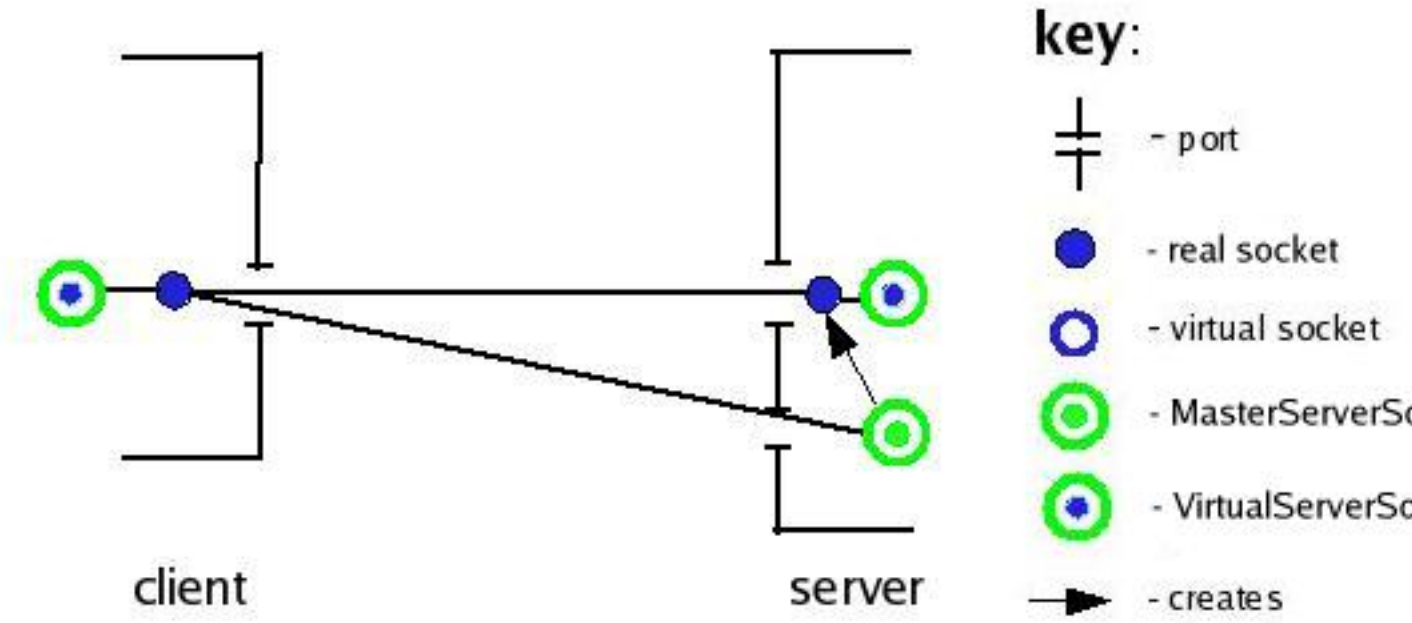

Figure 6b. The connection structure in the Symmetric Scenario: stage 2.

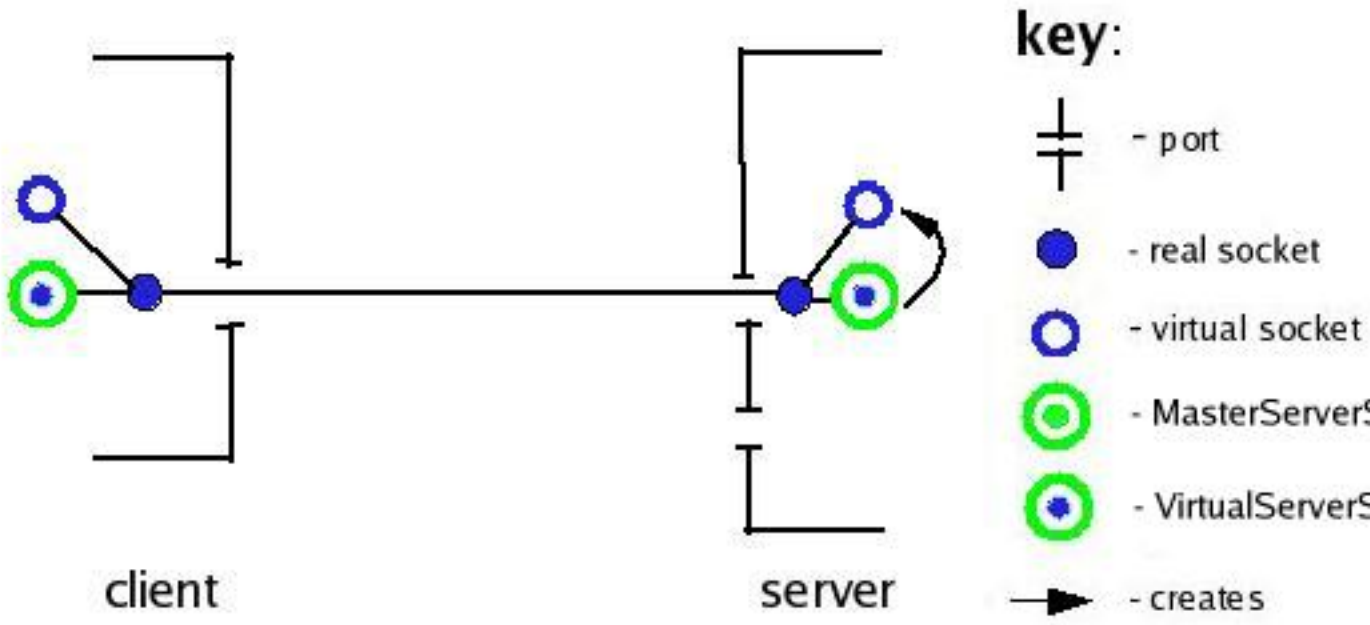

Figure 6c. The connection structure in the Symmetric Scenario: stage 3.

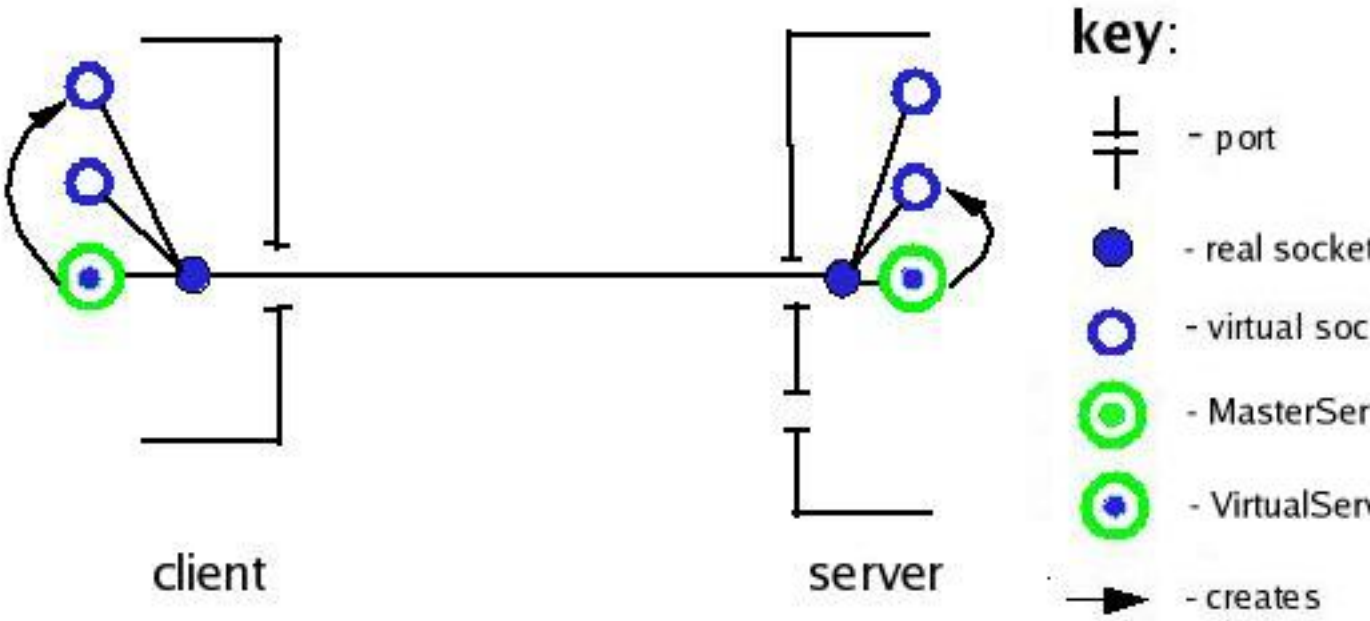

Figure 6d. The connection structure in the Symmetric Scenario: stage 4.

6. Factories.

In addition to virtual sockets and virtual server sockets, Multiplex also implements the two factories associated with sockets: the socket factory and the server socket factory. `VirtualSocketFactory` extends `javax.net.SocketFactory` and reimplements all of its methods. `VirtualServerSocketFactory` extends `javax.net.ServerSocketFactory` and reimplements all of its methods (though the backlog parameter is ignored). These two classes make it possible for a section of code to be completely unaware that it is using virtual sockets instead of actual sockets. The only configuration involved in the use of these factories is the need to tell `VirtualServerSocketFactory` whether it is running on a client or a server, which tells it whether to create `VirtualServerSockets` or `MasterServerSockets`, respectively. That notification is performed by the methods `setOnClient()` and `setOnServer()`. See Listing 7 for an illustration of the idiomatic use of these classes, where the method `useFactories()` refers only to the parent classes `SocketFactory` and `ServerSocketFactory`.

7. Configuration.

The Multiplex system may be used without any external configuration, but it exposes several parameters which may be set to adjust its behavior, and possibly performance. They affect the following classes:

MultiplexingManager:	The central Multiplex class, `MultiplexingManager` wraps a real socket. It is responsible for creating an environment, including multiple threads, which allow a single socket to be shared by multiple streams of communication. Note that the `MultiplexingManager`

	is the implementation of the concept of "virtual socket group." A virtual socket group is supported by exactly one `MultiplexingManager`, and each `MultiplexingManager` supports exactly one virtual socket group.
OutputMultiplexor:	`OutputMultiplexor` has two roles. (1) It is called by `MultiplexingOutputStream` to queue an array of bytes to be sent to a virtual socket at the other end of a connection. (2) It contains the inner class `OutputThread`, which takes byte arrays from the queue and writes them, along with appropriate header information, to the actual socket.
InputMultiplexor:	`InputMultiplexor` contains two inner classes, `MultiGroupInputThread` and `SingleGroupInputThread`, which are responsible for demultiplexing the virtual streams on the actual connection and directing the bytes to the appropriate `MultiplexingInputStreams`. `MultiGroupInputThread` can process all NIO sockets in its JVM. Since some socket factories, notably SSL socket factories, do not create NIO sockets, `SingleGroupInputThread` exists to process a single non-NIO socket.

These parameters may be passed to the appropriate classes by putting them in a configuration `HashMap`, using the keys given in `org.jboss.remoting.transport.multiplex.Multiplex`, and passing the map to a `VirtualSocket`, a `MasterServerSocket`, or a `VirtualServerSocket`. It may be passed either through a constructor or a call to `setConfiguration()`. **Note**, however, that the parameters have an effect **only** when a `MultiplexingManager` is first created, or to say the same thing differently, when a binding or connecting action leads to the creation of a virtual socket group. When a socket or server socket joins an existing socket group, or if `setConfiguration()` is called after a binding or connection action creates a new `MultiplexingManager`, the configuration map will have no effect.

7.1. Configuring MultiplexingManager.

Two aspects of the behavior of `MultiplexingManager` may be configured.

1. When a `MultiplexingManager` is created and it finds no other `MultiplexingManagers` in the JVM, it starts up several static threads. One of these threads periodically wakes up and monitors the existence of `MultiplexingManagers` in the JVM. If it wakes up two times in a row and finds no `MultiplexingManagers` in the JVM, it shuts down the other static threads.

2. When the last virtual socket supported by a particular `MultiplexingManager` closes, the `MultiplexingManager` will negotiate with its peer at the other end of the connection for permission to shut down, which will be withheld only if a virtual socket is being opened at the other end.

The following parameters affect the behavior of `MultiplexingManager`:

Name (in org.jboss.remoting.transport.multiplex.Multiplex)	Default value	Description
STATIC_THREADS_MONITOR_PERIOD:	30000 msec	Determines how often the monitor thread wakes up to look for the existence of `MultiplexingManagers` in the JVM.
SHUTDOWN_REQUEST_TIMEOUT:	5000 msec	When a `MultiplexingManager` requests permission from its remote peer to shut down, it will time out if it does not receive a reply within this period of time.
SHUTDOWN_REFUSALS_MAXIMUM:	5	When a `MultiplexingManager` requests permission from its remote peer to shut down, it will take "no" for an answer this many times before it assumes something is wrong and goes ahead and shuts down.
SHUTDOWN_MONITOR_PERIOD:	1000 msec	When a `MultiplexingManager` requests permission from its remote peer to shut down, it creates a `TimerTask` which periodically wakes up to see if and how the remote peer has responded. This parameter determines the period.

7.2. Configuring OutputMultiplexor

When `OutputMultiplexor` is passed some bytes by a `MultiplexingOutputStream`, it stores them in a `Message` data structure drawn from a pool of unused `Messages` and puts the `Message` on a queue. When the `OutputThread` gets the `Message` from the queue, it transmits some or all of its content, according to a set of fairness constraints. If the entire contents are not exhausted, the `Message` is returned to the queue.

The following parameters affect the behavior of `OutputMultiplexor`:

Name (in org.jboss.remoting.transport.multiplex.Multiplex)	Default value	Description
OUTPUT_MESSAGE_POOL_SIZE:	1024	This determines the maximum size of the pool of `Messages`. If the pool is empty when a transmission request is received, a new `Message` will be created, but when a `Message` has been emptied, it will be returned

		to the pool only if the pool has fewer than the maximum number of elements. Otherwise, the `Message` will be discarded.
OUTPUT_MESSAGE_SIZE:	256 bytes	This is the initial capacity of the `ByteArrayOutputStream` that holds the contents of a `Message`.
OUTPUT_MAX_CHUNK_SIZE:	2048 bytes	This determines the number of bytes transmitted by the `OutputThread` with a single `write()` call.
OUTPUT_MAX_TIME_SLICE:	500 msec	`OutputThread` will process a single virtual stream for this long before moving on to another stream.
OUTPUT_MAX_DATA_SLICE:	2048 * 8 bytes	`OutputThread` will transmit this many bytes for a single virtual stream before moving on to another stream.

7.3. Configuring InputMultiplexor.

The following parameters affect the behavior of `InputMultiplexor`:

Name (in org.jboss.remoting.transport.multiplex.Multiplex)	**Default value**	**Description**
INPUT_BUFFER_SIZE:	4096 bytes	Determines the size of the structure that holds bytes read from the real socket. The structure is a `ByteBuffer` for NIO sockets and a byte array for non-NIO sockets.
INPUT_MAX_ERRORS:	3	Both `MultiGroupInputThread` and `SingleGroupInputThread` count the number of non-fatal errors experienced on the socket(s) they manage. When this limit has been exceeded for a given socket, they will close the socket and throw an exception.

8. Performance.

It should come as no surprise that the classes in Muliplex perform more slowly than their non-virtual counterparts, since the multiplexing of data streams requires extra work. Multiplex uses two classes to perform input

and output multiplexing: `MultiplexingInputStream` and `MultiplexingOutputStream`, which are returned by the `VirtualSocket` methods `getInputStream()` and `getOutputStream()`, respectively. These classes subclass `java.io.InputStream` and `java.io.OutputStream` and reimplement all of their methods. Tests show that input/output by these classes is roughly four to five times slower than input/output by their counterpart classes used by actual sockets, `java.net.SocketInputStream` and `java.net.SocketOutputStream`. This information is gathered from multiple runs of three tests:

bare input:	compares the transmission of bytes from a `SocketOutputStream` to a `MultiplexingInputStream` with the transmission of bytes from a `SocketOutputStream` to a `SocketInputStream`
bare output:	compares the transmission of bytes from a `MultiplexingOutputStream` to a `SocketInputStream` with the transmission of bytes from a `SocketOutputStream` to a `SocketInputStream`
socket input/output:	compares the transmission of bytes from a `MultiplexingOutputStream` to a `MultiplexingInputStream` with the transmission of bytes from a `SocketOutputStream` to a `SocketInputStream`

Each of these tests was run 10 times, transmitting 100,000 bytes each time. Table 1 gives the factor by which the virtual socket version of each test was slower than the actual socket version.

Table 1. Factors by which virtual socket input/output is slower than actual socket input/output.

	bare input	**bare output**	**socket input/output**
minimum:	2.25	1.63	3.19
mean:	3.50	2.80	4.77
maximum:	4.42	4.67	8.58

9. APIs

One of the design goals of Multiplex is to make virtual sockets and their related classes as indistinguishable as possible from their real counterparts. There are two areas in which Multiplex is detectibly different.

1. The use of the two types of virtual server sockets entails an extra degree of complexity in setting up a multiplexed connection.

2. There are performance differences.

On the other hand, the virtual classes implement complete APIs, so that once a connection is established, a `VirtualSocket`, for example, can be passed to a method in place of a `Socket` and will demonstrate the same behavior. Similarly, `MultiplexingInputStreams` and `MultiplexingOutputStreams` are functionally indistinguishable from `SocketInputStreams` and `SocketOutputStreams`.

It may be useful, however, to be aware of some implementational differences between the two sets of classes. The public methods in the virtual classes can be placed in five categories.

1. methods implemented directly by the class
2. methods inherited from the real superclass
3. methods implemented by delegation to the underlying real socket
4. methods whose behavior is essentially null (though they may throw an `IOException` if called on a closed virtual socket)
5. methods which have no counterpart in the real class

Categories 3, 4, and 5 are particularly informative. Methods in category 3 can be used to fine tune a multiplexed connection by, for example, adjusting buffer sizes. Note that a method such as `setReceiveBufferSize()` may be called on any virtual socket in a socket group with the same effect as calling it on any other virtual socket in the same group. Methods in category 4 represent behavior that is not relevant to virtual sockets, and methods in category 5 represent behavior that is specific to the special nature of multiplexed connections. The category 5 version of `VirtualSocket.connect()`,

```
connect(SocketAddress remoteAddress, SocketAddress localAddress, int timeout)
```

exists to effect an atomic binding/connecting action to avoid the accidental connection problem discussed in the section on virtual socket groups. The notion of connection is irrelevant to ordinary server sockets, but `VirtualServerSocket` has methods

```
connect(SocketAddress remoteAddress, SocketAddress localAddress, int timeout)
```

and `isConnected()` because a connection must be established before `accept()` can function.

We also include in category 5 one of `VirtualServerSocket`'s nonstandard constructors, with the signature

```
VirtualServerSocket(InetSocketAddress remoteAddress, InetSocketAddress localAddress, int
```

which calls the two-address form of `connect()`.

The public methods of the main Multiplex classes are categorized in Table 2 and Table 3. The only inherited methods among the classes listed in Table 2 are found in `MasterServerSocket`, and we omit an explicit listing of them.

Note. The constructors of `VirtualServerSocket` that take a backlog parameter ignore its value. The same is true for methods of `VirtualServerSocketFactory`.

Table 2. Categories of public methods in the primary public Multiplex classes

category 1	close()	bind()	toString()
	connect()	close()	
	getInputStream()	getSoTimeout()	
	getOutputStream()	isBound()	
	getSoTimeout()	isClosed()	
	isClosed()	setSoTimeout()	
	isConnected()	toString()	
	isInputShutdown()		
	isOutputShutdown()		
	setSoTimeout()		
	shutdownInput()		
	shutdownOutput()		
	toString()		
category 3	getInetAddress()	getInetAddress()	
	getKeepAlive()/ setKeepAlive()	getLocalPort()	
	getLocalAddress()	getLocalSocketAddress()	
	getLocalPort()	getReceiveBufferSize() / setReceiveBufferSize()	
	getLocalSocketAddress()	getReuseAddress()/ setReuseAddress()	
	getPort()		
	getReceiveBufferSize() / setReceiveBufferSize()		
	getRemoteSocketAddress()		
	getReuseAddress()/ setReuseAddress()		
	getSendBufferSize()/ setSendBufferSize()		
	getSOLinger()/ setSOLinger()		
	getTCPNoDelay()/ setTCPNoDelay()		
	getTrafficClass()/ setTrafficClass()		

[a] This version of `connect()` is nonstandard in that it has both a local and remote address. It binds to a local address and connects to a remote address in a single atomic action.

[b]This constructor is nonstandard in that it has both a local and remote address. It binds to a local address and connects to a remote address in a single atomic action.

Table 3. Categories of public methods in the other public Multiplex classes

	MultiplexingInputStream	MultiplexingOutputStream	VirtualServerSocketFactory	VirtualSocketFactory
category 1	available()	close()	createServerSocket()	createSocket()
	close()	write()	getDefault()	getDefault()
	skip()			
	read()			
category 2	mark()			
	markSupported()			
	reset()			
category 4		flush()		
category 5			isOnClient()	
			isOnServer()	
			setOnClient()	
			setOnServer()	

10. Issues.

Please post issues and bugs to `http://jira.jboss.com/jira/browse/JBREM-91`.

11. Listings.

Listing 1. Client for Prime Scenario example.

```
public class PrimeScenarioExampleClient
{
   public void runPrimeScenario()
   {
      try {
         // Create a VirtualSocket and connect it to MasterServerSocket.
         Socket v1 = new VirtualSocket("localhost", 5555);

         // Do some asynchronous input in a separate thread.
         new AsynchronousThread(v1).start();

         // Do some synchronous communication.
```

```
                ObjectOutputStream oos = new ObjectOutputStream(v1.getOutputStream());
                ObjectInputStream ois = new ObjectInputStream(v1.getInputStream());
                oos.writeObject(new Integer(3));
                Integer i1 = (Integer) ois.readObject();
                v1.close();
             }
             catch (Exception e) {}
          }

          class AsynchronousThread extends Thread
          {
             private Socket virtualSocket;

             AsynchronousThread(Socket virtualSocket)
             {
                this.virtualSocket = virtualSocket;
             }

             public void run()
             {
                try {
                   // Create a VirtualServerSocket that shares a port with virtualSocket.
                   // (Note that it will be connected by virtue of joining a connected socket g
                   ServerSocket serverSocket = new VirtualServerSocket(virtualSocket.getLocalPo

                   // Create a VirtualSocket that shares a port with virtualSocket.
                   serverSocket.setSoTimeout(10000);
                   Socket v4 = serverSocket.accept();

                   // Get an object from the server.
                   v4.setSoTimeout(10000);
                   ObjectInputStream ois = new ObjectInputStream(v4.getInputStream());
                   Object o = ois.readObject();
                   serverSocket.close();
                   v4.close();
                }
                catch (Exception e) {}
             }
          }

          public static void main(String[] args)
          {
             new PrimeScenarioExampleClient().runPrimeScenario();
          }
       }
```

Listing 2. Server for Prime Scenario example.

```
       public class PrimeScenarioExampleServer
       {
          public void runPrimeScenario()
          {
             try {
```

```
            // Create a MasterServerSocket and get a VirtualSocket.
            ServerSocket serverSocket = new MasterServerSocket(5555);
            serverSocket.setSoTimeout(10000);
            Socket v2 = serverSocket.accept();

            // Do some asynchronous communication in a separate thread.
            Thread asynchronousThread = new AsynchronousThread(v2);
            asynchronousThread.start();

            // Do some synchronous communication.
            ObjectInputStream ois = new ObjectInputStream(v2.getInputStream());
            ObjectOutputStream oos = new ObjectOutputStream(v2.getOutputStream());
            v2.setSoTimeout(10000);
            Object o = ois.readObject();
            oos.writeObject(o);

            serverSocket.close();
            v2.close();
         }
         catch (Exception e) { }
      }

      class AsynchronousThread extends Thread
      {
         private Socket virtualSocket;

         public AsynchronousThread(Socket socket) throws IOException
         {this.virtualSocket = socket;}

         public void run()
         {
            try {
               // Connect to VirtualServerSocket.
               Thread.sleep(2000);
               String hostName = virtualSocket.getInetAddress().getHostName();
               int port = virtualSocket.getPort();
               Socket v3 = new VirtualSocket(hostName, port);

               // Send an object to the client.
               ObjectOutputStream oos = new ObjectOutputStream(v3.getOutputStream());
               oos.writeObject(new Integer(7));

               oos.flush();
               v3.close();
            }
            catch (Exception e) {}
         }
      }

      public static void main(String[] args)
      {
         new PrimeScenarioExampleServer().runPrimeScenario();
      }
   }
```

Listing 3. Sample client for N-socket scenario.

```
public class N_SocketScenarioClient
{
   public void runN_SocketScenario()
   {
      try
      {
         // Create a VirtualServerSocket and connect it to the server.
         VirtualServerSocket serverSocket = new VirtualServerSocket(5555);
         InetSocketAddress connectAddress = new InetSocketAddress(“localhost”, 6666);
         serverSocket.setSoTimeout(10000);
         serverSocket.connect(connectAddress);

         // Accept connection requests for 3 virtual sockets.
         Socket socket1 = serverSocket.accept();
         Socket socket2 = serverSocket.accept();
         Socket socket3 = serverSocket.accept();

         // Do some i/o.
         InputStream is1 = socket1.getInputStream();
         OutputStream os1 = socket1.getOutputStream();
         InputStream is2 = socket2.getInputStream();
         OutputStream os2 = socket2.getOutputStream();
         InputStream is3 = socket3.getInputStream();
         OutputStream os3 = socket3.getOutputStream();
         os1.write(3);
         os2.write(7);
         os3.write(11);
         System.out.println(is1.read());
         System.out.println(is2.read());
         System.out.println(is3.read());

         socket1.close();
         socket2.close();
         socket3.close();
         serverSocket.close();
      }
      catch (Exception e) {}
   }

   public static void main(String[] args)
   {
      new N_SocketScenarioClient().runN_SocketScenario();
   }
}
```

Listing 4. Sample server for N-socket scenario.

```
public class N_SocketScenarioServer
```

```
{
   public void runN_SocketScenario()
   {
      try
      {
         // Create and bind a MasterServerSocket.
         MasterServerSocket serverSocket = new MasterServerSocket(6666);

         // Accept connection request from VirtualServerSocket.
         serverSocket.setSoTimeout(10000);
         serverSocket.acceptServerSocketConnection();

         // Create 3 virtual sockets
         Thread.sleep(2000);
         Socket socket1 = new VirtualSocket("localhost", 5555);
         Socket socket2 = new VirtualSocket("localhost", 5555);
         Socket socket3 = new VirtualSocket("localhost", 5555);

         // Do some i/o.
         InputStream is1 = socket1.getInputStream();
         OutputStream os1 = socket1.getOutputStream();
         InputStream is2 = socket2.getInputStream();
         OutputStream os2 = socket2.getOutputStream();
         InputStream is3 = socket3.getInputStream();
         OutputStream os3 = socket3.getOutputStream();
         os1.write(is1.read());
         os2.write(is2.read());
         os3.write(is3.read());

         socket1.close();
         socket2.close();
         socket3.close();
         serverSocket.close();
      }
      catch (Exception e) {}
   }

   public static void main(String[] args)
   {
      new N_SocketScenarioServer().runN_SocketScenario();
   }
}
```

Listing 5. Symmetric Scenario client.

```
public class SymmetricScenarioClient
{
   public void runSymmetricScenario()
   {
      try {
         // Get a virtual socket to use for synchronizing client and server.
         Socket syncSocket = new Socket("localhost", 6666);
         InputStream is_sync = syncSocket.getInputStream();
```

```
                OutputStream os_sync = syncSocket.getOutputStream();

                // Create a VirtualServerSocket and connect
                // it to MasterServerSocket running on the server.
                VirtualServerSocket serverSocket = new VirtualServerSocket(5555);
                InetSocketAddress address = new InetSocketAddress("localhost", 7777);
                is_sync.read();
                serverSocket.setSoTimeout(10000);
                serverSocket.connect(address);

                // Call constructor to create a virtual socket and make a connection
                // request to the port on the server to which the local VirtualServerSocket
                // is connected, i.e., to the remote VirtualServerSocket.
                os_sync.write(5);
                is_sync.read();
                int port = serverSocket.getRemotePort();
                Socket virtualSocket1 = new VirtualSocket("localhost", port);
                InputStream is1 = virtualSocket1.getInputStream();
                OutputStream os1 = virtualSocket1.getOutputStream();

                // Create a virtual socket with VirtualServerSocket.accept().
                Socket virtualSocket2 = serverSocket.accept();
                InputStream is2 = virtualSocket2.getInputStream();
                OutputStream os2 = virtualSocket2.getOutputStream();

                // Do some i/o and close sockets.
                os1.write(9);
                System.out.println(is1.read());
                os2.write(11);
                System.out.println(is2.read());
                virtualSocket1.close();
                virtualSocket2.close();
                syncSocket.close();
                serverSocket.close();
             }
             catch (Exception e) {}
          }

          public static void main(String[] args)
          {
             new SymmetricScenarioClient().runSymmetricScenario();
          }
       }
```

Listing 6. Symmetric Scenario server.

```
       public class SymmetricScenarioServer
       {
          public void runSymmetricScenario()
          {
             try {
                // Create ServerSocket and get synchronizing socket.
                ServerSocket ss = new ServerSocket(6666);
```

```
                Socket syncSocket = ss.accept();
                ss.close();
                InputStream is_sync = syncSocket.getInputStream();
                OutputStream os_sync = syncSocket.getOutputStream();

                // Create MasterServerSocket, accept connection request from remote
                // VirtualServerSocket, and get the bind port of the local actual
                // socket to which the VirtualServerSocket is connected.
                MasterServerSocket mss = new MasterServerSocket(7777);
                os_sync.write(3);
                mss.setSoTimeout(10000);
                int port = mss.acceptServerSocketConnection();
                mss.close();

                // Wait until remote VirtualServerSocket is running, then create local
                // VirtualServerSocket, bind it to the local port to which the remote
                // VirtualServerSocket is connected, and connect it to the remote
                // VirtualServerSocket.
                is_sync.read();
                VirtualServerSocket vss = new VirtualServerSocket(port);
                InetSocketAddress address = new InetSocketAddress("localhost", 5555);
                vss.setSoTimeout(5000);
                vss.connect(address);

                // Indicate that the local VirtualServerSocket is running.
                os_sync.write(7);

                // Create a virtual socket by way of VirtualServerSocket.accept();
                serverSocket.setSoTimeout(10000);
                Socket virtualSocket1 = vss.accept();
                InputStream is1 = virtualSocket1.getInputStream();
                OutputStream os1 = virtualSocket1.getOutputStream();

                // Call constructor to create a virtual socket and make a connection
                // request to the remote VirtualServerSocket.
                Socket virtualSocket2 = new VirtualSocket("localhost", 5555);
                InputStream is2 = virtualSocket2.getInputStream();
                OutputStream os2 = virtualSocket2.getOutputStream();

                // Do some i/o and close sockets.
                os1.write(is1.read());
                os2.write(is2.read());
                virtualSocket1.close();
                virtualSocket2.close();
                syncSocket.close();
                vss.close();
             }
             catch (Exception e) {}
          }

          public static void main(String[] args)
          {
             new SymmetricScenarioServer().runSymmetricScenario();
          }
       }
```

Listing 7. Sample use of VirtualServerSocketFactory and VirtualSocketFactory.

```
public class FactoryExample
{

   void runFactoryExample()
   {
      ServerSocketFactory serverSocketFactory = VirtualServerSocketFactory.getDefault();
      ((VirtualServerSocketFactory) serverSocketFactory).setOnServer();
      SocketFactory socketFactory = VirtualSocketFactory.getDefault();
      useServerSocketFactory(serverSocketFactory);
      useSocketFactory(socketFactory);
   }

   void useServerSocketFactory(final ServerSocketFactory serverSocketFactory)
   {
      new Thread()
      {
         public void run()
         {
            try
            {
               ServerSocket serverSocket = serverSocketFactory.createServerSocket(5555);
               Socket socket = serverSocket.accept();
               int b = socket.getInputStream().read();
               socket.getOutputStream().write(b);
               socket.close();
               serverSocket.close();
            }
            catch (Exception e)
            {
               e.printStackTrace();
            }
         }
      }.start();
   }

   public void useSocketFactory(SocketFactory socketFactory)
   {
      try
      {
         Thread.sleep(1000);
         Socket socket = socketFactory.createSocket("localhost", 5555);
         socket.getOutputStream().write(7);
         System.out.println(socket.getInputStream().read());
         socket.close();
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }

   }
```

```
    public static void main(String[] args)
    {
       new FactoryExample().runFactoryExample();
    }
}
```